Carl von Ossietzky
Universität Oldenburg

Fachbachelor Informatik, zur Erlangung des akademischen Grades Bachelor of Science (B. Sc.)

Bachelorarbeit

Datenbankentwurf und Optimierung einer Software zur Simulation des Ausbreitungsverhaltens von Mücken

vorgelegt von
Stephan Adolf

Betreuende Gutachterin:
Dr. Ute Vogel

Zweiter Gutachter:
Prof. Dr. Michael Sonnenschein

Oldenburg, 4. Oktober 2016

Datenbankentwurf und Optimierung einer Software zur Simulation des Ausbreitungsverhaltens von Mücken

Stephan Adolf

4. Oktober 2016

Zusammenfassung

Das Ziel dieser Bachelorarbeit ist es ein Lösungsverfahren für Zellulare Automaten zur Simulation des Ausbreitungsverhaltens von Mücken bzgl. der Rechenzeit zu optimieren und eine effiziente Anbindung an eine räumliche Datenbank zu erstellen. Dazu wurde eine erweiterbare Architektur entworfen, welche die Dateneingabe, -ausgabe sowie den Simulationskern in verschiedene Softwarekomponenten trennt.

Um die Rechenzeit bei dünn besetzten Zellularen Automaten zu senken, wurde ein Stapelspeicher zur Verwaltung der aktiven Bereiche des Zellularen Automaten verwendet. Es hat sich herausgestellt, dass dieses neue Verfahren bessere Ergebnisse erzielt, als wenn alle Zellen des Automaten neu berechnet werden.

Bei der Nutzung von Rasterdaten einer räumlichen Datenbank zur persistenten Speicherung der Simulationsergebnisse ist festzustellen, dass diese eine Vielzahl an funktionalen Möglichkeiten wie bspw. die Nutzung von unterschiedlichen Koordinatensystemen oder die Anbindung an Geoinformationssysteme aufweisen. Im Vergleich zu Dateien ist insbesondere bei großen Zellularen Automaten eine verbesserte Ladeperformanz erzielt worden. Bei der Speicherung schnitt die Datenbank jedoch schlechter ab als Dateien.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ACA	Asymmetric Cellular Automaton
ADT	Abstrakter Datentyp
BSD	Berkeley Software Distribution
CA	Cellular Automaton
CPU	Central Processing Unit
DBMS	Datenbankenmanagementsystem
DPKG	Debian Package Manager
DTD	Document Type Definiton
GIS	Geoinformationssystem
GiST	Generalized Search Tree
GNU	GNU's not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
HACA	Hierarchical Asymmetric Cellular Automaton
IDE	Integrated Development Environment
LIFO	Last-In-First-Out
MAS	Multiagentensystem
MBB	Minimum Bounding Box
OGC	Open Geospatial Consortium
PPA	Personal Package Archive
RAM	Random-Access Memory
TPR	Time-Parameterized R-Baum
UI	User Interface

UML	Unified Modeling Language
XML	eXtensible Markup Language

Kapitel 1

Einleitung

In der Arbeitsgruppe Umweltinformatik an der Carl von Ossietzky Universität Oldenburg ist seit 2012 ein Werkzeug zur Simulation des Ausbreitungsverhaltens von Mücken [59] entwickelt worden. Dabei handelt es sich um einen Plugin namens MosquitoCA für das freie Geoinformationssystem QGIS. Die Entwicklung des Prototyps geht dabei auf die Bachelorarbeit von Daniel Klich aus dem Jahr 2012 zurück. Ziel der Arbeit ist es gewesen, ein Softwaresystem zu entwickeln, mit dem das Ausbreitungsverhalten von eingeschleppten Neozoen computergestützt untersucht werden kann [59, S. 177].

In der vorliegenden Arbeit soll das bestehende Werkzeug verbessert werden. Folgende Ziele sollen erreicht werden:

- Der Simulationsalgorithmus des Werkzeugs soll effizienter gestaltet werden, sodass unterschiedlich dicht besetzte Zellulare Automaten möglichst effizient ausgewertet werden können.
- Die Erweiterbarkeit des Werkzeugs bzgl. Datenein- und ausgabe soll verbessert werden.
- Die Datenausgabe soll in eine räumliche Datenbank erfolgen. Es ist dabei insbesondere wichtig, dass effizient auf die Daten zugegriffen werden kann. Ferner sollen die Daten aus der Datenbank komfortabel in QGIS eingelesen werden können.
- Der Simulationskern soll aus dem bisherigen Werkzeug herausgelöst werden, sodass die Simulation unabhängig von QGIS durchgeführt werden kann. Ggf. ist eine Neuimplementierung notwendig.

Für diese Arbeit ist das Entwickeln oder Integrieren einer graphischen Nutzungsoberfläche nicht relevant. Visualisierungen von Simulationsergebnissen in QGIS oder

anderen Drittprogrammen erfolgen in dieser Bachelorarbeit nur zu Testzwecken.

Diese Bachelorarbeit beginnt mit einem Kapitel über räumliche- und temporale Datenbanken (s. Kapitel 2), welches inhaltlich dem Forschungsseminarvortrag, der am 12.05.2016 im Oberseminar Umweltinformatik gehalten wurde, entspricht.

Im anschließenden Kapitel 3 werden die für diese Arbeit wesentlichen theoretischen Grundlagen (insbesondere Zellulare Automaten und das MALCAM-Modell) sowie die relevanten Teile das bisherigen Werkzeugs besprochen.

Im Kapitel 4 erfolgt die Analyse der Problemstellung. Dabei wird das zu entwickelnde System definiert. Anschließend werden die Anforderungen aufgestellt.

Der Entwurf des zu implementierenden Werkzeugs wird in Kapitel 5 diskutiert. Dabei liegt der Fokus auf der Verbesserung der Simulation, der Datenhaltung und der Architektur, welche es ermöglichen soll, dass das Werkzeug unabhängig von QGIS nutzbar ist.

Daran anschließend werden in Kapitel 6 Implementierungsaspekte und -Entscheidungen vorgestellt, wobei mögliche Erweiterungen des Werkzeugs auf Implementationsebene angesprochen werden.

Das Testkapitel 7 dokumentiert exemplarisch die zur Qualitätssicherung durchgeführten Tests. In diesem Kapitel wird auch das Laden von Simulationsdaten aus der Datenbank in QGIS betrachtet.

In Kapitel 8 erfolgt die Evaluation des Werkzeugs. Dabei liegt der Fokus auf der Performanz der Berechnung des Zellularen Automaten. Ferner wird die Datenausgabe auf ihre Performanz untersucht.

Im abschließenden Kapitel 9 wird ein Fazit gezogen sowie ein Ausblick auf mögliche weitere Arbeiten an diesem Werkzeug gegeben.

Kapitel 2

Räumliche- und temporale Datenbanken

In diesem Kapitel werden grundlegende Technologien von räumlichen Datenbanken vorgestellt. Hierbei soll der Fokus auf der Verarbeitung von Rasterdaten liegen, da diese auch im MosquitoCA-Plugin verwendet werden. Ferner werden einige Index-Technologien vorgestellt, mit denen auf räumliche Daten effizient zugegriffen werden kann. In einem weiteren Unterkapitel werden kurz (spatio-)temporale Datenbanken vorgestellt, dabei werden auch einige Alternativen zu PostgreSQL genannt, welche für die weitere Bachelorarbeit von Bedeutung werden könnten.

Dieses Kapitel entspricht inhaltlich dem Forschungsseminarvortrag über räumliche- und temporale Datenbanken, wie er am 12.05.2016 im Oberseminar Umweltinformatik an der Carl von Ossietzky Universität Oldenburg gehalten worden ist.

2.1. Was sind Räumliche Datenbanken?

Räumliche Datenbanken oder *Geodatenbanken* (engl. *spatial databases* oder *geodatabases*) dienen dem Speichern, Analysieren und Verarbeiten von geografischen Daten. Unter *geografischen Daten* sind solche Daten zu verstehen, welche einen räumlichen Bezug aufweisen (z. B. Wetterdaten). Wie bei den meisten Datenbanken, werden auch bei Geodatenbanken Datenbankenmanagementsysteme (DBMS) eingesetzt [22, S. 19]. Diese dienen der Verwaltung von Datenbanken [22, S. 17 - 19] und [110, S. 5]. Geodaten sind jedoch wegen ihres räumlichen Bezugs deutlich komplexer als alphanumerische Daten, wie sie bspw. in Webanwendungen vorkommen. Aus diesem Grund werden bei Geodatenbanken andere Technologien für die Speicherung und Indexierung eingesetzt [14, S. 1]. Als Beispieldatenbank wird in dieser Bachelorarbeit PostGIS [99] verwendet, welche eine freie Erweiterung der ebenfalls freien Datenbank PostgreSQL [112] um geografische Daten und Funktionen ist. Wie schon oben angedeutet, werden bei räumlichen Datenbanken komplexere Datentypen benötigt, als dies bei „normalen“ Datenbanken der Fall wäre.

2.2. Rasterdaten

2.2.1. Was sind Rasterdaten?

Rasterdaten zeichnen sich dadurch aus, dass das zweidimensionale Universum in kleine Zellen unterteilt wird, welche gemäß einem Gitter angeordnet sind (vgl. Abbildung 2.1). Alle Zellen sind gleich groß und die Position im Universum ist eindeutig bestimmt. Jede dieser Zellen repräsentiert ein Tupel, welches einen oder mehrere Werte im höher dimensionalen Raum speichert (z. B. Temperatur, Windrichtung, Windgeschwindigkeit, ...). Die *Auflösung* hängt jedoch stark von der Anzahl der Zellen ab. Für hohe Auflösungen ist jedoch mit einem verhältnismäßig hohen Speicheraufwand zu rechnen. Daher sind Rasterdaten für Große Universen schlecht geeignet. Allerdings sind in vielen Fällen durch die Datenerfassung Rasterdaten gegeben (z. B. Satellitenbilder) [9, S. 31 f.].

2.2.2. Unterscheidung von Raster- und Vektordaten

Bei *Vektordaten* werden räumliche Objekte als Punkte und Vektoren gespeichert. Bei Letzterem werden die Verläufe von eindimensionalen Objekten und die Umrisse

von zweidimensionalen Objekten mittels Vektoren gespeichert [65, S. 347 f.]. Dadurch lassen sich Vektordaten im Gegensatz zu Rasterdaten unabhängig von ihrer Auflösung darstellen [103, S. 77]. In der Rastervisualisierung Abbildung 2.1 können die Umrisse der Figur im Gegensatz zur Vektrodatendarstellung Abbildung 2.2 nur unzureichend aufgelöst werden.

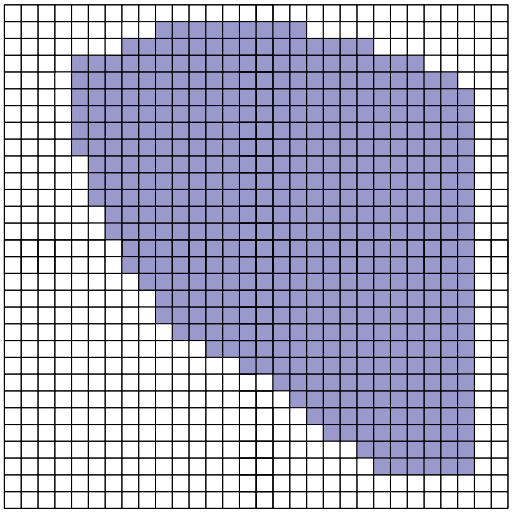


Abbildung 2.1.: Beispiel für visualisierte Rasterdaten.

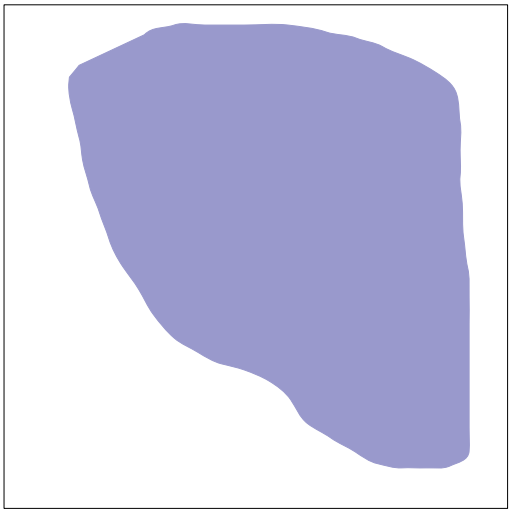


Abbildung 2.2.: Beispiel für visualisierte Vektordaten.

2.2.3. Rasterdaten in PostGIS

In PostGIS werden die folgenden Datentypen unterstützt [76, S. 14 - 17]:

Geometry ist im zweidimensionalen Raum eine Oberklasse der Datentypen Punkt, Linie und Polygon. Damit lässt sich bspw. die Erde als „Scheibe“ modellieren. Diese Beschreibungsart ist bspw. für Atlanten nützlich. Hierbei handelt es sich um einen sogenannten *Vektordatentyp*.

Topology ist ein *Vektordatentyp* welcher das Universum als Netzwerk modelliert. Topology kann in PostGIS mit dem add-on *pgRouting* als Routenplaner genutzt werden.

Raster modelliert das Universum als ein Gitter und wird in dieser Bachelorarbeit verwendet. Daher ist eine genauere Darstellung in Abschnitt 2.2.1 zu finden.

2.3. Indexierungstechnologien räumlicher Datenbanken

In modernen Datenbanken werden *Indexe* verwendet, um Suchanfragen schnell verarbeiten zu können. Dabei handelt es sich um eine interne Speicherstruktur, die eine Ordnung auf den Daten definiert [117, S. 65 f.]. Bei vielen verfügbaren Datenbanken werden B*- oder B$^+$-Bäume verwendet [104, S. 242 f.]. Diese Speicherstrukturen lassen sich sehr gut auf alphanumerische Daten anwenden, da hier eine vollständige Ordnung definiert ist. Wie oben beschrieben, zeichnen sich räumliche Daten durch einen mehrdimensionalen räumlichen Bezug aus. Daher ist auf Geodaten keine vollständige Ordnung definiert. Folglich ist bei Geodatenbanken ein Einsatz von B*-Bäumen nicht möglich. Wie soll bspw. ein Vergleich von Ort(Bremer Roland) > Ort(Brandenburger Tor) ausgewertet werden? Eine wichtige Anfrageform ist die sogenannte *Intervallanfrage*, dabei wird nach allen mehrdimensionalen Objekten gesucht, welche einen Schnitt mit dem zu suchenden Intervall aufweisen [14, S. 177]. Damit solche Anfragen effizient ausgeführt werden können, sollten Orte, welche räumlich benachbart sind, auch im Index nahe beieinander liegen. Das liegt daran, dass auf Festplatten Daten in Blöcken gespeichert werden. Dabei besteht ein Block aus mindestens einer Spur. Befinden sich räumlich benachbarte Daten physikalisch auf der Festplatte sehr weit voneinander entfernt, so kommt es beim Zugriff zu einem großen Spurwechsel des Schreib-/Lesekopfes der Festplatte. Das verursacht eine große Latenzzeit, wodurch der Datenzugriff ineffizient wird [14, S. 175 f.]. Wegen der Mehrdimensionalität sind bei räumlichen Daten B*- oder B$^+$-Bäume ungeeignet, da diese nur eindimensionale Daten indexieren können. Daher werden im Vergleich zu B*- oder B$^+$-Bäumen andere Technologien verwendet, von denen eine kleine Auswahl im Folgenden dargestellt wird, da diese für PostGIS und damit die weitere Bachelorarbeit von Bedeutung sind [101, S. 203 f.].

Um beliebig geformte Objekte im zweidimensionalen Raum sowie in Raumregionen verarbeiten zu können, wird die sogenannte *Minimum Bounding Box (MBB)* [101, S. 202] eingeführt. Dabei handelt es sich um das kleinstmögliche Rechteck, welches das Objekt bzw. die Raumregion umschließt. Ein Beispiel ist in Abbildung 2.3 zu sehen. Dabei wird die blau gezeichnete Fläche durch die MBB approximiert. Das Abspeichern der MBB benötigt weniger Speicherplatz als das Abspeichern des eigentlichen Objektes, da lediglich die minimal- und maximal Werte für die beiden Raumrichtungen benötigt werden.

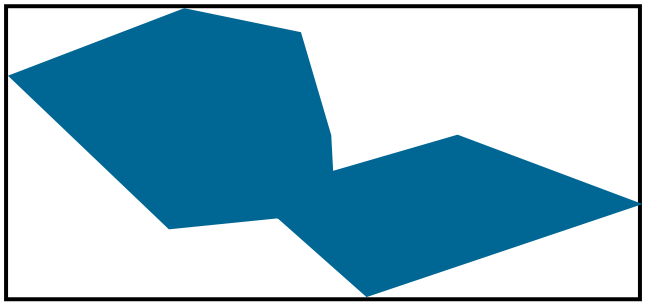
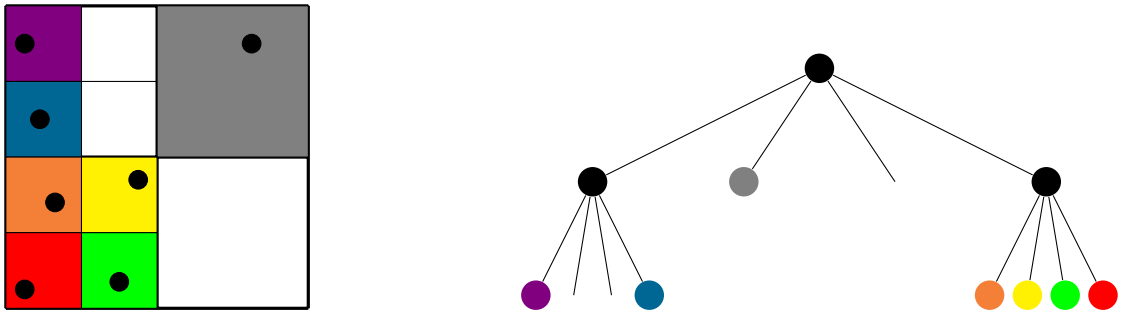


Abbildung 2.3.: Beispiel für eine *minimum bounding box* (MBB).

2.3.1. Quadtree

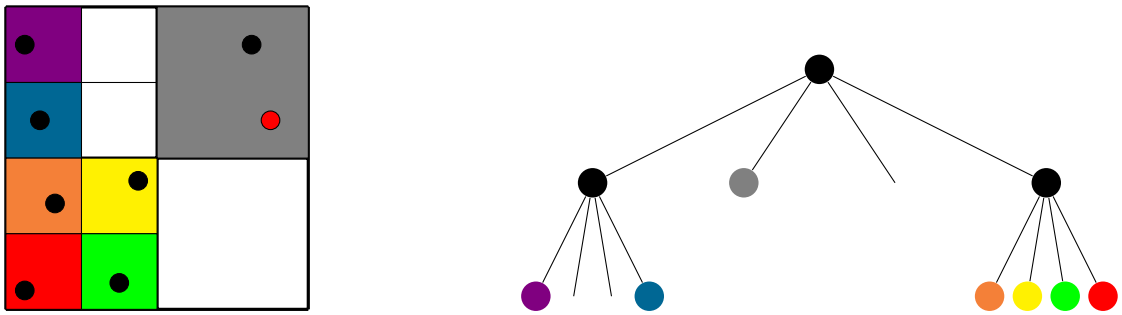
Der sogenannte *Quadtree* nach Finkel und Bentley [24], ursprünglich als Speichersystem für den Hauptspeicher entwickelt [28, Abschnitt 3.2], zeichnet sich dadurch aus, dass jeder innere Knoten im d-dimensionalen Fall genau 2^d-Nachfolger hat. Hierbei sei angemerkt, dass ein Quadtree nur im zweidimensionalen Raum vier Nachfolger pro inneren Knoten hat. Der Quadtree wird dadurch aufgebaut, dass das Universum durch *d-Hyperebenen* geteilt wird, welche orthogonal zueinander stehen und das Universum in gleichgroße Regionen aufteilt. Im zweidimensionalen Fall bilden die vier Regionen jeweils vier Knoten bzw. Unterbäume, die analog aufgebaut werden. Die vier Regionen im zweidimensionalen Fall werden i. A. nach den vier Himmelsrichtungen Nordwest, Nordost, ... benannt. Die Daten sind ausschließlich in den Blättern abgespeichert. Dabei enthält jedes Blatt eine maximale Anzahl an Objekten. Das führt dazu, dass Quadtrees nicht notwendigerweise balanciert sein müssen. Aufgrund der Struktur im Zweidimensionalen eignet sich der Quadtree als Datenstruktur zur Modellierung von Rasterdaten. Wie in Abbildung 2.4(a) dargestellt, befinde sich je genau ein Objekt in den farbig markierten Feldern. Jeder den farbig markierten Feldern des Universums zugeordnete Blattknoten in Abbildung 2.4(b) enthält dabei maximal ein Objekt. Das Suchen beginnt an der Wurzel. Es wird getestet, in welchem der Nachfolger sich das zu suchende Objekt befindet. Anschließend wird rekursiv mit dem entsprechenden Nachfolgeknoten fortgefahren, bis ein Blatt erreicht worden ist. Bei Punktanfragen ist zu bemerken, dass immer maximal ein Nachfolgeknoten bzw. Blatt gefunden wird. Bei Bereichsanfragen muss das nicht immer der Fall sein. [31, S. 74 f.]



(a) Modelliertes Universum. (b) Zugehöriger Quadtree.

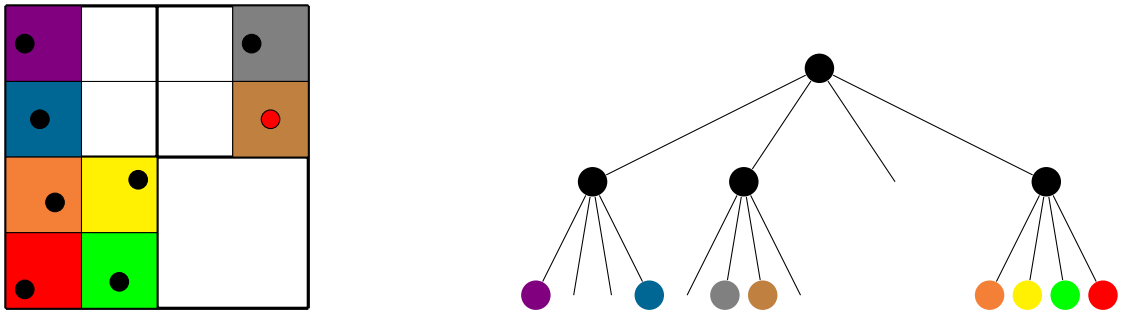
Abbildung 2.4.: Beispiel eines Quadtrees [adapiert nach 14, S. 195].

Bei der Einfügeoperation wird von der Wurzel aus das entsprechende Blatt gesucht. Danach wird dieses ggf. solange geteilt, bis das neue Objekt eingefügt werden kann. In Abbildung 2.5(a) wird das rot markierte Objekt eingefügt. Da die grau markierte Region schon ein Objekt enthält, wird eine Teilung dieser Region erforderlich. Entsprechend wird im Baum der grau markierte Blattknoten zu einem inneren Knoten, welcher auf die Blätter mit den eigentlichen Daten verweist. Das Ergebnis ist in Abbildung 2.6 zu sehen. Die Löschoperation funktioniert analog, wobei das zu löschende Objekt aus dem Baum entfernt wird und anschließend getestet wird, ob eine Teilung rückgängig gemacht werden kann. [14, S. 195 f.]



(a) Modelliertes Universum. (b) Zugehöriger Quadtree.

Abbildung 2.5.: Einfügen des rot markierten Objektes in den Quadtree führt zu einem Überlauf [adapiert nach 14, S. 195].



(a) Modelliertes Universum.

(b) Zugehöriger Quadtree.

Abbildung 2.6.: Überlaufbehandlung während des Einfügens des rot markierten Objektes in den Quadtree durch Teilung der Region mit zwei neuen Hyperebenen [adapiert nach 14, S. 195].

2.3.2. Grid File

Nach Nievergelt et al. [74] können mit *Grid Files* mehrdimensionale Punkte abgespeichert werden. Dabei handelt es sich um eine Art Hashing, wobei die Hashfunktion durch ein Grid Directory (s. u.) übernommen wird. Hierbei wird das Universum in ein Gitter partitioniert. In Abbildung 2.7 wird das Gitter durch sogenannte *Partitionierungslinien* symbolisiert. Die einzelnen Partitionen werden dabei *Zellen* genannt. Jede dieser Zellen ist dabei genau einem *Datenblock* zugeordnet, in dem die einzelnen Daten gespeichert werden. Um die Einteilung des Universums vorzunehmen und zu speichern, werden sogenannte *lineare Skalen* verwendet. Dabei wird für jede Dimension des Gitters genau eine lineare Skala definiert, welche die Koordinaten der Partitionierungslinien speichern. Technisch können diese lineare Skalen bspw. als Array realisiert werden. I. A. können die linearen Skalen aufgrund ihrer verhältnismäßig geringen Größe im Hauptspeicher gehalten werden. Das *Grid Directory* stellt die Zuordnung der Zellen zu den eigentlichen Datenblöcken her. Für eine schnelle Suche muss sich die Reihenfolge der Einträge des zugehörigen Speicherarrays an der Durchlaufreihenfolge der Zellen orientieren (s. Abbildung 2.8). Hierzu wird eine dynamische Speicherstruktur benötigt, um einen effizienten Zugriff zu ermöglichen. Die Originalpublikation von Nievergelt et al. [74] lässt die Details dieser Struktur offen. Auf neuere Ansätze kann in diesem Kapitel nicht eingegangen werden. Das Grid Directory sowie die Datenblöcke sind wesentlich größer als die linearen Skalen und werden daher zumeist nicht im Hauptspeicher gehalten. Die Speicherstruktur der Datenblöcke wird als Buckets bezeichnet, wie die Buckets intern aufgebaut sind ist nach [74, S. 44] relativ unbedeutend. Ähnlich wie bei baum-

basierten Verfahren speichern die Buckets Verweise auf die eigentlichen Daten. [62, S. 92 f.], [14, S. 206], [74, S. 46] und [31, S. 77 f.]

Ein Datenblock darf nicht leer sein und kann nur eine begrenzte Kapazität speichern. Heuristisch sollten Datenblöcke maximal bis etwa 70 % ihrer Kapazität belegt werden, da sich ansonsten die Performanz verschlechtert [74, S. 53].

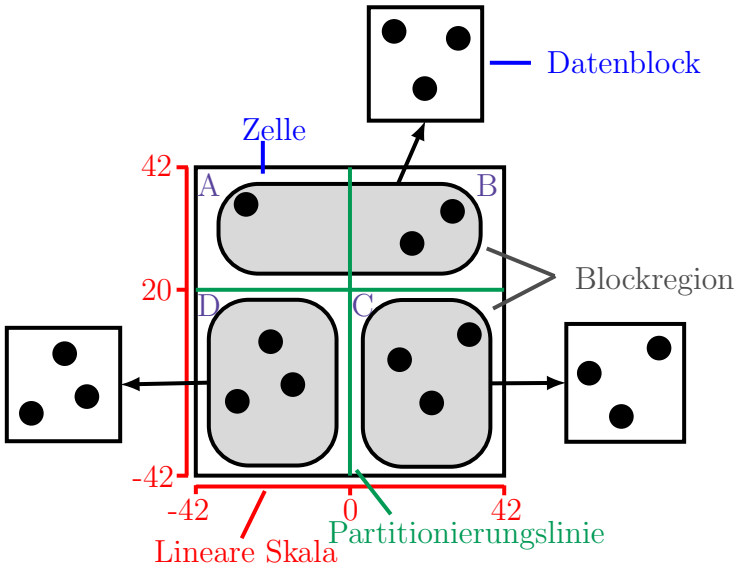


Abbildung 2.7.: Beispiel eines Grid Files [adapiert nach 62, S. 93].

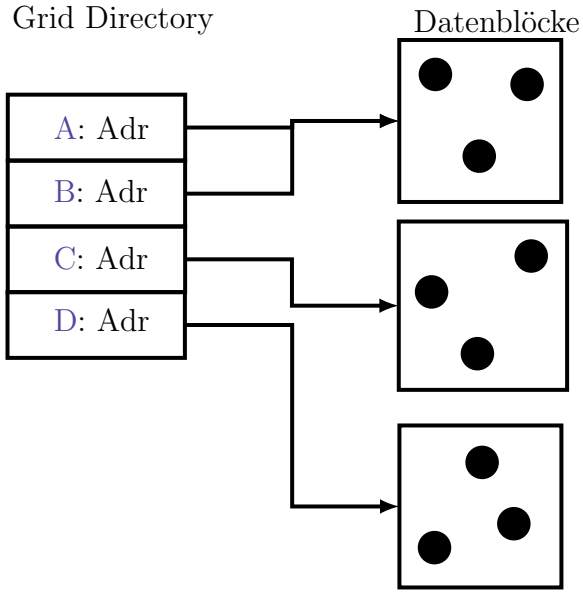


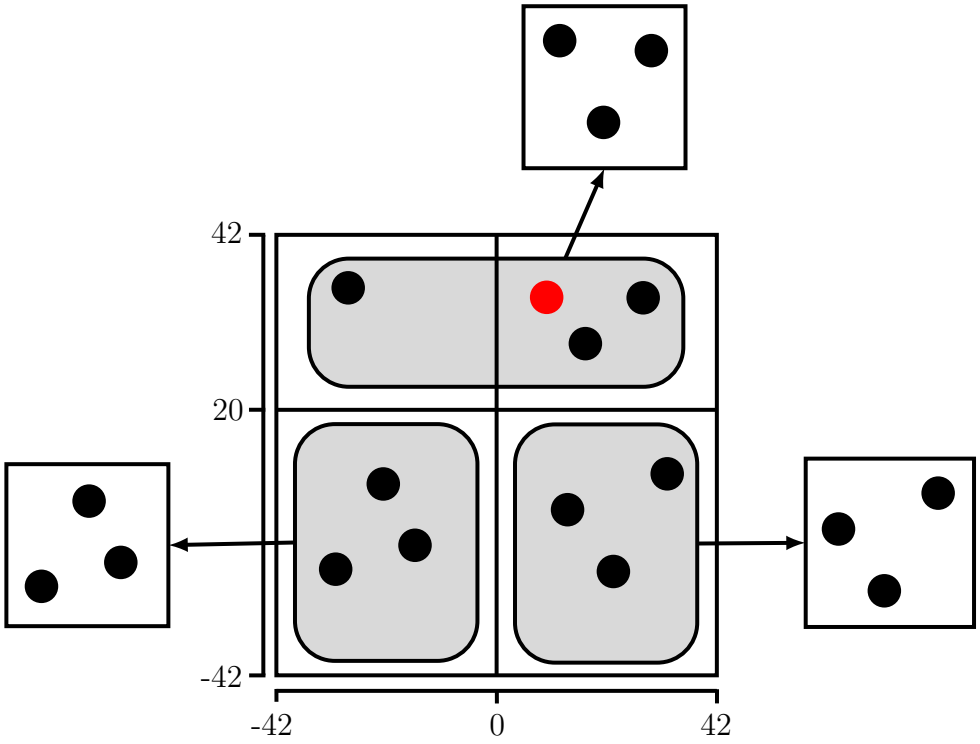
Abbildung 2.8.: Zuordnung des Grid Directories zu den Datenblöcken aus Beispiel Abbildung 2.7.

Suche

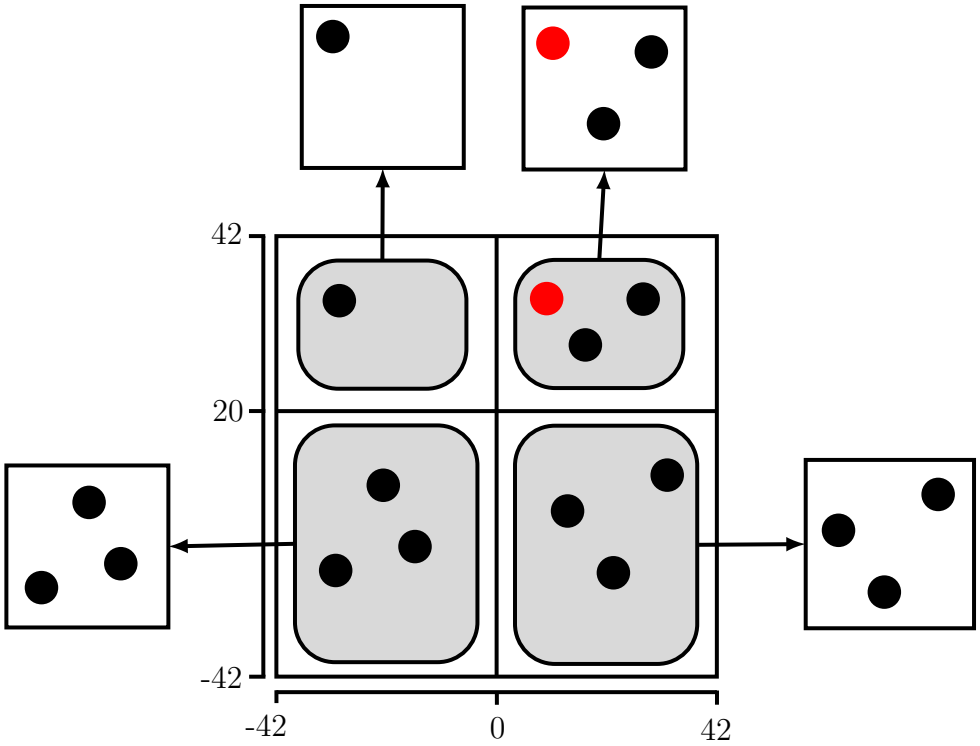
Datenzugriffe werden mit Hilfe der linearen Skalen durchgeführt. Mit den linearen Skalen werden dabei zunächst die betreffenden Blockregionen, die einen Schnitt mit dem Anfrageintervall bzw. Objekt aufweisen, bestimmt. Die entsprechenden Einträge des Grid Directories werden bestimmt und in den Hauptspeicher geladen. Das Grid Directory speichert die Adressen der korrespondieren Datenblöcke zu den Zellen. Danach kann mit den ausgelesenen Adressen auf die eigentlichen Datenblöcke zugegriffen werden. Während bei Intervallanfragen u. U. mehrere Datenblöcke gefunden werden, handelt es sich bei Punktzugriffen um maximal einen Datenblock [62, S. 92 f.].

Einfügen

Um neue Daten zu speichern, wird die entsprechende Zelle mit Hilfe der linearen Skalen bestimmt. Wenn im adressierten Datenblock nicht genügend Platz zum Abspeichern vorhanden ist, gibt es mehrere Möglichkeiten fortzufahren. Die erste Möglichkeit kann genutzt werden, wenn die korrespondierende Blockregion sich über mehrere Zellen erstreckt (vgl. Abbildung 2.9(a)). Dann kann der Block gemäß der vorhandenen Partitionierungslinien aufgeteilt werden (vgl. Abbildung 2.9(b)). Entsprechend werden neue Datenblöcke gebildet (es dürfen jedoch keine leeren Blöcke entstehen). Läuft jedoch ein Datenblock über, der nur von einer Zelle adressiert wird (vgl. Abbildung 2.10(a)), so wird eine neue Partitionierungslinie eingefügt. Es müssen neue Adressfelder gemäß dem Hashing eingefügt werden, sodass u. U. vorhandene Adressen umkopiert werden. Dieses kopieren ist eine verhältnismäßig teure Operation, sodass dass Einfügen relativ ineffizient ist. Abschließend werden die Objekte so auf die Datenblöcke verteilt, dass keine leeren Datenblöcke entstehen (vgl. Abbildung 2.10(b)) [62, S. 93 f.] und [14, S. 207].

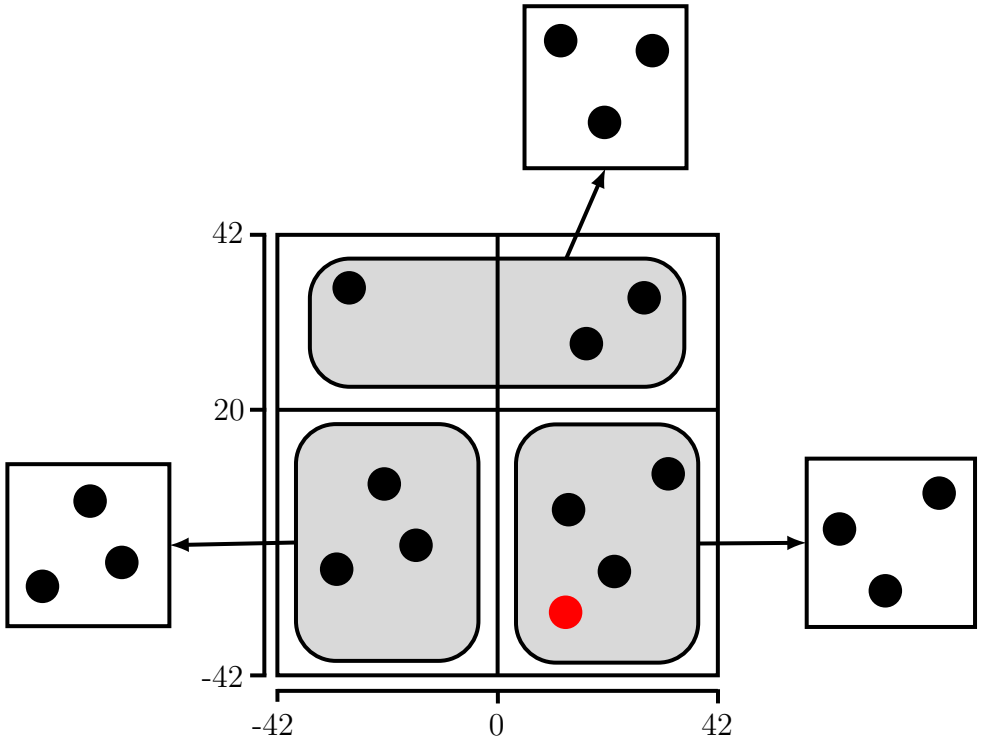


(a) Einfügen des rot markierten Objektes führt zum Überlauf der entsprechenden Blockregion.

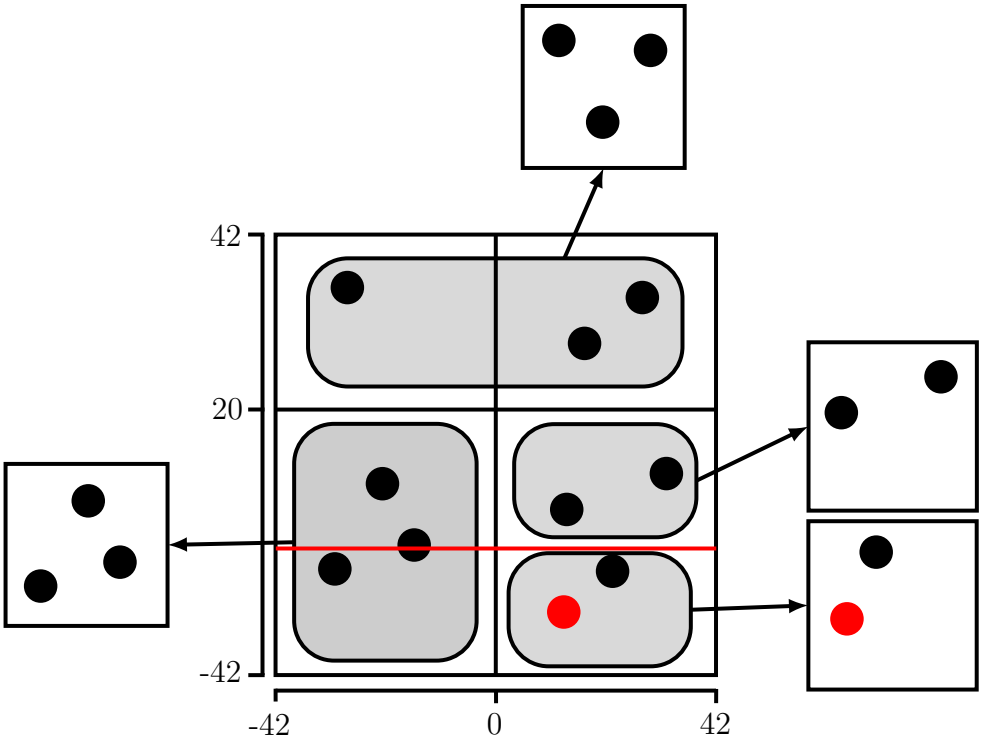


(b) Aufteilen der Blockregion an der vorhandenen Partitionierungslinie.

Abbildung 2.9.: Einfügen eines Objektes in das Grid File, wobei der Überlauf mittels Aufteilung der Blockregion anhand einer vorhandenen Partitionierungslinie gelöst wird [adapiert nach 62, S. 94].



(a) Einfügen des rot markierten Objektes führt zum Überlauf der entsprechenden Blockregion.



(b) Aufteilen der Blockregion durch eine neue Partitionierungslinie (rot markiert).

Abbildung 2.10.: Einfügen eines Objektes in das Grid File, wobei der Überlauf mittels Aufteilung der Blockregion gelöst wird [adapiert nach 62, S. 94].

Löschen

Wird ein Objekt aus dem Grid File gelöscht, so muss überprüft werden, ob verschiedene Blockregionen zu einer verschmolzen werden können. Wie oben angesprochen muss das Grid Directory gemäß dem Hashing angeordnet sein. Daher werden Adressen ggf. ähnlich wie beim Einfügen umkopiert. Entsprechend ist auch die Löschoperation relativ ineffizient. Die Details lassen sich bei Nievergelt et al. [74, Abschnitt 4.3] finden.

2.3.3. R-Baum

R-Bäume sind nach Guttman [30] Speicherstrukturen um räumliche Daten indexieren zu können. Sie zeichnen sich dadurch aus, dass ihre Knoten Rechtecke im d-dimensionalen Raum repräsentieren. Im Folgenden werden alle Beispiele aus praktischen Gründen im zweidimensionalen Raum aufgeführt. Die Größe bzw. Anzahl an Einträgen für die Knoten wird so angepasst, dass ein Knoten einer Seite des genutzten Speichermediums (i. A. einer Festplatte) entspricht. Der Grund ist, dass i. A. nicht der gesamte Suchbaum im Hauptspeicher gehalten werden kann. Daher müssen u. U. während des Suchens Seiten vom Hintergrundspeicher in den Hauptspeicher geladen werden. Entspricht jeder Knoten genau einer Seite des Hintergrundmediums, so wird für jeden Knoten maximal eine Seite neu geladen. Dadurch kann die Anzahl an Festplattenzugriffen minimiert werden. Bei modernen Computern ist der Festplattenzugriff der Flaschenhals für den Prozessor, sodass sich durch eine Verringerung der Zugriffsanzahl eine verbesserte Performanz ergibt [80, S. 341]. Für die Anzahl an Einträgen pro Knoten τ gelte

$$m \leq \tau \leq M, \tag{2.1}$$

wobei m,

$$0 \leq m \leq \frac{M}{2}, \tag{2.2}$$

die untere Schranke und M die obere Schranke seien [101, S. 238]. Der Wurzelknoten ist entweder ein Blatt, oder besitzt immer mindestens zwei Einträge. Für die

maximale Anzahl an Einträgen pro Knoten gilt

$$M = \left\lfloor \frac{size(p)}{size(E)} \right\rfloor , \tag{2.3}$$

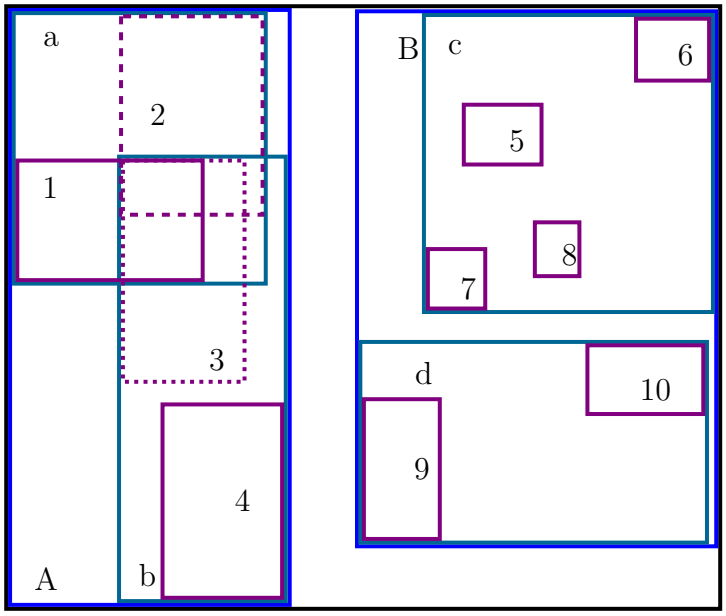
wobei $size(p)$ die Seitenkapazität und $size(E)$ die Eintragsgröße seien [101, S. 239].

Es wird bei R-Bäumen zwischen *inneren Knoten* und *Blattknoten* unterschieden. Die eigentlichen zu speichernden Objekte werden von den Blättern aus referenziert. Die Einträge der inneren Knoten enthalten eine Information über das Rechteck, welches sie repräsentieren, sowie eine Knoten-ID. Blatteinträge speichern hingegen anstelle der Knoten-ID die Objekt-ID des gespeicherten Objektes ab. [101, S. 237 – 239]

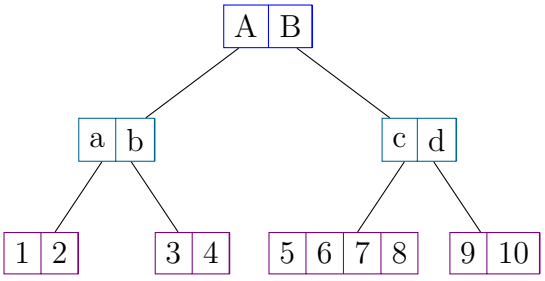
Da R-Bäume höhenbalanciert sind, gilt für die maximale Höhe eines R-Baumes $t_{\max}$ mit $N > 1$ Einträgen [31, S. 96]

$$t_{\max} = \lceil \log_m(N) \rceil . \tag{2.4}$$

Angenommen es gelte $m = 2$ und $M = 4$. Dann könnte ein Universum wie in Abbildung 2.11(a) aussehen. Hierbei wurden Unterteilungen des Universums vorgenommen, dabei werden die oben angesprochenen MBB-Eigenschaften ausgenutzt. Das bedeutet, dass die eingeführten Rechtecke so klein wie möglich sind. Die verschiedenen Farben in Abbildung 2.11(a) symbolisieren hierbei die verschiedenen Stufen des zu konstruierenden Baumes. Der R-Baum wird in Abbildung 2.11(b) angegeben. Wie zu erkennen ist, ist der Baum balanciert, d. h. alle Blätter befinden sich auf gleicher Höhe. [14, S. 216]



(a) Modelliertes Universum.

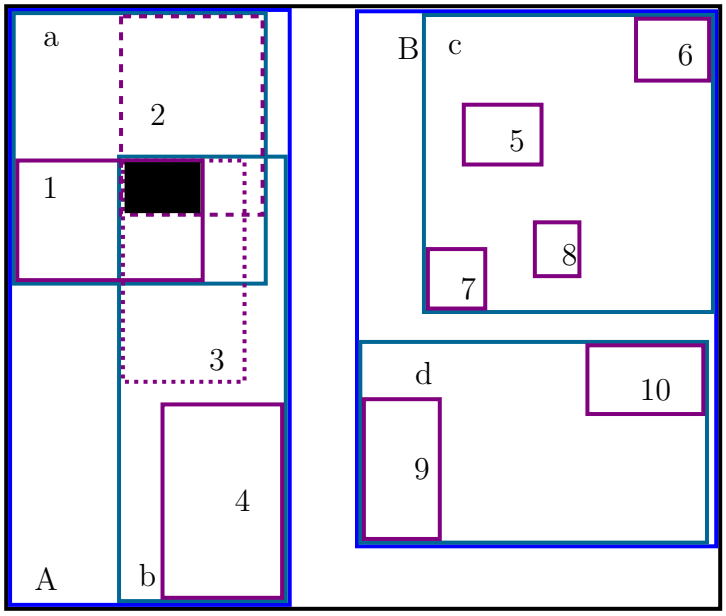


(b) Zugehöriger R-Baum.

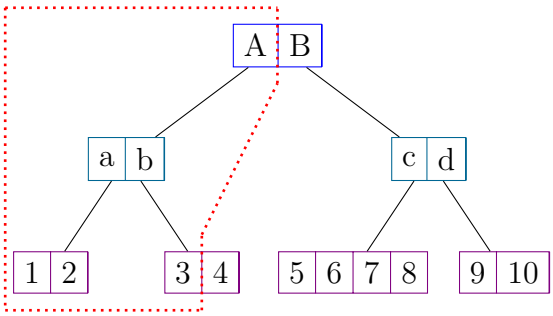
Abbildung 2.11.: Beispiel eines R-Baums [adapiert nach 14, S. 216].

Suche

Um ein Objekt oder eine Raumregion (vgl. schwarz markierte Fläche in Abbildung 2.12(a)) zu suchen, wird bei der Wurzel (vgl. Abbildung 2.12(b)) getestet, welcher Eintrag zu dem Anfrageobjekt einen nicht leeren Schnitt aufweist. Mit den entsprechenden Unterbäumen wird analog verfahren, bis die Blattebene erreicht ist. Da in jeder Stufe mehrere Einträge einen Schnitt mit dem zu suchenden Objekt aufweisen können, kann es vorkommen, dass mehrere Blätter gefunden werden. In Abbildung 2.12(b) wird der durchlaufene Teil des R-Baumes rot umrandet dargestellt. [31, S. 96 f.] und [30, S. 48 f.]



(a) Die schwarz markierte Fläche im Universum soll gesucht werden.



(b) Durchlaufene Pfade des R-Baums.

Abbildung 2.12.: Suche einer Fläche im R-Baum [adapiert nach 14, S. 216].

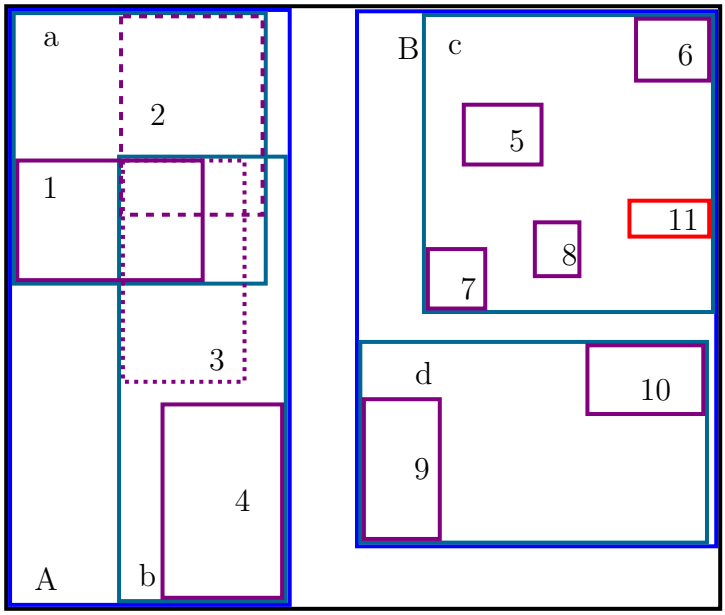
Einfügen

Um ein neues Objekt einzufügen, wird zunächst das entsprechende Blatt gesucht [28, S. 206]. Dabei wird für jeden besuchten Knoten geprüft, welcher der zu den Einträgen gehörenden Rechtecken im Universum die kleinste Vergrößerung benötigt, um das einzufügende Objekt zu enthalten und die geforderte MBB-Eigenschaft zu erfüllen. Für den Fall, dass es mehrere solcher Einträge gibt, wird derjenige ausgewählt, dessen zum Universum gehörendes Rechteck den kleinsten Flächeninhalt aufweist. Diese Heuristiken werden verwendet, um sicher zu stellen, dass das einzufügende Objekt genau einmal eingefügt wird [30, S. 49 f.]. Falls das betreffende Blatt weniger als M-Einträge hat, wird das neue Objekt eingefügt. Unter Umständen muss der

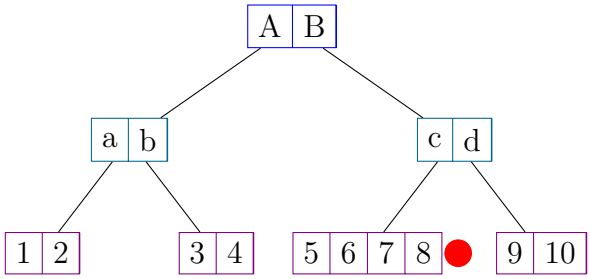
zugehörige Blockeintrag vergrößert werden. Diese Änderung muss ggf. bis zur Wurzel propagiert werden [27, S. 44]. Dadurch wird sichergestellt, dass das Objekt genau einmal eingefügt wird.

Falls es nicht möglich ist das Objekt einzufügen, weil es zu einem Überlauf im Blatt kommt, wird das Blatt so aufgeteilt, dass die Blockgrößen möglichst klein sind. Diese Änderungen werden wieder bis zur Wurzel propagiert [27, S. 44]. In Abbildung 2.13 soll als neues Objekt Nummer 11 eingefügt werden. Wie in Abbildung 2.13(a) zu erkennen ist, liegt dieses neu einzufügende Objekt in der Region c. Das korrespondierende Blatt im zugehörigen Baum Abbildung 2.13(b) enthält jedoch schon vier Einträge. Daher muss das Blatt aufgeteilt werden, wobei die Einträge 6 und 11 dem Vaterknoteneintrag c zugeordnet werden und die verbleibenden Einträge einem neu anzulegendem Vaterknoteneintrag e zugeordnet werden. Der Vaterknoteneintrag c muss anschließend verkleinert werden, um die MBB-Eigenschaft wiederherzustellen. Sollte der Knoten mit den Einträgen c, e und d überlaufen, so ist analog wie hier mit dem Blatt vorgestellt zu verfahren. Das ist notwendig, damit der Baum balanciert ist.

Das hier vorgestellte Beispiel soll nur das Prinzip des Aufteilens verdeutlichen. Für das Aufteilen der Knoten gibt es zwei verschiedene Algorithmen, die hier nicht vorgestellt werden, den *linearen Split-Algorithmus* und den *quadratischen Split-Algorithmus* [14, S. 219] und [30, S. 51 f.]. Ferner sind Varianten des R-Baumes wie bspw. der R*-Baum [6] entwickelt worden. Der Hauptunterschied liegt in einem anderen Knotenaufteilungsverfahren, welches die überlappenden Blockregionen möglichst klein hält. Auf R*-Bäume und weitere Varianten des R-Baum soll hier nicht detailliert eingegangen werden.

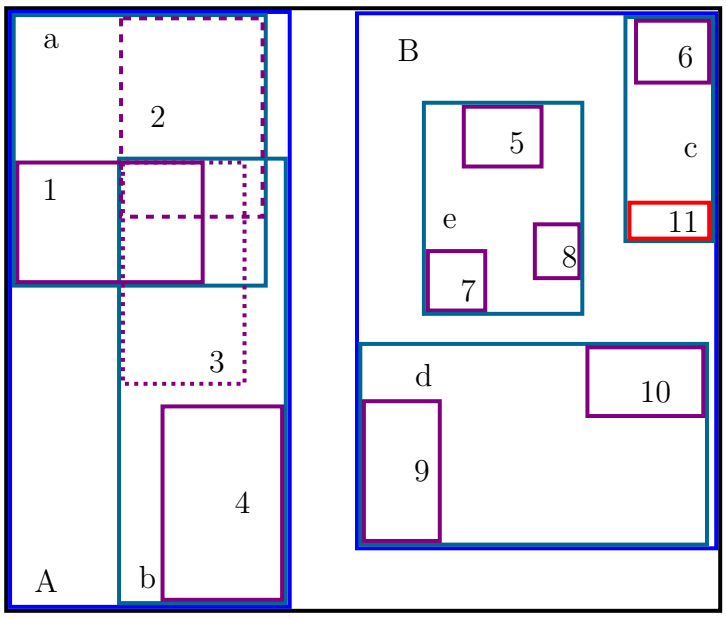


(a) Die rot makierte Fläche 11 soll eingefügt werden.

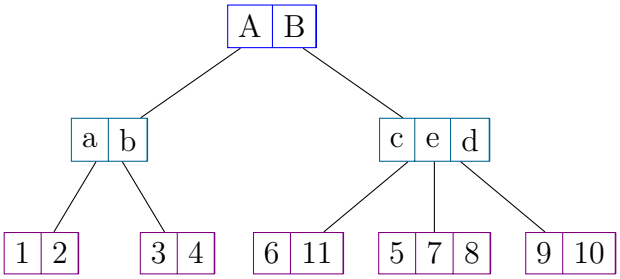


(b) Überlauf im R-Baum, durch Einfügung an der markierten Stelle.

Abbildung 2.13.: Einfügen einer Fläche im R-Baum [adapiert nach 14, S. 216].



(a) Aufteilung der Fläche c.



(b) Änderung des R-Baumes nach dem Aufteilen und Einfügen.

Abbildung 2.14.: Einfügen einer Fläche im R-Baum und Behebung des Überlaufs [adapiert nach 14, S. 216].

Löschen

Das Objekt wird nach der Suche aus dem entsprechenden Blatt gelöscht. Sind noch mindestens m Einträge in der entsprechenden Blockstruktur vorhanden, wird geprüft, ob die Blockstruktur verkleinert werden kann. Ggf. müssen die Änderungen bis zur Wurzel propagiert werden. Sollte es jedoch zu einem Unterlauf durch das Löschen kommen, werden zunächst alle nicht zu löschenden Objekte temporär aus dem Baum entfernt. Danach wird die entsprechende Blockregion im übergeordneten Knoten gelöscht und die Änderung bis zur Wurzel propagiert. Anschließend werden alle temporär aus dem Baum gelöschten Objekte abermals in den Baum eingefügt

[31, S. 97]. Prinzipiell sind für die Unterlaufbehandlung auch andere Strategien denkbar. Beispielhaft könnte die Unterlaufbehandlung ähnlich wie bei B-Bäumen durchgeführt werden. Allerdings sei der hier vorgestellte Algorithmus nach [30, S. 51] einfacher zu implementieren und keine signifikante Performanzverbesserung zu erwarten, da die entsprechenden Seiten für die Löschoperation im Hauptspeicher gehalten werden müssten.

2.3.4. GiST

Ein Vergleich von R-Bäumen mit B-Bäumen (bzw. deren Variationen, z.B. B^+, R^+, R^*, ...)[1], wie sie in vielen Datenbanken für alphanumerische Daten Verwendung finden, zeigt eine recht hohe Ähnlichkeit dieser Datenstrukturen. Sowohl R- als auch B-Bäume sind höhenbalanciert. Die inneren Knoten enthalten keine Verweise auf Daten, sondern nur die Blätter. Des Weiteren werden Überläufe beim Einfügen durch einen Split verarbeitet, der bis zur Wurzel propagiert wird. Analoges gilt für Unterläufe beim Löschen von Objekten. Mit den bisher dargestellten Mitteln müssten auf Implementierungsebene zwei gänzlich unabhängige Datenstrukturen vorgehalten werden, um in einer Datenbank sowohl alphanumerische- als auch räumliche Daten verarbeiten zu können. Dadurch kommt es zu einer verhältnismäß hohen Coderedundanz. Um dieses Problem zu beheben, kann der *Generalized Search Tree (GiST)* nach Hellerstein et al. [36] eingesetzt werden. Hierbei handelt es sich um einen *Abstrakten Datentypen (ADT)*. Das Ziel des Generalized Search Trees (GiST) ist es für die Implementierung von R- und B-Bäumen einen Abstrakten Datentypen bereitzustellen. Dadurch können „andere“ Indexe als Spezialfälle des GiSTs schnell erstellt werden. [40, S. 286 f.]

Der GiST ist mit folgenden Eigenschaften definiert, jeder innere Knoten hat $k \cdot M$ bis M Nachfolger, wobei k der sogenannte *Füllfaktor* ist, für den

$$\frac{2}{M} \leq k \leq \frac{1}{2} \tag{2.5}$$

gilt. Dadurch ergibt sich eine höhere Flexibilität [40, S. 287 f.]. Da der GiST eine Verallgemeinerung von B- und R-Bäumen ist, gilt für die Wurzel die Ausnahme 2 bis M Nachfolger. Entsprechend ist auch der verallgemeinerte Suchbaum *höhenbalanciert* [36, S. 103]. Die Methoden Suchen, Einfügen und Löschen funktionieren analog wie

[1] Für eine genaue Betrachtung von B-Bäumen und deren Variationen wird auf die einschlägige Literatur in den Bereichen Algorithmen und Datenstrukturen (z. B. [80, Kapitel 5.5]) sowie Datenbanken (z.B. [45, Kapitel 7.8 ff.]) verwiesen.

beim R-Baum und werden hier daher nicht erneut aufgeführt. Der große Unterschied zum R-Baum besteht darin, dass die genannten Methoden intern *abstrakte Methoden* aufrufen. Die abstrakten Methoden sind vorgegeben und müssen für jeden Datentypen einzeln konkretisiert werden, diese unterscheiden sich bspw. für R- und B-Bäume. Es werden vier abstrakte Methoden benötigt [35, S. 1223]:

Consistent prüft bei einer Suchanfrage, ob das zu suchende Objekt sich nicht im aktuellen Knoteneintrag befindet. Sollte das nicht sichergestellt werden, darf nicht `false` zurückgegeben werden.

Union vereinigt zwei oder mehrere Knoten und wird benötigt, wenn Daten verändert und insbesondere gelöscht werden.

Penalty berechnet für ein einzufügendes Objekt die Kosten es an der aktuellen Position einzufügen und gibt diese zurück. Auf Grundlage dieses Wertes entscheidet die Einfügeoperation, wo das Objekt in den Baum eingebaut wird.

PickSplit wird benötigt, um einen Knotenüberlauf beim Einfügen eines neuen Objektes zu verarbeiten. Die Methode teilt den übergebenen übergelaufenen Knoten und gibt das Ergebnis zurück.

In PostgreSQL wird GiST als Standard Index verwendet [82, S. 305].

2.4. Temporale- und spatiotemporale Datenbanken

2.4.1. Temporale Datenbanken

Temporale Datenbanken (engl. *temporal databases*) ermöglichen es zeitbezogene Daten zu speichern und zu indexieren [21, S. 819]. Hierbei wird nach [61, S. 82] bei Zeiteinträgen zwischen der *valid time* und der *transaction time* unterschieden. Die *valid time* bezeichnet dabei den Zeitpunkt (bzw. den Zeitraum), in dem das beschriebene Objekt den gespeicherten Zustand im modellierten Universum aufweist. Die *transaction time* gibt den Zeitpunkt an, an dem das Objekt letztmalig geändert worden ist. Nach [106, S. 20] werden bei *bitemporal data* beide vorherigen Zeitstempel gespeichert. Es können sowohl Zeitintervalle als auch -punkte gespeichert werden. Der SQL2-Standard sieht hierfür bspw. die Datentypen DATE, TIME, INTERVAL, TIMESTAMP und PERIOD vor [21, S. 821]. Dadurch können bspw. Anfragen bzgl.

dem Überlappen von Zeitintervallen beantwortet werden. Zeitpunktbasierte Anfragen lassen auf Standard-SQL-Mittel zurückführen. Auf die Anfragedetails soll hier nicht weiter eingegangen werden, diese sind bspw. bei [100, Kap. 5.4] zu finden.

2.4.2. Spatiotemporale Datenbanken

Datenbanken, die sowohl raum- als auch zeitbezogene Daten speichern und verarbeiten, werden *spatiotemporale Datenbanken* (engl. *spatiotemporal databases*) genannt. Um spatiotemporale Daten indexieren zu können, wurde von [124] der *Time-Parameterized R-Baum (TPR-Baum)* entwickelt. Analog zum R-Baum wird eine MBB eingeführt, die jedoch zeitparametrisiert ist. Um dabei sich bewegende Objekte erfassen zu können, werden für jede Dimension der MBB die Maximal- und Minimalwerte der Raumrichtung in Abhängigkeit von der Zeit sowie ein Geschwindigkeitsvektor gespeichert. Dabei werden die Bewegungen der Objekte als linear approximiert. Dadurch lassen sich sowohl die Positionen der Objekte als auch die Koordinaten der MBB in Abhängigkeit von der Zeit berechnen. Um nichtlineare Bewegungen modellieren zu können, müssen die gespeicherten MBB-Werte mit der Zeit geupdatet werden. Die Festlegung des Updateintervalls hängt dabei stark von den gegebenen Anforderungen ab. [124, S. 332 f.] und [14, S. 464]

Die Operationen wie Suchen, Einfügen, usw. entsprechen weitestgehend denen des oben vorgestellten R-Baums und werden daher an dieser Stelle nicht erneut vorgestellt [14, S. 464].

2.4.3. (Spatio-)Temporale Technologien in PostgreSQL

PostgreSQL bietet standardmäßig folgende Datentypen zur Speicherung von temporalen Fakten [82, S. 108 - 110]:

DATE kann alle Daten im Intervall 4713 vor Christi Geburt bis 32 767 nach Christi Geburt speichern und benötigt 4 Bytes Speicherplatz.

TIME speichert die Uhrzeit mit einer Genauigkeit von 1 µs[2], hierfür wird ein Speicherplatz von 8 Bytes benötigt. Optional kann TIME auch Informationen über die Zeitzone speichern. Dadurch erhöht sich der Speicherbedarf auf 12 Bytes.

[2] $1\,\mu s \mathrel{\widehat{=}} 1 \times 10^{-6}\,s$ [37, S. 125].

INTERVAL kann zum Abspeichern von Zeitintervallen von −178 000 000 Jahren bis 178 000 000 Jahren verwendet werden. Dieser Datentyp benötigt bei einer Genauigkeit von 1 µs 12 Bytes Speicherplatz.

TIMESTAMP speichert Datum und Zeit mit einer Genauigkeit von 1 µs im Intervall von 4713 vor Christi Geburt bis 1 465 001 nach Christi Geburt [112, S. 126][3]. Der Speicherbedarf beträgt 8 Bytes. Analog wie beim Datentyp TIME können auch hier optional Informationen zur Zeitzone gespeichert werden.

Mit Hilfe von Range Types lassen sich auch Zeiträume abspeichern. Hierfür stehen TSRANGE (TIMESTAMP ohne Zeitzone), TSTZRANGE (TIMESTAMP mit Zeitzone) sowie DATERANGE zur Verfügung [112, S. 169 f.].

Des Weiteren kann die Temporal Table Extension [122] für Versionierung mittels Transaktionszeit genutzt werden [29, S. 1919]. Dazu wird eine zusätzliche Tabelle verwendet, um alte Daten zu speichern.

In PostGIS 2.2 werden einige spatiotemporale Funktionen unterstützt, welche hier nicht im Detail diskutiert werden sollen. Stattdessen wird auf [98, S. 387 - 390] verwiesen.

2.4.4. Alternativen zu PostgreSQL

Wie oben angesprochen sieht der SQL2-Standard temporale Datentypen vor, daher können sowohl Datum als auch Uhrzeit in vielen relationalen Datenbanken gespeichert werden. Das Speichern und Indexieren temporaler Daten wird von relationalen Datenbanken zumeist nicht standardmäßig unterstützt [71, S. 165 - 168]. Allerdings sind für einzelne Datenbankensysteme entsprechende Erweiterungen verfügbar.

TerraLib [32, S. 127] ist eine Bibliothek für GIS Anwendungen und speichert die spatiotemporalen Daten in Relationalen Datenbanken wie PostgreSQL oder MySQL. Des Weiteren werden in MongoDB [72] und GeoMesa [25] Geohashes eingesetzt. GeoMesa ist hierbei eine spatiotemporale Datenbank, die bspw. auf Apache Accumulo, Apache HBase, Apache Cassandra und anderen aufbaut. Eine Beschreibung dieser Technologie ist in [25] zu finden. In Apache Hadoop werden für räumliche Anfragen R^+-Bäume verwendet [20, S. 48]. Um spatiotemporale Anfragen durchführen zu

[3] Nach [82, S. 108 - 110] bis 1 465 001 nach Christi Geburt, laut Dokumentation ist das jedoch falsch [112, S. 126].

können, führt [20, S. 50] einen zusätzlichen Index ein, der die Daten zunächst gemäß der Zeit aufteilt. Für jede der dadurch gebildeten Partitionen wird ein räumlicher Index definiert.

2.5. Ausblick

Seit dem Jahre 2012 ist in der Abteilung Umweltinformatik ein Plugin für das freie Geoinformationssystem QGIS namens MosquitoCA entwickelt worden [59]. MosquitoCA zielt darauf ab die Ausbreitung von eingeschleppten Mückenarten zu simulieren. Die Simulation wird intern mit Hilfe eines Zellularen Automaten durchgeführt. Hierbei werden die Daten aktuell in eine XML-Datei gespeichert. In dieser Arbeit ist das Ziel die Daten in einer PostgreSQL-Datenbank zu speichern. Es müssen dabei insbesondere Möglichkeiten geschaffen werden auf räumliche Rasterdaten effizient zuzugreifen, dazu soll PostGIS verwendet werden. Des Weiteren werden sowohl für die Simulation als auch für mögliche Anwendungen die Verläufe der einzelnen Datenzellen benötigt. Daher ist es erforderlich alle „alten“ Zustände des Zellularen Automaten zu speichern und effizient abrufen zu können. Dazu kann u. U. die oben angesprochene Temporal Table Extension von PostgreSQL genutzt werden. Ferner soll geprüft werden, inwiefern sich die Performanz der Simulation durch den Einsatz von Datenbankentechnologien verbessern lässt.

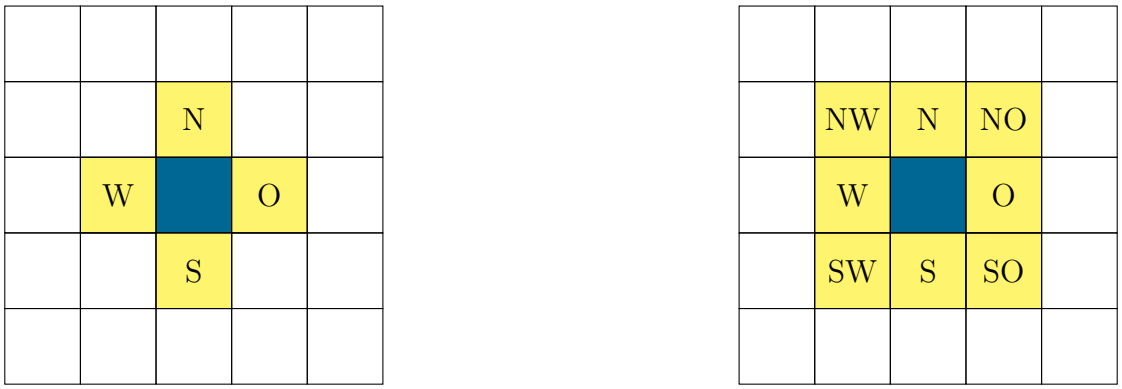
Kapitel
3

Theoretische Grundlagen und bisherige Arbeit

In diesem Kapitel werden zunächst die theoretischen Grundlagen der Simulation besprochen. Dabei sind insbesondere Zellulare Automaten sowie das MALCAM-Modell von Bedeutung. Mit Hilfe von Zellularen Automaten lassen sich diskrete dynamische Vorgänge simulieren. Dazu wird das Universum (in dieser Arbeit handelt es sich stets um ein zweidimensionales Rechteck) in kleine gitterartig angeordnete Zellen partitioniert. In jedem Zeitschritt wird für jede Zelle der neue Zustand aus dem Vorherigem sowie den Nachbarzellen berechnet. In dieser Arbeit wird zur Berechnung der Ausbreitung von Mücken das MALCAM-Modell angewendet, welches es ermöglicht die Anzahl der Mückenlarven sowie der erwachsenen Mücken mithilfe eines Zellularen Automaten zu berechnen.
Danach wird das bisherige MosquitoCA-Plugin für QGIS vorgestellt. Das Werkzeug führt dabei auf Grundlage des MALCAM-Modells und Zellularen Automaten Simulationen des Ausbreitungsverhaltens von Mücken durch.

3.1. Zellulare Automaten

Zellulare Automaten (engl. *Cellular Automaton* (*CA*)) werden eingesetzt um Simulationen über räumlich- und zeitlich diskretisierten Universen durchzuführen. Dabei wird das Universum in einzelne gitterartig angeordnete Zellen, ähnlich wie in Kapitel 2.2.1 für Rasterdaten beschrieben, unterteilt [8, S. 8 f.]. Im Folgenden werden alle Betrachtungen zur Vereinfachung im zweidimensionalen Raum vorgenommen, da auch im bisher implementierten Programm ein zweidimensionales Universum angenommen wird. Es sei angemerkt, dass prinzipiell auch andere Geometrien wie bspw. Zylinder denkbar wären. Die Wahl der Geometrie hängt stark von dem zu untersuchenden System ab [46, S. 12]. Des Weiteren werden hier quadratische Zellen angenommen. Jede dieser Zellen repräsentiert hierbei Werte (z. B. einen Spin im Ising-Modell), welche sich aus dem jeweiligen zu untersuchenden System ergeben. Um eine Simulation durchführen zu können, muss der Automat zunächst initialisiert werden. D. h. jeder Zelle wird ein Startzustand (bspw. Spin-Up bzw. Spin-Down) zugewiesen [46, S. 11]. Danach kann mit einer zeitdiskretisierten Simulation begonnen werden. Der Zustand einer beliebigen Zelle des Automaten zum Zeitpunkt t hängt dabei stark vom Zustand dieser Zelle zum Zeitpunkt $t-1$, sowie der Nachbarzellen ab. Prinzipiell wird wie in Abbildung 3.1 zu erkennen ist zwischen der Von-Neumann- und der Moore-Nachbarschaft unterschieden. Bei der *Von-Neumann-Nachbarschaft* werden nur die vier Nachbarn in die Simulation einbezogen, welche sich eine Seite mit der betrachteten Zelle teilen (vgl. Abbildung 3.1(a)). Die *Moore-Nachbarschaft* untersucht zusätzlich zu den vier Zellen der Von-Neumann-Nachbarschaft die Zellen, welche direkt an eine der vier Ecken der betrachteten Zelle angrenzen (vgl. Abbildung 3.1(b)). Damit werden bei der Von-Neumann-Nachbarschaft 4 und bei der Moore-Nachbarschaft 8 Nachbarzellen betrachtet.



(a) Von-Neumann-Nachbarschaft. (b) Moore-Nachbarschaft.

Abbildung 3.1.: Unterschiedliche Nachbarschaften (Radius 1) von Zellularen Automaten im zweidimensionalem Universum um die blau markierte Zelle [adapiert nach 81, S. 902].

Formal kann ein Zellularer Automat ZA als Quadrupel $ZA = (d, N, Z, \phi)$ beschrieben werden [19, S. 18]. Dabei sind die einzelnen Einträge des Quadrupels wie folgt definiert [19, S. 18]:

d gibt die Dimension des kartesischen Koordinatensystems an. Allgemein gilt $d \in \mathbb{N}$, in dieser Arbeit gilt jedoch immer $d = 2$. In diesem Koordinatensystem werden wie oben beschrieben die einzelnen Zellen eindeutig bestimmt.

N definiert für jede Zelle die Nachbarschaft.

Z definiert für jede Zelle eine endliche Zustandsmenge, wobei die Zustände im Laufe der diskretisierten Zeit angenommen werden.

ϕ definiert die Zustandsübergangsfunktion, die jede Zelle zur Zeit $t-1$ in den Zustand zur Zeit t überführt. Für eine ausführliche Definition s. Gleichung (3.1) sowie Gleichung (3.2).

Um eine Simulation durchführen zu können, wird eine Übergangsfunktion definiert, welche die Zustandsänderung im Zeitschritt $t-1 \rightarrow t$ beschreibt. [8, S. 8 f.] und [120, S. 48]

Der Zustand einer beliebigen Zelle x zum Zeitpunkt t lässt sich aus der Zelle x zum Zeitpunkt $t-1$, sowie der Nachbarzellen x_* mit einer Übergangsfunktion ϕ berechnen. $*$ bezeichnet die in Abbildung 3.1 angegebenen Namen der Nachbarzel-

len: $* \in \{NW, N, NO, O, SO, S, SW, W\}$ bzw. $* \in \{N, O, S, W\}$. Bei der Moore-Nachbarschaft gilt [41, S. 49]

$$x^t = \phi\left(x^{t-1}, x_{NW}^{t-1}, x_N^{t-1}, x_{NO}^{t-1}, x_O^{t-1}, x_{SO}^{t-1}, x_S^{t-1}, x_{SW}^{t-1}, x_W^{t-1}\right). \tag{3.1}$$

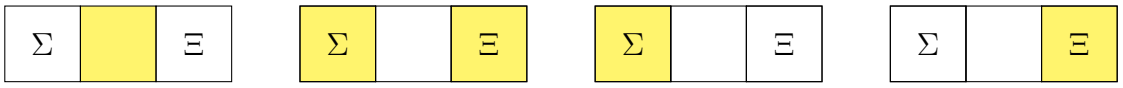
Entsprechend gilt für die Von-Neumann-Nachbarschaft [41, S. 49]

$$x^t = \phi\left(x^{t-1}, x_N^{t-1}, x_O^{t-1}, x_S^{t-1}, x_W^{t-1}\right). \tag{3.2}$$

Die hier angegebene Funktion ϕ muss für jedes zu untersuchende System implementiert werden. In dieser Arbeit wird als Regelsatz das MALCAM-Modell von Linard et al. [68] verwendet [59, S. 183]. Es wird in Abschnitt 3.3 beschrieben.

Prinzipiell kann die Auswertung der Zustandsübergangsfunktion auf die Zellen in einem Simulationsschritt sequenziell oder parallel erfolgen [46, S. 23].

Probleme können bei einzelnen mobilen Objekten auftreten, da diese sich mit einem einfachen Zellularen Automaten nicht optimal modellieren lassen. In dieser Bachelorarbeit wird eine relativ hohe Anzahl an mobilen Objekten (Mücken) betrachtet. Wird die Zustandsübergangsfunktion so definiert, dass bspw. α % der Mücken einer betrachteten Zelle diese verlassen und auf die Nachbarzellen verteilt werden, so treten die im Folgenden allgemein beschriebenen Probleme nicht auf. In Abbildung 3.2 sei ein Zellularer Automat gegeben, wobei sich in jeder Zelle maximal ein Objekt befinden soll. Die Übergangsregel sei so gewählt, dass die betrachtete Zelle prüft, ob sich in der Von-Neumann-Nachbarschaft im Zeitschritt $t-1$ Objekte befunden haben. Falls ja, wird eines ausgewählt zufällig auf das betrachtete Feld zu wechseln. In Abbildung 3.2(a) liegt ein Objekt in der Von-Neumann-Nachbarschaft der Zellen Σ und Ξ. Da während der Simulation nur die Von-Neumann-Nachbarschaft zum Zeitpunkt $t-1$ und *nicht* zum Zeitpunkt t betrachtet wird, kann die Massenerhaltung der mobilen Objekte nicht gewährleistet werden (vgl. Abbildung 3.2(b)). Gewünscht wäre, dass die Anzahl an Objekten auch nach einem Simulationsschritt gleich bleibt (s. Abbildung 3.2(c) und Abbildung 3.2(d)). Entsprechend müssen geeignete Gegenmaßnahmen ergriffen werden, um dieses Problem zu umgehen.



(a) Zeitpunkt $t-1$. (b) Zeitpunkt t (invalide). (c) Zeitpunkt t (valide). (d) Zeitpunkt t (valide).

Abbildung 3.2.: Mobile Objekte (gelb gefärbte Zellen) in einem Zellularen Automaten.

Anmerkung: Nach dem Simulationsschritt der Konfiguration 3.2(a) zum Zeitpunkt $t-1$ ist das Ergebnis die Konfiguration 3.2(b) zum Zeitpunkt t. Tatsächlich sollte das Ergebnis eine der Konfigurationen 3.2(c) oder 3.2(d) sein, weil sich die Anzahl an Objekten während der Simulation nicht verändern soll.

Des Weiteren müssen die Ränder u. U. gesondert betrachtet werden, da hier die in Abbildung 3.1 gezeigten Nachbarschaftsbeziehungen nicht erfüllt werden. Eine Lösungsmöglichkeit liegt in sogenannten *periodischen Randbedingungen*. Dabei wird angenommen, dass der obere Nachbar einer Zelle in der obersten Reihe die entsprechende Zelle in der untersten Reihe ist [46, S. 7]. Eine Alternative besteht darin die fehlenden Nachbarn als „Nullzellen“ zu behandeln, d. h. alle Werte dieser Zellen werden für alle Zeitschritte als Null behandelt [41, S. 41].

Ferner sind in der Literatur weitere Ansätze zu finden, um das hier aufgeführte Zellulare Automatenmodell zu erweitern. Santos et al. [102] bspw. führen einen *Dreilevel Zellularen Automaten* ein. Dieser enthält eine aquatische Phase, die Mücken-Schicht und eine Menschen-Schicht [102, S. 2]. Dadurch können bspw. Bewegungen von Menschen, welche Mücken über größere Distanzen transportieren, simuliert werden [102, S. 8]. Die Moore-Nachbarschaftseigenschaften müssen jedoch angepasst werden, so dass es zu Interaktionen zwischen den Schichten kommt. Daher hat die Mücken-Schicht jeweils neun Nachbarn aus der aquatischen Phase und der Menschen-Schicht. Entsprechend gelten diese Nachbarschaftseigenschaften umgekehrt [102, S. 2].

Das *Asymmetric Cellular Automaton (ACA)*-Modell von Sonnenschein und Vogel [108] bricht die reguläre Struktur der Zellen auf und führt dafür beliebig geformte Polygone als Zellen ein. Dadurch wird es ermöglicht die Umwelt genauer zu modellieren und geografische Gegebenheiten wie z. B. Flüsse oder Wälder können besser im Zellularen Automaten repräsentiert werden [108, S. 1 f.]. Eine Erweiterung des ACA-Modells ist das *Hierarchical Asymmetric Cellular Automaton (HACA)*-Modell von Sonnenschein und Vogel [109]. Das Modell ist hierarchisch aufgebaut, wobei jede Zelle einer höheren Ebene aus mehreren Zellen der Ebene darunter zusammengesetzt

ist. Dadurch könnten Bewegungen von Mücken durch den Menschen über größere Distanzen simuliert werden [109, S. 1 ff.].

3.2. Multiagentensysteme

Multiagentensysteme (MAS) zeichnen sich dadurch aus, dass sie ein System bilden, in dem verschiedene Agenten mit ihrer Umwelt interagieren können. Dabei hat jeder Agent seine eigene unvollständige Sichtweise und agiert autonom. Der Zustand eines jeden Agenten ändert sich auf Grundlage der ihm zur Verfügung stehenden Informationen [105, S. 14].

In der Literatur wird eine Vielzahl von Charakterisierungen für Agenten beschrieben. Eine Auflistung ist bei [19, Abschnitt 2.1.1] zu finden. Hier werden nur die beiden für diese Arbeit relevanten Agententypen aufgeführt. *Mobile Agenten* zeichnen sich dadurch aus, dass sie sich selbstständig in einem Universum bewegen können. Im Gegensatz dazu sind *immobile Agenten* hierzu nicht in der Lage und verharren somit immer an einem Ort [19, S. 6]. Als Beispiel könnte ein Wald (Universum) angesehen werden, in dem sich Füchse als mobile Agenten bewegen können. Bäume hingegen stehen für gewöhnlich immer an einem Ort und sind somit die immobilen Agenten in diesem Universum.

Nach [19, S. 25 ff.] weisen Multiagentensysteme und Zellulare Automaten Ähnlichkeiten auf und überschneiden sich bzgl. ihres Einsatzes. Werden die Zellen eines Zellularen Automaten als immobile Agenten aufgefasst, so ist der Zellulare Automat ein Sonderfall von Multiagentensystemen. Wie schon oben angesprochen lassen sich mobile Agenten jedoch nur unzureichend mit Zellularen Automaten modellieren, sodass nicht jedes Multiagentensystem ohne Weiteres als ein Zellularer Automat aufgefasst werden kann. Für eine Erweiterung des Zellularen Modells zur verbesserten Beschreibung von mobilen Agenten sei auf [19, S. 25 ff.] verwiesen.

3.3. MALCAM-Modell

Das MALCAM-Modell ist von Linard et al. [68] entwickelt worden, um eine mögliche Ausbreitung von Malaria in der Region Camargue im Süden Frankreichs simulieren zu können. Als potenzieller Malaria-Vektor ist dort die Mückenart Anopheles (Culicidae) verbreitet. Das MALCAM-Modell nutzt Zellulare Automaten und Multiagentensysteme. Dabei werden Mücken sowie Menschen als mobile Agenten und die

Geländezellen als immobile Agenten modelliert [68, S. 162]. Für die Beschreibung bspw. der Anzahl an Larven pro Zelle werden einige Gleichungen verwendet, welche im Folgenden aufgeführt werden sollen, ohne hierbei auf alle Details einzugehen. Eine genauere Darstellung findet sich bei [68].

Im MALCAM-Modell sind für eine Zelle x zum Zeitpunkt t die Anzahl der weiblichen Mücken a_x^t sowie die Anzahl an weiblichen Mückenlarven l_x^t die zentralen Variablen. Der *gonotrophische Zyklus* u gibt die Zeit zwischen zwei Blutmahlzeiten der Mücken in Tagen an und lässt sich mit Hilfe der Tagestemperatur ϑ in °C als

$$u = \left(\frac{36.5}{\vartheta - 9.9}\right) + 1 \tag{3.3}$$

berechnen [68, S. 167]. Für die Anzahl an weiblichen Mückenlarven l_x^t in der Zelle x zum Zeitpunkt t gilt

$$l_x^t = l_x^{t-1} + \left(\left[a_x^{t-1} \cdot \left\{\frac{3 \cdot P}{u}\right\} \cdot r\right] - \left[l_x^{t-1} \cdot 3 \cdot P \cdot d\right]\right) \cdot \left(1 - \frac{l_x^{t-1}}{l_{max}}\right), \tag{3.4}$$

wobei a_x^{t-1} die Anzahl an erwachsenen weiblichen Mücken in Zelle x zum Zeitpunkt $t-1$ und l_x^{t-1} die Larven zum Zeitpunkt $t-1$ seien [68, S. 167]. Dabei sei P die Länge eines Zeitschritts, wobei $P = \frac{1}{3}$ gilt, sodass $3 \cdot P$ immer genau ein Tag ist [68, S. 165]. Ferner sei r die Reproduktionsrate der Mücken, d die Entwicklungsrate der Larven und l_{max} die maximale Anzahl an weiblichen Mückenlarven in einer Zelle. Der Term $\left[a_x^{t-1} \cdot \left\{\frac{3 \cdot P}{u}\right\} \cdot r\right]$ beschreibt die geschlüpften Larven, während der Term $[l_x^{t-1} \cdot 3 \cdot P \cdot d]$ die im Vergleich zum vorherigen Zeitschritt erwachsen gewordenen Larven berechnet. $\left(1 - \frac{l_x^{t-1}}{l_{max}}\right)$ limitiert die Anzahl und Larven pro Zelle und sorgt für ein logistisches Wachstum der Larven [120, S. 35]. Die oben verwendete Entwicklungsrate der Larven d lässt sich zu

$$d = 0.021 \cdot \left(\exp\left[0.162 \cdot \{\vartheta - 10\}\right] - \exp\left[0.162 \cdot \{35 - 10\} - \left\{\frac{35 - \vartheta}{5.007}\right\}\right]\right) \tag{3.5}$$

berechnen und hängt hauptsächlich von der Wassertemperatur ϑ ab [68, S. 167]. Die Anzahl an erwachsenen weiblichen Mücken a_x^t von Zelle x zum Zeitpunkt t lässt sich mittels

$$a_x^t = a_x^{t-1} + \left(l_x^{t-1} \cdot 3 \cdot P \cdot d\right) - \left(a_x^{t-1} \cdot 3 \cdot P \cdot m\right) \tag{3.6}$$

berechnen [118] und [68, S. 168]. Dabei ist m die Sterberate erwachsener Mücken. Der Term $(l_x^{t-1} \cdot 3 \cdot P \cdot d)$ beschreibt analog wie oben die Larven, welche erwachsen geworden sind. $(a_x^{t-1} \cdot 3 \cdot P \cdot m)$ hingegen berechnet die Mücken, die im Vergleich zum letzten Zeitschritt verstorben sind. Die Anzahl der erwachsenen Mücken a_M, die nach neuer Blutnahrung suchen kann mit

$$a_M = a \cdot \frac{3 \cdot P}{u} \tag{3.7}$$

berechnet werden [68, S. 169]. Bei Linard et al. wird a nicht genauer spezifiziert. Aus dem Kontext ergibt sich, dass a die Anzahl an Mücken in einer Zelle x ist und a_M die Anzahl an Mücken ist, welche die Zelle x verlassen. Dabei wird die Bewegungsrichtung bei Linard et al. nicht genauer angegeben [68, S. 169]. Solange die Mücken keine Zelle mit Menschen gefunden haben, bewegen sie sich zufällig.

Linard et al. schreiben, dass das MALCAM-Model relativ gute Resultate liefert [68, S. 171 ff.].

3.4. Review des bisherigen MosquitoCA-Plugins

Das QGIS-Plugin MosquitoCA [49] von Daniel Klich im Rahmen der Bachelorarbeit „Entwicklung eines Software-Prototypen zur Modellierung des Ausbreitungsprozesses von Mückenarten“ [51] aus dem Jahr 2012 bildet den Ausgangspunkt dieser Arbeit. Daher wird das Plugin in diesem Abschnitt vorgestellt. Da in dieser Bachelorarbeit die Simulation sowie insbesondere die Datenspeicherung in einer Datenbank im Vordergrund stehen und diese unabhängig von QGIS funktionieren sollen, wird hier nicht auf alle Details des Plugins eingegangen. Relevant sind hier lediglich der Simulationskern sowie die Datenspeicherung in XML.

3.4.1. Modell von MosquitoCA

Das bisherige Plugin wurde nach dem Model-View-Controller-Pattern entworfen [51, S. 45]. Hier ist vor allem das Model von Interesse. Wie im Klassendiagramm Abbildung 3.3 zu erkennen ist, repräsentiert die Klasse `CellularAutomaton` [50] einen Zellularen Automaten. `CellularAutomaton` enthält ein Array mit ein- bis beliebig viele `Layer`. Da die Layer nicht untereinander interagieren, wird das Modell von Santos et al. [102] in der vorliegenden Version nicht umgesetzt [52]. Entsprechend speichert `CellularAutomaton` einige Attribute wie z.B. die Größe der Zellen und den Offset

(dieser beschreibt die Position des Zellularen Automaten relativ zum nullten Längen- und Breitengrad) [51, S. 46]. Des Weiteren startet `CellularAutomaton` die Simulation, speichert die XML-Bäume für die `Layer` und setzt diverse andere Attribute (die Details können im Quellcode [50] nachvollzogen werden). Die schon erwähnte Klasse `Layer` definiert eine „Schicht“ eines Zellularen Automaten, ein `Layer` enthält dabei Zellen (s. Klasse `Cell` [48]), welche in einem zweidimensionalem Array abgespeichert werden. Die Parameter wie Größe, Zellgröße und Offset in den Klassen `Layer` und `Cell` werden von der Klasse `CellularAutomaton` gesetzt und sind entsprechend für alle Schichten bzw. Zellen gleich. Des Weiteren besteht in dem hier besprochenen Werkzeug die Möglichkeit verschiedene Regelsätze für die Simulation zu nutzen. Um das zu ermöglichen enthält die Klasse `Layer` das `ruleset` Attribut vom Typ `Ruleset` [55]. `Ruleset` enthält dabei die anzuwendenden Regeln. Die Klasse `Rule` [54] bildet hierbei die Oberklasse für die erbenden Klassen `StdRule` [56], `ParseRule` [53] und `UserImplemented` [57]. `StdRule` implementiert dabei die im Abschnitt 3.3 vorgestellten Regeln des MALCAM-Modells. `ParseRule` bietet die Möglichkeit eigene Regeln über die GUI zu nutzen. Hierzu wird intern das Parser-Modul von Python genutzt um die Eingaben zu parsen [51, S. 47]. `UserImplemented` hingegen ist eine Oberklasse für Regeln, die vom Benutzer in Python implementiert werden können.

3.4.2. Simulation in MosquitoCA

Der Ablauf der Simulation wird in diesem Abschnitt beschrieben. Auch hier werden nur die für diese Arbeit relevanten Aspekte des Werkzeugs betrachtet. Abbildung 3.4 zeigt ein Aktivitätsdiagramm, welches den Verlauf der Simulation beschreibt. Wie zu erkennen ist, wird die Simulation für jedes `Layer`-Objekt durchgeführt [50]. In jedem Simulationsschritt wird zunächst das komplette Gitter kopiert. Anschließend werden für alle Regeln, die im jeweiligen `rules` Attribut des `Layer`s abgespeichert sind die Berechnung des neuen Zeitschritts durchgeführt. Dabei werden alle Zellen sequenziell nacheinander unabhängig davon, ob sich in der näheren Umgebung Mücken bzw. Larven befinden, abgearbeitet. Nach jedem Simulationsschritt wird geprüft, ob das aktuelle Gitter in den XML-Baum zur Datenspeicherung geschrieben werden soll [52].

Bei Verwendung der Standard-Regeln (MALCAM-Modell) wird für die Modellierung der Bewegungen der Mücken die Moore-Nachbarschaft mit Radius 1 verwendet [56]. Derzeit findet wie schon oben beschrieben kein Zugriff auf Zellen anderer Ebenen statt. Daher lässt sich diese Implementierung nicht ohne Weiteres analog zu

Santos et al. [102] nutzen.

3.4.3. Datenspeicherung in XML

Die Datenspeicherung erfolgt wie schon oben angesprochen in XML (eXtensible Markup Language) [51, S. 36 f.]. XML ist eine Struktur, die es ermöglicht komplexe Daten standardisiert in einer Textdatei abzuspeichern und ist sowohl menschen- als auch maschinenlesbar. Jede XML-Datei sollte mit einer Deklaration beginnen, welche auf die verwendete XML-Version sowie die verwendete Codierung (z. B. UTF-8) hinweist. Danach kann ein Kommentar eingefügt werden, der von XML-Parsern ignoriert wird, welcher weitere Informationen für menschliche Leser bereitstellt. Jedes Dokument enthält genau einen Wurzelknoten sowie beliebig viele Unterknoten [75, S. 19 ff.]. XML-Dateien gelten als „wohlgeformt“, wenn u. A. gilt, dass sie eine XML-Deklaration am Beginn des Dokuments aufweisen, genau einen Wurzelknoten besitzen und alle Knoten einen Start- und einen Endtag haben. Eine detailliertere und ausführlichere Beschreibung findet sich in [75, S. 26 ff.]. Zur Definition von XML-Schemata kann die *Document Type Definiton (DTD)* verwendet werden [75, S. 45 f.].

Wie der Beispieldatei in Listing 3.1 für ein 5×5-Gitter zu entnehmen ist, werden hierbei zum Einen die globalen Informationen und zum Anderen die Simulationsergebnisse abgespeichert. Das Wurzelelement ist hierbei **<simulation>**. Als globale Informationen (**<globalinfo>**) werden die Automatengröße, die Zellgrößen, der Offset, die Art der verwendeten Regeln sowie die Sterblichkeit und die Reproduktionsrate der Mücken gespeichert. Für die Simulationsergebnisse werden für jeden Tag (**<day >**) die Tagesnummer, die Temperatur und das Gitter mit allen Zellen ausgegeben. Dabei wird für jede Gitterzelle die Anzahl der weiblichen erwachsenen Mücken, die Anzahl der weiblichen Mückenlarven und die Brutstättenqualität ausgegeben. Die Brutstättenqualität nimmt Werte im Intervall $[0, 1]$ an [51, S. 52].

Listing 3.1: Gekürzte XML-Ausgabe eines Simulationstages [erzeugt mit 49].

```
<simulation>
  <globalinfo>
    <mortality>0.1</mortality>
    <reproduction>1.0</reproduction>
    <width>5</width>
    <height>5</height>
    <xoffset>8.96756</xoffset>
    <yoffset>48.777443</yoffset>
    <cellwidth>0.03</cellwidth>
    <cellheight>0.03</cellheight>
    <rules>integrated</rules>
  </globalinfo>
  <day>
    <dayofyear>2</dayofyear>
    <temperature>20</temperature>
    <grid>
      <cell>
        <adults>0</adults>
        <larvaes>0</larvaes>
        <hatchery>0.01</hatchery>
      </cell>
      ...
    </grid>
  </day>
</simulation>
```

Die ausgegebene XML-Repräsentation ist aufgrund der fehlenden XML-Deklaration nicht „wohlgeformt“. Des Weiteren wird der Startzustand am Tag eins der Simulation nicht gespeichert.

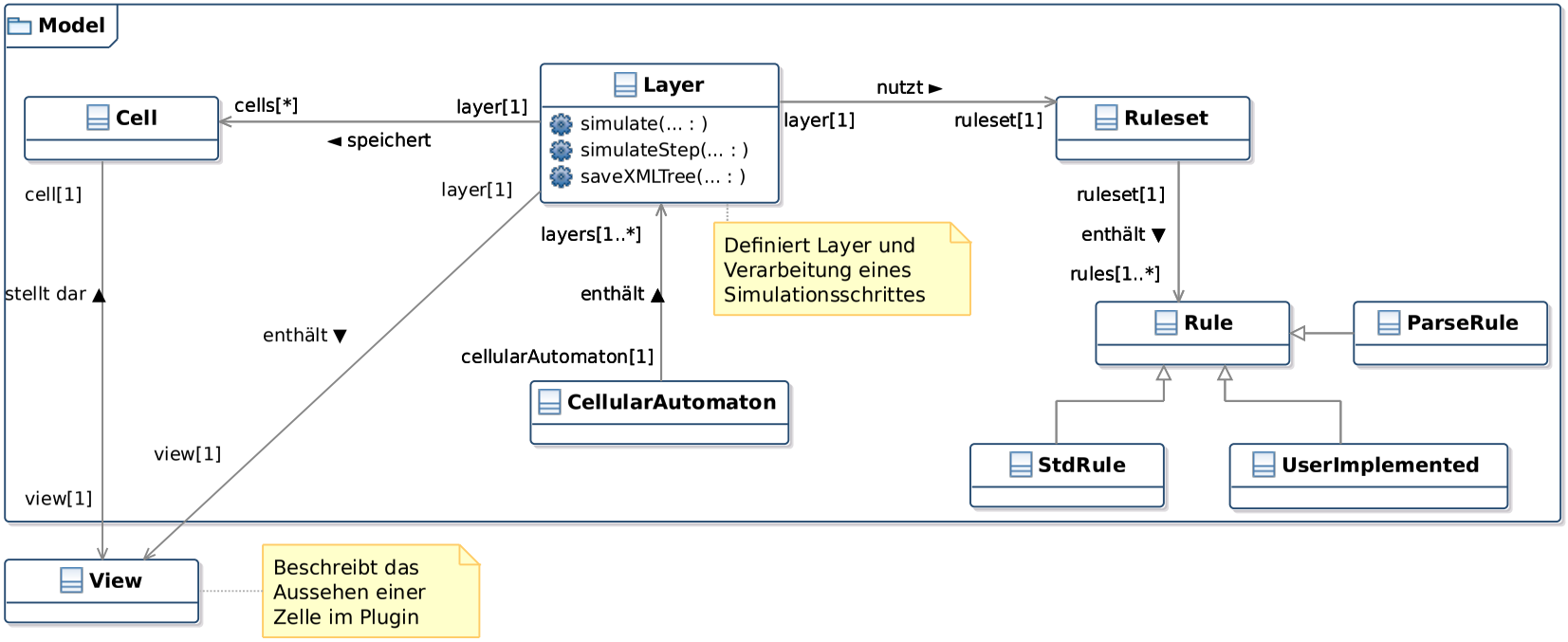


Abbildung 3.3.: Klassendiagramm des Models des MosquitoCA-Plugins (die Attribute und Methoden der Klassen werden nicht in Gänze aufgeführt) [48], [52], [50], [55], [54], [56], [53], [57] und [58].

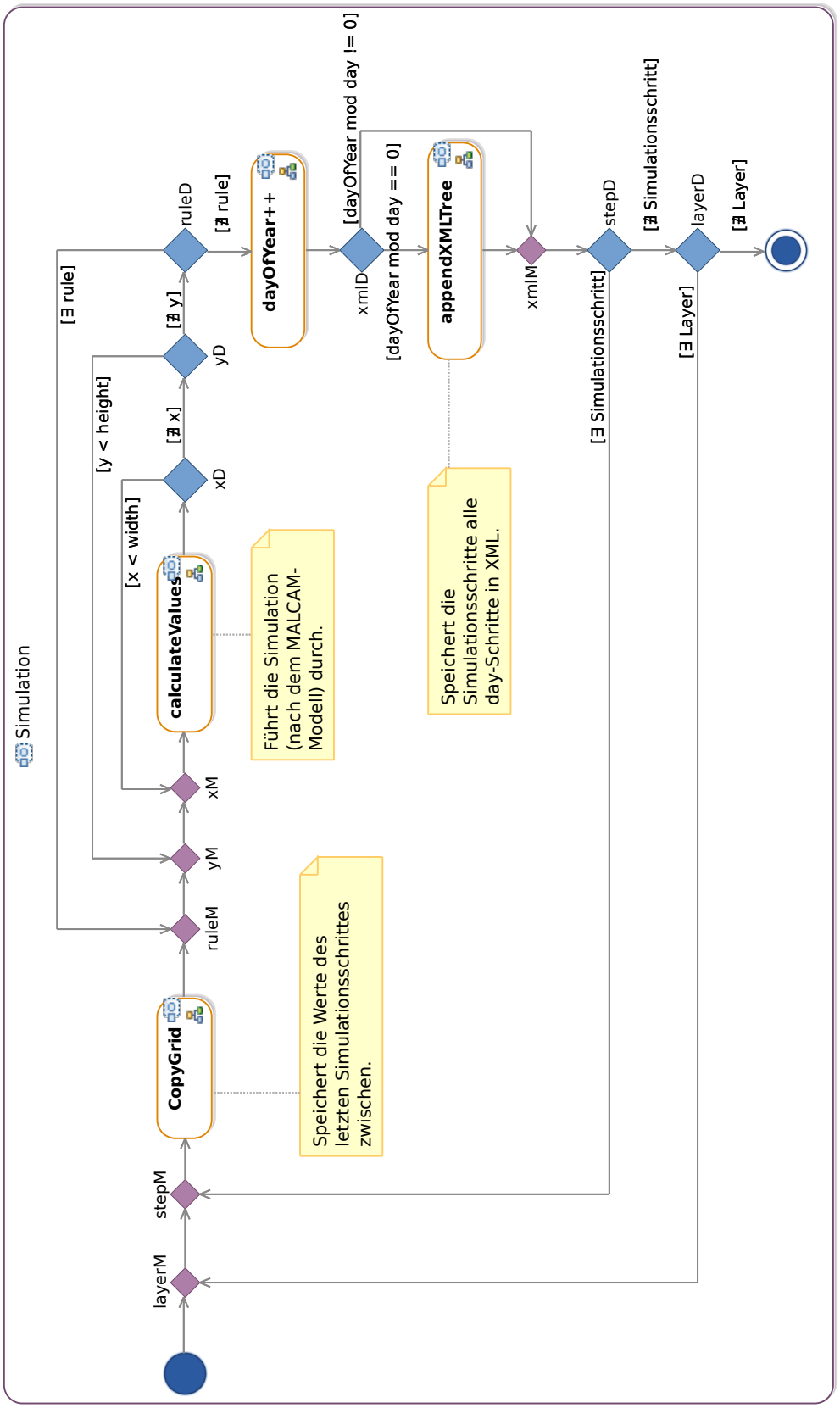


Abbildung 3.4.: Aktivitätsdiagramm der Ausführung einer Simulation des MosquitoCA-Plugins [50] und [52].

3.4.4. Diskussion

Das hier vorgestellte QGIS-Plugin MosquitoCA ist prinzipiell lauffähig und lässt sich insbesondere in Richtung eines Mehrlevel Zellularen Automaten erweitern. Dennoch sind einige Probleme zu erkennen, welche in dieser Arbeit teilweise behoben werden können. Eine Schwachstelle ist die Datenspeicherung in XML. XML ist zwar eine weit verbreitete Auszeichnungssprache, um Daten textbasiert zu speichern, wird allerdings im Bereich Biologie eher selten eingesetzt. Gebräuchlichere Formate sind dort CSV, TXT oder DAT. Daher könnte es zu Problemen bei der Weiterverarbeitung der Daten in anderen Programmen seitens der Zielgruppe dieses Werkzeugs kommen. Des Weiteren gibt es bei XML keine Möglichkeit Strukturen für einen effizienten Zugriff zu schaffen (dieses Problem gilt auch für alle anderen oben aufgeführten Textformate). Im worst case muss das gesamte XML-File eingelesen und durchsucht werden, um den Zustand des Zellularen Automaten zu einem bestimmten Zeitpunkt zu bestimmen. Bei Verwendung einer Datenbank, welche Indexe für die räumliche- und die temporale Dimension der Daten zur Verfügung stellt, ließe sich der Datenzugriff signifikant effizienter gestalten. Ferner wird der Startzustand der Simulation nicht gespeichert.

Des Weiteren ist keine strikte Trennung zwischen Dateneingabe, Datenausgabe und Simulationskern vorgenommen worden. Teile der Speicherung in XML sind direkt im Simulationskern implementiert. Dadurch wird eine Erweiterung um neue Datenausgabeformate erschwert. Für das mögliche Einlesen von Simulationsdaten wie bspw. Temperaturdaten aus externen Quellen steht in diesem Werkzeug keine Schnittstelle bereit.

Ein weiteres Problem des Plugins ist, dass die „View“ nicht strikt vom Modell getrennt ist (s. Assoziationen `Cell` bzw. `Layer` zu `View` Abbildung 3.3). Dadurch kann die farbliche Darstellung der Zellen in der GUI während der Simulation geändert werden. Dieses Vorgehen ist jedoch problematisch, da für eine Portierung des eigentlichen Simulationskerns dieser verändert werden muss. Des Weiteren werden in der derzeitigen Implementierung keine verschiedenen Layer benötigt. Wodurch sich die Komplexität des Werkzeugs unverhältnismäßig erhöht. Analoges gilt für die Regelsätze, welche in `Ruleset` gespeichert werden. Verschiedene Modelle gleichzeitig auf einen Zellularen Automaten anzuwenden erscheint fragwürdig, da jedes Modell in sich geschlossen ist.

Abbildung 3.4 zeigt, dass für jeden Zeitschritt der Simulation eines Zellularen Automaten einmal das komplette Gitter kopiert wird, um während der Berechnun-

gen auf den Zustand des vorherigen Zeitschrittes zurückgreifen zu können. Es wird im Laufe dieser Arbeit untersucht werden, inwiefern eine Abänderung der Datenstrukturen dazu führt, dass diese Kopieroperationen nicht mehr erforderlich sind. Dadurch ließe sich Rechenzeit sparen. Des Weiteren berechnet der Algorithmus in jedem Schritt die Zustände aller Zellen neu. Aufgrund der Nachbarschaftseigenschaften kann es jedoch sein, dass es Regionen des Zellularen Automaten gibt, deren Zustand sich während der Simulation nicht ändert. Das tritt insbesondere auf, wenn der Zellulare Automat sehr viele Zellen enthält, jedoch nur einige wenige Zellen Mücken oder Larven beheimaten. Dann gilt für die Anzahl an Mücken(-larven) verhältnismäßig vieler Zellen vor - und nach der Simulation, dass beide Werte Null sind. Entsprechend wird verhältnismäßig viel Rechenzeit auf Zellen verwandt, deren Zustand ohne Berechnung determiniert ist.

Die Schwächen der GUI werden an dieser Stelle nicht einzeln aufgeführt, da diese Arbeit sich hauptsächlich mit dem Simulationskern sowie der Datenhaltung unabhängig von der Nutzerschnittstelle befasst.

Kapitel
4

Analyse

In diesem Kapitel wird das zu entwickelnde System zunächst mithilfe eines Anwendungsfalldiagramms definiert. Anschließend werden die Anforderung an das zu entwickelnde Werkzeug definiert. Dabei werden die Anforderungen in vier Kategorien (aus Anwendungssicht, Datenhaltung, Simulationskern und Generell) eingeteilt. Die Anforderungen werden innerhalb einer Kategorie nach der Priorität sortiert. Abschließend werden die herausgearbeiteten Anwendungsfälle auf Grundlage der Anforderungen beschrieben.

4.1. Anwendungsfalldiagramm

In Abbildung 4.1 wird ein Anwendungsfalldiagramm des hier zu besprechenden Werkzeugs gezeigt. Es lassen sich zwei Kernanwendungsfälle, das *Initialisieren* der Simulation sowie die eigentliche *Simulation*, herauskristallisieren.

Während der Initialisierung muss die Konfiguration des Zellularen Automaten (Größe, Anzahl Zellen, Zellgröße und Startzustand) festgelegt werden. Des Weiteren muss festgelegt werden, wie die Simulationsergebnisse gespeichert bzw. ausgegeben werden sollen. Möglichkeiten sind bspw. Datenbanken (z. B. PostgreSQL / PostGIS) oder Dateien (z. B. CSV). Dieser Umstand wird durch den Akteur „AbstractData“ in Abbildung 4.1 symbolisiert, von dem das Datenbanksystem erbt und somit beispielhaft für eine mögliche Form der Datenhaltung steht. Im Folgenden impliziert, sofern nicht anders angegeben, „Simulationsergebnisse“ auch den Startzustand.

Die für die Simulation erforderlichen Daten wie z. B. die Temperatur, können aus verschiedenen Quellen stammen. Entsprechend müssen während der Initialisierung ggf. Dateien oder Datenbankenanbindungen geöffnet werden. Es soll an dieser Stelle nicht angenommen werden, dass Daten im Verlaufe der Simulation konstant sind. Daher müssen ggf. während der Simulation Daten eingelesen werden.

Die eigentliche Simulation beinhaltet die Simulationsschritte, sowie ggf. die Datenein- und ausgabe.

Die Initialisierung der Simulation und die Simulation selbst werden vom Nutzer initiiert. Es wird an dieser Stelle davon ausgegangen, dass das hier entwickelte Werkzeug an anderer Stelle in eine GUI integriert wird. Prinzipiell wäre es auch denkbar, dass der „Benutzer“ eine andere Simulation oder Datenhaltung ist, welche diese Simulation von außen triggert.

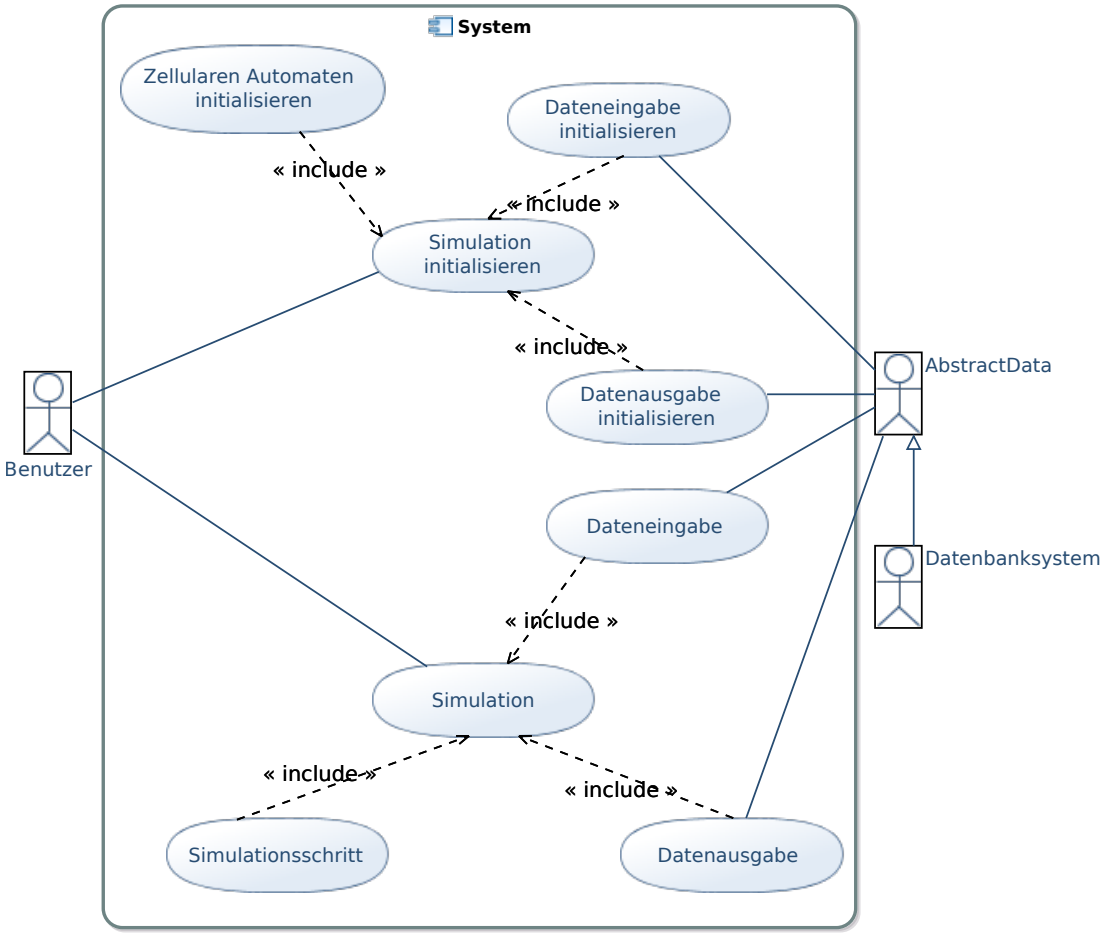


Abbildung 4.1.: Anwendungsfalldiagramm des zu entwickelnden Werkzeugs.

4.2. Anforderungen

4.2.1. Anforderungen aus Anwendungssicht

A1 Das System muss das Ausbreitungsverhalten von Mücken simulieren können.

A2 Es müssen für die Simulation relevante Daten eingegeben werden können:

- Temperatur
- Sterberate der erwachsenen Mücken
- Reproduktionsrate der Mücken
- Initialzustand des Zellularen Automaten

A3 Das System muss Simulationsergebnisse ausgeben können.

A4 Das System muss die Simulationsergebnisse abspeichern können.

4.2.2. Anforderungen an die Datenhaltung

D1 Die Simulationsergebnisse sollen in einer Datenbank gespeichert werden, welche die OGC-Standards (Open Geospatial Consortium) [78] umsetzt.

D2 Es soll ohne großen Implementierungsaufwand[1] möglich sein, die Daten in anderen Formaten (z. B. XML, CSV, ...) abzuspeichern.

D3 Es soll eine Initialisierung geben. Erforderliche Daten gemäß [48, 52, 56] sind hierfür:

- Temperatur
- Sterberate der erwachsenen Mücken
- Reproduktionsrate der Mücken
- Initialzustand des Zellularen Automaten

D4 Die Dateneingabe muss analog wie die Datenausgabe ohne großen Implementierungsaufwand um neue Quellen erweiterbar sein.

D5 Die Datenhaltung soll unabhängig vom Simulationskern sein.

D6 Die Datenhaltung soll unabhängig von dem UI (User Interface) sein.

D7 Die Simulationsergebnisse sollen möglichst komfortabel aus der Datenbank in QGIS geladen und dort angezeigt werden können (es gibt *keine* Verbindung des Werkzeugs zu QGIS).

4.2.3. Anforderungen an den Simulationskern

S1 Der Simulationskern soll das MALCAM-Modell (s. Kapitel 3.3) implementieren.

S2 Der Simulationskern soll das Zellulare Automaten-Modell (s. Kapitel 3.1) implementieren.

[1] „Ohne großen Implementierungsaufwand“ bedeutet in dieser Arbeit, dass lediglich eine neue Klasse implementiert werden muss, welche bspw. von einem Interface oder einer (abstrakten) Klasse erbt. Keinesfalls darf hierbei der Simulationskern oder die übrige Datenhaltung verändert werden.

S3 Zellulare Automaten, die ein oder mehrere Mückencluster[2] enthalten, müssen effizient[3] verarbeitet werden.

S4 Dicht besetzte Zellulare Automaten müssen effizient verarbeitet werden.

S5 Die Simulation soll um andere Regelsätze als das MALCAM-Modell, welche einen flachen Zellularen Automaten als Grundlage haben und die Anzahl an Mückenlarven sowie die Anzahl an erwachsenen Mücken berechnen, erweiterbar sein.

S6 Der Simulationskern soll unabhängig von der Datenhaltung sein.

S7 Der Simulationskern soll unabhängig von dem UI sein.

4.2.4. Generelle Anforderungen

G1 Das Werkzeug lässt sich leicht erweitern.

G2 Der Quellcode weist eine hohe Wiederverwendbarkeit auf.

G3 Der Quellcode wird dokumentiert.

G4 Das Werkzeug wird in der Programmiersprache Python 3.5.1 [92] implementiert.

G5 Der Simulationskern soll unabhängig von dem verwendeten Betriebssystem oder anderer Drittsoftware (z. B. QGIS) lauffähig sein.

G6 Das Werkzeug soll unter einer freien Lizenz stehen.

G7 Das Werkzeug soll keine proprietäre Drittsoftware nutzen.

4.3. Anwendungsfälle

Es wird im Folgenden davon ausgegangen, dass es verschiedene Dateneingabemöglichkeiten, Simulationsmodelle und Datenausgabemodelle gibt, da das Werkzeug leicht erweiterbar sein soll (vgl. Anforderungen G1, D2 und D4). Die in Abbildung 4.1 definierten Anwendungsfälle werden im Folgenden konkretisiert:

[2] Mückencluster sind in dieser Bachelorarbeit einzelne Regionen, in denen sich Mücken(larven) befinden. Zellulare Automaten, die einige Mückencluster enthalten, sollen nur in einigen wenigen Regionen Mücken(larven) enthalten und ansonsten keine.

[3] Effizient bedeutet in dieser Bachelorarbeit, dass möglichst wenig Rechenzeit benötigt wird.

Name:	**Initialisierung der Simulation**
	Der Benutzer initialisiert eine neue Simulation mit allen erforderlichen Parametern.
Akteur:	initiiert von Benutzer
Ablauf:	• Der Benutzer wählt ein Simulationsmodell aus (u. U. ist hier keine Wahl möglich). • Der Benutzer legt die Zellgröße, die Anzahl der Zellen im zweidimensionalem Raum und die Position des Zellularen Automaten bzgl. des nullten Längen- und Breitengrades fest. • Der Benutzer legt den Startzustand des Zellularen Automaten fest. • Der Benutzer legt die Datenausgabeformate fest und übergibt ggf. die notwendigen Parameter wie bspw. Dateinamen (u. U. ist hier keine Wahl möglich).
Anfangsbedingung:	hier nicht spezifizierbar
Abschlussbedingung:	Die Simulation ist initialisiert und kann prinzipiell gestartet werden.
Qualitätsanforderungen:	Auf fehlerhafte Eingaben wird eine Exception ausgegeben. Die Verarbeitung der Exception (z. B. anzeigen einer Fehlermeldung in einer GUI) ist nicht Teil dieser Arbeit.

Name:	**Durchführung der Simulation**
	Der Benutzer startet eine neue Simulation.
Akteur:	initiiert von Benutzer
Ablauf:	• Der Benutzer startet die Simulation. Simulationsparameter sind dabei die Schrittweite in Tagen, sowie die Anzahl der Simulationsschritte. Des Weiteren muss angegeben werden, mit welcher Granularität Zwischenergebnisse gespeichert bzw. ausgegeben werden sollen.
Anfangsbedingung:	Das System ist gemäß dem Anwendungsfall „Initialisierung der Simulation“ initialisiert worden.
Abschlussbedingung:	Die Simulation ist beendet und die Daten können mittels Drittsoftware (z. B. QGIS) ausgewertet werden.

Kapitel 5

Entwurf

In diesem Kapitel werden unterschiedliche Entwurfsentscheidungen vorgestellt und diskutiert. Im ersten Abschnitt werden grundlegende Fragen wie bspw. die Nutzung einer bestimmten Programmiersprache besprochen. Der zweite Abschnitt bildet den Kern dieses Entwurfskapitels und widmet sich der Frage, wie der Simulationsalgorithmus des bisherigen QGIS-Plugins verbessert werden kann. Anschließend wird die zu entwickelnde Datenhaltung besprochen, wobei der Fokus auf der Nutzung einer räumlichen Datenbank liegt. Abschließend wird die dem Entwurf zugrunde liegende Architektur vorgestellt.

5.1. Grundlegende Entscheidungen

Zunächst ist zu entscheiden, ob das bisherige MosquitoCA-Plugin verändert oder aber der Simulationskern neu entwickelt werden soll. Für eine Weiterentwicklung spricht, dass das bisherige Werkzeug eine GUI besitzt, welche übernommen werden könnte. Dagegen spricht, dass der Simulationskern des bisherigen Plugins nicht strikt von der View getrennt ist (s. Kapitel 3.4.1). Dadurch lässt sich der Simulationskern nicht ohne weitere Arbeiten unabhängig von QGIS nutzen. Dieser Umstand widerspricht Anforderung G5. Ähnliches gilt für die Trennung des Simulationskerns von der Datenhaltung (vgl. Anforderungen D5 und S6). Des Weiteren ist in Kapitel 3.4.2 festgestellt worden, dass die Simulation insbesondere aufgrund der Kopieroperation des Zellularen Automaten ineffizient ist. Um diese Kopieroperation zu vermeiden, muss die Datenstruktur des Zellularen Automaten grundlegend neu entwickelt werden. Daher wird ein Großteil des bisherigen Quellcodes unbrauchbar. Deshalb ist es sinnvoll den Simulationskern neu zu implementieren. Aus diesem Grunde erscheint es wenig sinnvoll Code des alten Projekts zu übernehmen. Prinzipiell wird das neu zu entwickelnde Werkzeug das MALCAM-Modell umsetzen, jedoch kann es sein, dass die Implementierung von der Version von Daniel Klich abweichen wird, um die technischen Möglichkeiten der verwendeten Programmiersprache optimal auszunutzen.

Ferner ist zu entscheiden, ob das Werkzeug als QGIS-Plugin entwickelt werden soll. QGIS ist ein freies Geoinformationssystem (GIS) unter der GNU GPL-Lizenz und ist für alle gängigen Plattformen verfügbar [62, S. 953]. Eine Implementierung als Plugin böte die Möglichkeit für die Datenhaltung und -visualisierung QGIS zu nutzen. Der Nachteil ist jedoch, dass eine Nutzung ohne QGIS nicht möglich ist. Nach Anforderung G5 wird gefordert, dass der Simulationskern unabhängig vom verwendeten Betriebssystem sowie anderer Drittsoftware lauffähig sein soll. Diese Forderungen sind plausibel, da es aus Sicht der zu implementierenden Funktionalitäten keine Gründe gibt, das Werkzeug auf eine Plattform (und damit auch QGIS) einzuschränken. Daher ist hier eine Implementierung als eigenständiges Werkzeug sinnvoll.

Gemäß Anforderung G4 soll das Werkzeug in Python 3.5.1 entwickelt werden. Es gibt einige Gründe, welche für bzw. gegen diese Anforderung sprechen. Wie schon oben erwähnt, basiert diese Arbeit auf einem QGIS-Plugin. Da es prinzipiell denkbar wäre, dass dieses Werkzeug zu einem späteren Zeitpunkt in ein QGIS-Plugin

reintegriert werden könnte, wäre es sinnvoll eine Programmiersprache zu nutzen, welche diese Option vereinfacht. QGIS-Plugins können prinzipiell in C++ oder Python entwickelt werden [94]. Für die Nutzung von C++ spricht, dass i. A. effizientere Programme entwickelt werden können als in Python [85]. Nachteilig ist jedoch, dass C++ schwieriger zu erlernen ist als Python [47, S. 1]. Ferner unterstützt Python den Programmierer u. A. beim Speichermanagement, welches in C++ manuell erfolgt und bei der Indexierung von Listen [66, S. 155]. Des Weiteren soll dieses Werkzeug von Biologen genutzt und ggf. möglichst einfach erweitert werden können. Daher ist es sinnvoll die Implementierung in Python vorzunehmen. Ein weiterer Aspekt ist die in Anforderung G5 geforderte Plattformunabhängigkeit. Python bietet als interpretierte Programmiersprache den Vorteil einer hohen Plattformunabhängigkeit [121, S. 27]. Es wird lediglich ein Interpreter für das verwendete Betriebssystem benötigt. Entsprechende Software ist für alle gängigen Plattformen frei verfügbar [73, S. 13 f.]. Bei in C++ implementierter Software ist die Plattformunabhängigkeit nicht zwingend gegeben und müsste entsprechend für alle gängigen Plattformen verifiziert werden. Daher spricht auch die Plattformunabhängigkeit für die Verwendung von Python. Prinzipiell wäre es denkbar Matlab (widerspricht Anforderung G7) [11, S. 8] und [96, S. 1 f.], GNU Octave [43] und [96, S. 1 f.] (derzeit kein adäquater Matlab-Ersatz), Julia [7] und [11, S. 101] (noch nicht sehr verbreitet), GNU R [97] und [70, S. 204 ff.] (auf Statistik spezialisiert) oder andere zu nutzen. Aufgrund der fehlenden Kompatibilität zu QGIS werden diese Programmiersprachen hier nicht weiter diskutiert.

Aufgrund der einfachen Erlernbarkeit und hohen Kompabilität zu QGIS wird Python als Programmiersprache in dieser Arbeit genutzt.

Im Folgenden soll entschieden werden, ob für die Speicherung der Daten in einer Datenbank Raster- oder Vektordaten verwendet werden sollen (vgl. Abschnitt 2.2). Der Vorteil der Nutzung von Rasterdaten ist darin zu sehen, dass Zellulare Automaten das Universum gemäß einem Raster partitionieren (vgl. Abschnitt 3.1). Daher bietet es sich an auch in der Datenbank Rasterdaten zu verwenden. Andererseits bieten Vektordaten eine von der Auflösung unabhängige Datenspeicherung, so dass bei dünn besetzten Zellularen Automaten möglicherweise weniger Speicherplatz belegt wird, als das für Rasterdaten der Fall wäre.

Für die weitere Arbeit wird von einem rechteckigen Zellularem Automaten ausgegan-

gen, dessen Zellen die gleiche Größe (Breite sowie Höhe) aufweisen und gitterartig in einem zweidimensionalem Universum angeordnet sind.

5.2. Simulation

Die Simulation erfolgt unter Ausnutzung des Zellularen Automatenmodells, da dieses leicht zu implementieren ist und das Modell schon erfolgreich für die Simulation des Ausbreitungsverhaltens eingesetzt worden ist [102] und [59]. Für die Übergangsfunktion wird ähnlich wie bei Klich [49] das MACAM-Modell von Linard et al. verwendet [68].

Die Ränder des Zellularen Automaten müssen gesondert betrachtet werden. Häufig werden bei ähnlichen Problemen periodische Randbedingungen [46, S. 11] oder Nullränder [41, S. 41] angenommen. Beide Lösungen haben Vor- und Nachteile. Während periodische Randbedingungen einfacher als andere Lösungen zu implementieren sind und die Massenerhaltung gilt, können jedoch unrealistische Szenarien entstehen. Angenommen ein Mückencluster stößt an die „nördliche Grenze“ des Zellularen Automaten. Dann sorgen periodische Randbedingungen dafür, dass Mücken auf Zellen an der „südlichen Grenze“ des Zellularen Automaten gelangen. Dadurch würde es Mücken in Regionen des Zellularen Automaten geben, wo diese ohne Randbedingungen nicht auftreten. Entsprechend könnte es sein, dass falsche Schlüsse aus der Simulation gezogen werden. Bei Nullrändern hingegen treten diese Probleme nicht auf. Dafür kann die Massenerhaltung nicht sichergestellt werden, da Mücken den Zellularen Automaten verlassen können [41, S. 41]. Eine dritte Möglichkeit besteht darin die Ränder zu vernachlässigen und die Mücken entsprechend auf weniger Zellen zu verteilen. Auch diese Lösung spiegelt die Realität nur unzureichend wieder, erscheint jedoch besser geeignet als die anderen beiden Lösungen, da die Masse erhalten bleibt und Mücken nicht an unrealistischen Orten erscheinen können. Daher wird in dieser Arbeit die letzte Lösung implementiert.

Des Weiteren wird bei Linard et al. [68, S. 169] nicht genauer spezifiziert, in welche Richtung sich die Mücken beim Verlassen einer Zelle bewegen sollen. Bei Klich [56] ist die Aufteilung der Mücken auf die Nachbarzellen anhand von ökologischen Daten vorgenommen worden. Die gewählte Aufteilung ist aus Sicht des Autors nicht nachvollziehbar und wird aus diesem Grunde nicht implementiert. Daher werden in dieser Arbeit die Mücken gleichmäßig auf die Nachbarzellen verteilt, da der Implementierungsaufwand relativ gering ist.

5.2.1. Übersicht der verschiedenen Verfahren

Um den in Kapitel 3.4.2 besprochenen Simulationsalgorithmus von Klich [52] zu verbessern, werden hier einige Vorschläge diskutiert und abgewogen, welcher unter den in Kapitel 4.2 vorgestellten Anforderungen der am besten geeignete ist. Zur Vereinfachung werden die Verfahren nummeriert:

Alg1 „Einfaches Verfahren“ (verbesserte Version der Simulation in [52], die ohne Kopieroperation auskommt)

Alg2 „Indexverfahren“

Alg3 „Stapelverfahren“

Alg4 „Matrizenverfahren“

Alg5 „Kombiverfahren“

„Einfaches Verfahren“ (Alg1) Das „Einfache Verfahren“ ist sehr ähnlich zu dem Algorithmus des MosquitoCA-Plugins [52]. Allerdings wird der Zellulare Automat nicht explizit kopiert. Dafür werden zwei Matrizen für die Repräsentation des Zellularen Automaten benötigt (eine Diskussion verschiedener Datenstrukturen in Python erfolgt in Abschnitt 5.2.4). Eine Visualisierung des im Folgenden beschriebenen Rollenwechselverfahrens findet sich in Tabelle 5.1. Enthalte Matrix Υ zum Zeitpunkt t den Automatenzustand zum Zeitpunkt $t-1$. Der zu berechnende Zustand des Automaten zum Zeitpunkt t wird in Matrix Ξ gespeichert. Im nächsten Zeitschritt wechseln die Rollen der Matrizen Υ und Ξ. D. h. Matrix Ξ enthält den „alten“ Zustand des Automaten (Zeitpunkt t) und Matrix Υ wird mit den Daten zum Zeitpunkt $t+1$ beschrieben.

Tabelle 5.1.: Rollenwechsel der beiden Matrizen während der Simulation anhand eines zeitlichen Ausschnittes.

Anmerkung: Die Zeitangaben für die beiden Matrizen geben den aktuell repräsentierten Zustand an. Ist die entsprechende Tabellenzelle zu einer Matrix blau unterlegt, so wird diese Matrix im entsprechenden Zeitschritt nur gelesen, da sie den Zustand des Zellularen Automaten aus dem vorherigen Zeitschritt enthält. Die Matrix, deren zugehörige Tabellenzelle gelb eingefärbt ist, wird in dem entsprechenden Zeitschritt mit dem zu berechnenden Zustand beschrieben.

Simulationsschritt	t	$t+1$	$t+2$	$t+3$	$t+4$
Matrix Υ	$t-1$	$t+1$	$t+1$	$t+3$	$t+3$
Matrix Ξ	t	t	$t+2$	$t+2$	$t+4$

Während der Simulation werden alle Zellen des Zellularen Automaten neu berechnet. Vorteile dieses Verfahrens sind die Einfachheit der Implementierung sowie die Tatsache, dass bei dicht besetzten Matrizen gute Resultate erzielt werden können. Der Grund liegt darin, dass weder Rechenzeit noch Speicherplatz für die Verwaltung von aktiven Regionen gebraucht werden. Eine Verwaltung von aktiven Regionen bei dicht besetzten Zellularen Automaten senkt die Anzahl der neu zu berechnenden Zellen nur in geringem Maße, da aufgrund der Automatenkonfigurationen (fast) alle Zellen neu berechnet werden müssen. Der Nachteil dieses Verfahrens ist, dass bei dünn besetzten Zellularen Automaten ein schlechtes Laufzeitverhalten zu erwarten ist, da ein Großteil der Zellen unbesetzt ist und somit keine Neuberechnung erforderlich ist.

„Indexverfahren“ (Alg2) Die erweiterten-aktiven Bereiche (im Folgenden e-a-Bereiche) des Zellularen Automaten werden unter Ausnutzung der MBB-Eigenschaften [101, S. 202] durch sich nicht überlappende Unter-Zellulare Automaten (im Folgenden Teilautomaten) repräsentiert. Die aktiven Bereiche müssen um diejenigen Nachbarschaftszellen erweitert werden, deren Zustand sich im nächsten Simulationsschritt ändern. Dadurch arbeiten die Teilautomaten auf disjunkten Regionen und es gibt keine Zelle, die während eines Simulationsschrittes gleichzeitig von zwei Teilautomaten geändert wird. Dadurch kann die Berechnung der verschiedenen Teilautomaten parallel ablaufen.

Die Teilautomaten werden in einem R^+-Baum [31, S. 102 - 109] verwaltet, wobei sich während der Simulation auf Grund von Vergrößerungen der e-a-Bereiche sich überschneidende Teilautomaten zu einem Automat zusammengefügt werden.

In jedem Teilautomaten werden Matrizen analog wie im Abschnitt über Alg1 beschrieben verwendet. Entsprechend werden alle Zellen des Teilautomaten in jedem Simulationsschritt neu berechnet. Dieses Verfahren hat den Vorteil, dass bei Zellularen Automaten die gut mehrfach geclustert sind eine parallele Simulationsführung möglich ist. Problematisch ist jedoch, dass sich auch bei dünn besetzten Zellularen Automaten ein schlechtes Laufzeitverhalten einstellen kann, wenn die Konfiguration ungünstig ist. Ein weiterer Nachteil ist, dass dieses Verfahren einen sehr hohen Implementierungsaufwand impliziert.

Für Python sind einige Erweiterungen verfügbar, welche räumliche Indexe definieren. Das Paket Rtree [15] definiert einen R-Baum, wie in Kapitel 2.3.3 vorgestellt. Rtree hat derzeit jedoch den Nachteil, dass die letzte Version aus dem Jahre 2011 stammt. Daher ist davon auszugehen, dass das Paket nicht weiter entwickelt wird und mit Problemen in Python 3.5.1 zu rechnen ist. Eine Alternative ist Pyqtree [91], welches eine Implementierung des Quadtrees aus Kapitel 2.3.1 bereitstellt. Allerdings ist Pyqtree derzeit nur als beta-Version verfügbar, so dass ein Einsatz in diesem Werkzeug aktuell nicht in Frage kommt. Die letzte hier aufgeführte Option ist das Paket python-geohash [44], welches als Hashing-Verfahren für räumliche Daten genutzt werden könnte. Python-geohash ist ähnlich wie der Rtree in den letzten Jahren nicht mehr weiterentwickelt worden. Daher sind alle Erweiterungen derzeit nicht sinnvoll für das hier zu entwickelnde Werkzeug nutzbar.

„Stapelverfahren“ (Alg3) Der Zellulare Automat wird mit Hilfe von Matrizen gespeichert. Analog wie oben für Verfahren Alg1 beschrieben wird das Rollenwechselverfahren genutzt, um zum Zeitpunkt t auf den Zustand $t-1$ zugreifen zu können. Die Verwaltung der aktiven Zellen erfolgt mit Hilfe zweier Stapelspeicher [17, S. 95] „ToDo“ und „Next“, welche analog wie die Matrizen Ξ und Ψ das Rollenwechselverfahren ausnutzen. Wie in Abbildung 5.1 visualisiert enthält der Stapelspeicher „ToDo“ die Indexe der blau markierten Zelle(n), welche im aktuellen Simulationsschritt neu berechnet werden müssen. Der Stapelspeicher „Next“ wird während der Simulation aufgebaut und enthält die Indexe der blau bzw. gelb markierten Zellen, welche im nächsten Simulationsschritt neu berechnet werden müssen. Vorteil dieses Verfahrens ist, dass die Zugriffsoperationen auf dem Stapelspeicher sehr effizient sind und die benötigten Datenstrukturen unter der Voraussetzung, dass die verwendete Programmierumgebung einen Datentyp für Matrizen oder Arrays bereitstellt, relativ leicht zu implementieren sind. Problematisch ist, dass der Stapelspeicher insbesondere bei großen- und dicht besetzten Zellularen Automaten einen großen

Speicherbedarf aufweist.

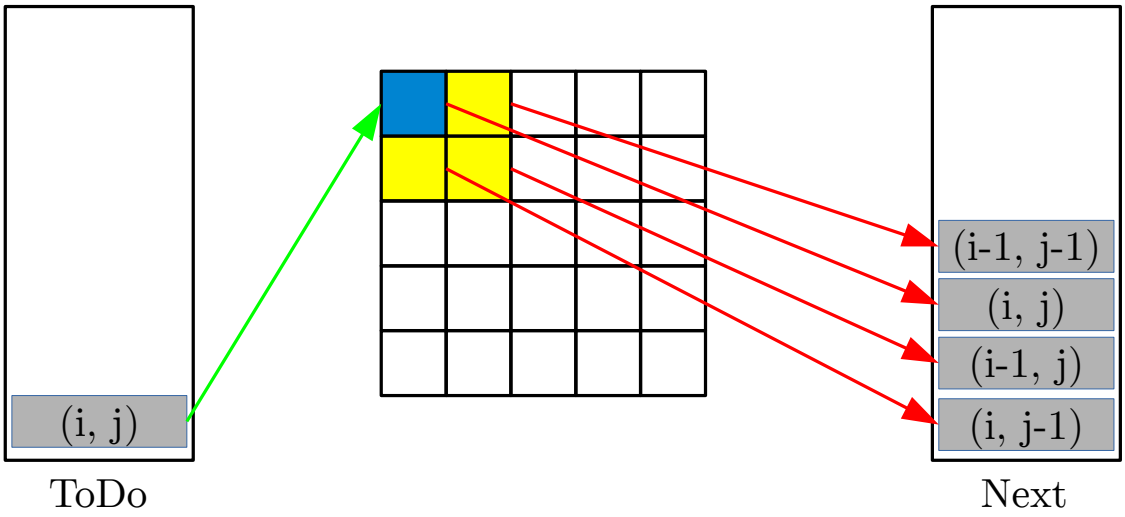


Abbildung 5.1.: Visualisierung des „Stapelverfahrens“ (Alg3).

„Matrizenverfahren“ (Alg4) Der Zellulare Automat wird mit dünn besetzten Matrizen (Sparse Matrizen) unter Ausnutzung des Rollenwechselverfahrens aus Alg1 abgespeichert. Bei dünn besetzten Matrizen werden nur Elemente ungleich Null gespeichert [111, S. 327]. Daher wird eine Liste der Indexe der neu zu berechnenden Zellen aus der Repräsentation des Zellularen Automaten erstellt. Der Vorteil besteht darin, dass keine zusätzliche Datenstruktur benötigt wird, um die Indexe der neu zu berechnenden Zellen zu verwalten. Die Implementierung ist sehr einfach, wenn die verwendete Programmiersprache einen Datentyp Sparse Matrix bereitstellt, wobei Zugriff auf die Indexliste der Elemente ungleich Null bestehen muss (in Abschnitt 5.2.4 wird eine entsprechende Datenstruktur für Python vorgestellt). Der Nachteil dieses Verfahrens ist, dass dicht besetzte Zellulare Automaten bzgl. Speicherplatznutzung nicht effizient mit Hilfe von Sparse Matrizen gespeichert werden können.

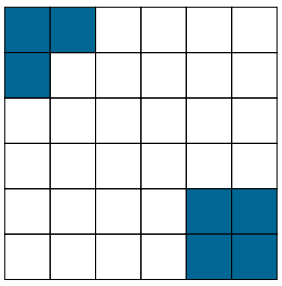
„Kombiverfahren“ (Alg5) In diesem Verfahren wird zwischen einem dicht- und einem dünn besetzten Zellularen Automaten unterschieden. Im Falle eines dicht besetzten Zellularen Automaten wird die Lösung aus Alg1 verwendet. Ist der Zellulare Automat jedoch dünn besetzt, so wird eine der Lösungen Alg2 bis Alg4 verwendet. Eine Heuristik entscheidet, welches Verfahren bei einer Simulation genutzt wird. Die Vorteile dieses Verfahrens sind, dass sowohl bei dicht- als auch bei dünn besetzten Zellularen Automaten gute Resultate erzielt werden können. Nachteilhaft ist jedoch,

dass zwei Verfahren implementiert werden müssen. Des Weiteren hängt die Güte des Kombiverfahrens maßgeblich davon ab, dass die Heuristik effizient und zuverlässig das am Besten geeignete Verfahren auswählt.

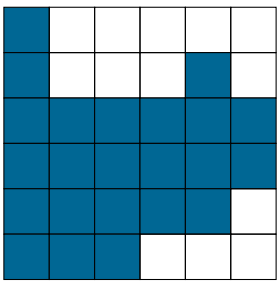
5.2.2. Vergleich der Verfahren bzgl. verschiedener Konfigurationen des Zellularen Automaten

Um die fünf im vorigen Abschnitt vorgestellten Verfahren vergleichen zu können, müssen zunächst einige „Extremfälle“ bzgl. der Konfiguration des Zellularen Automaten betrachtet werden. Aus grafischen Gründen werden im Folgenden in Abbildung 5.2 Zellulare Automaten mit einigen wenigen Zellen vorgestellt. Die Argumentation funktioniert jedoch auch mit beliebig großen Zellularen Automaten.

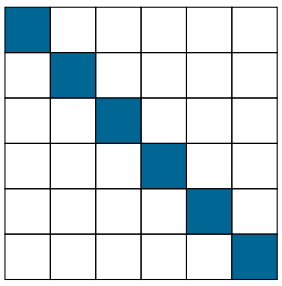
Der in Abbildung 5.2(a) gezeigte Zellulare Automat zeichnet sich dadurch aus, dass er dünn besetzt ist und es zwei Mückencluster gibt (die besetzten Zellen befinden sich in zwei verschiedenen Regionen), welche sich nicht überlappen. Daher sind während der Simulation viele Zellen ohne eine mathematische Berechnung determiniert. Die Konfiguration in Abbildung 5.2(b) hingegen ist dicht besetzt, es müssen während eines Simulationsschrittes nahezu alle Zellen neu berechnet werden. In Abbildung 5.2(c) ist zu erkennen, dass es zwar wenige Zellen gibt, welche Mücken(larven) enthalten, als solche die Mücken(larven) enthalten. Allerdings treten hier keine Mückencluster auf.



(a) Dünn besetzt mit Clustering.



(b) Dicht besetzt.



(c) Dünn besetzt ohne Clustering.

Abbildung 5.2.: Unterschiedliche Konfigurationen von mit Mücken(larven) (blau markierte Felder) besetzten Zellularen Automaten.

Zunächst werden die Verfahren Alg1 bis Alg4 gemäß den drei Extremfällen verglichen. Für den in Abbildung 5.2(a) angegebenen Fall eines dünn besetzten Zellularen Automaten mit disjunkten Mückenclustern ist festzustellen, dass das „Einfache Ver-

fahren“ (Alg1) erhebliche Defizite aufweist, da in jedem Simulationsschritt immer alle Zellen neu berechnet werden, obwohl das für viele Zellen nicht notwendig ist. Die Verfahren Alg2 bis Alg4 zielen darauf ab, die Anzahl der in einem Simulationsschritt neu zu berechnenden Zellen zu minimieren. Mit Ausnahme des „Indexverfahrens“ (Alg2) gelingt das unabhängig von der Konfiguration des Zellularen Automaten. Bei dem „Indexverfahren“ kann es im worst case bei einer Konfiguration wie in Abbildung 5.2(c) passieren, dass alle Zellen neu berechnet werden, da kein Mückenclustering auftritt. Aus diesem Grund ist das „Indexverfahren“ sowohl dem „Stapelverfahren“ als auch dem „Matrizenverfahren“ unterlegen. Im Falle eines dicht besetzten Zellularen Automaten ist zu erwarten, dass das „einfache Verfahren“ die besten Ergebnisse erzielt, da dieses keinen Overhead für die Verwaltung der Simulation erzielt. Die Verfahren Alg2 bis Alg4 verwenden zum Teil sowohl Speicherplatz als auch Rechenzeit, um möglichst wenige Zellen neu zu berechnen. In Abbildung 5.2(b) müssen jedoch (fast) alle Zellen neu berechnet werden. Daher lohnt es sich nicht in diesem Fall eine Simulationsverwaltung zu nutzen.

5.2.3. Vergleich der Verfahren bzgl. der Anforderungen

In Tabelle 5.2 erfolgt ein tabellarischer Vergleich der fünf Verfahren bzgl. der in Kapitel 4.2.3 aufgestellten Anforderungen. Die Verfahren erfüllen die gestellten Anforderungen jeweils ähnlich gut, wobei Alg5 marginal besser als die übrigen abschneidet.

Tabelle 5.2.: Übersicht des Vergleichs der Verfahren bzgl. der Anforderungen in Kapitel 4.2.3.

Anmerkung: „+“: wird erfüllt, „-“: wird nicht erfüllt, „0“: weder + noch -, „?“: hier nicht entscheidbar / relevant, „(+/-/0)?“: nicht sicher, ob Wahl immer korrekt.

	Alg1	**Alg2**	**Alg3**	**Alg4**	**Alg5**
D1	?	?	?	?	?
D2	?	?	?	?	?
D3	?	?	?	?	?
D4	?	?	?	?	?
D5	+	+	+	+	+
D6	?	?	?	?	?
D7	?	?	?	?	?
S1	+	+	+	+	+

S2	+	+	+	+	+
S3	-	+	+	+	+
S4	+	0	0	0	+
S5	+	+	+	+	+
S6	+	+	+	+	+
S7	+	+	+	+	+
G1	?	?	?	?	?
G2	+	+	+	+	+
G3	?	?	?	?	?
G4	?	?	?	?	?
G5	+	+	+	+	+
G6	+	+	+	+	+
G7	+	?	+	+	+
$\sum$	**+10**	**+10**	**+11**	**+11**	**+12**

5.2.4. Vergleich von Datenstrukturen in Python

In den oben aufgeführten Verfahren ist vorgesehen die Simulationsdaten in Matrizen abzuspeichern. Hier wird analysiert, welche Datenstrukturen in Python verfügbar sind und inwiefern diese sich für diese Arbeit eignen.

Python bietet die Möglichkeit mehrdimensionale Listen zu nutzen [121, S. 189]. Dabei handelt es sich um eine dynamische Datenstruktur, mit der beliebige Objekte abgespeichert werden können. Damit böte sich in dieser Arbeit die Möglichkeit einen Datentyp zu entwerfen, der die Attribute einer Zelle des Zellularen Automaten abspeichert. Der Zellulare Automat würde sich als Liste von Zellen ergeben. Ferner kann die Liste in Python auch als Stapelspeicher genutzt werden [47, S. 135] und würde sich somit für die Verwaltung des „Stapelverfahrens“ (Alg3) eignen.

Eine Alternative zu den Listen von Python besteht darin den Zellularen Automaten als NumPy-Array zu repräsentieren. Dabei handelt es sich um ein n-dimensionales Array, das numerische- bzw. boolesche Daten enthalten kann [73, S. 36 ff.]. Der Vorteil von NumPy-Arrays ist, dass es möglich ist mathematische Operationen auf einem ganzen Array durchzuführen, ohne alle Elemente explizit in einer Schleife zu durchlaufen. Aufgrund des internen Aufbaus von NumPy ist dieses Vorgehen effizienter, als das explizite Durchlaufen des Arrays [67, S. 139].

Die dritte hier aufgeführte Möglichkeit ist die Nutzung von Sparse Matrizen des

SciPy-Pakets [42, S. 236]. Der Unterschied zu den oben aufgeführten NumPy-Arrays ist, dass die Sparse Matrizen nur Werte ungleich Null explizit abspeichern. Diese Datenstruktur ist insbesondere bei dünn besetzten Zellularen Automaten interessant, da sich somit der Hauptspeicherbedarf des Programms senken lässt. Ein Nachteil ist jedoch, dass intern zusätzlich zu den eigentlichen Datenwerten ein Index gespeichert werden muss. Daher sind Sparse Matrizen für dicht besetzte Zellulare Automaten schlechter geeignet als NumPy-Arrays [42, S. 253].

Eine weitere Möglichkeit wäre die Nutzung von Dictionaries. Dabei handelt es sich um eine Zuordnung von Schlüsseln zu einem Werten [121, S. 221]. Prinzipiell ist es denkbar den Zellularen Automaten als Dictionary zu repräsentieren. Ferner könnten Dictionaries als Austauschdatenstruktur zwischen verschiedenen Teilen des Werkzeugs genutzt werden.

Um besser entscheiden zu können, welche der aufgeführten Datenstrukturen für das hier zu entwickelnde Werkzeug bzgl. Performanz und Speicherplatzbelegung bestmöglich geeignet ist, werden hier einige Experimente vorgestellt. Verglichen werden dabei die Python internen Datenstrukturen Lists und Dictionaries, sowie das NumPy-Array und die SciPy lil-Matrix als Beispiel einer Sparse Matrix. Zur Bestimmung der Zugriffsperformanz wird das Skript `profile.py` verwendet. Das Skript liest zunächst eine Zahl n (Datentyp `Integer`, vgl. Listing 5.1 Z. 9) ein, welche die Größe der jeweiligen Datenstrukturen repräsentiert. Im Folgenden wird von einer $n \times n$-Matrix ausgegangen. Das Skript speichert die bestimmten Zeiten in TXT-Dateien, welche mit einem externen Programm / Skript weiter verarbeitet werden können. Es werden die Ausführungszeiten für die Schreibzeit (Z. 13: öffnen der Datei & 18: Schreiben der aktuellen Messwerte), Lesezeit (Z. 14 & 19) sowie Schreibe-Lesezeit (Z. 15 & 20) in TXT-Dateien gespeichert. In einer Zeile dieser TXT-Dateien stehen jeweils Leerzeichen separiert der Füllgrad als Ganzzahlen, sowie jeweils die für die vier Datenstrukturen gemessenen Zeiten als Gleitkommazahlen. Dabei werden Messwerte für verschiedene Füllgrade aufgenommen. Der Startwert der Schleife Z. 16 ist ungleich Null, da ansonsten beim ersten Versuchsschritt keine sinnvolle Messung durchgeführt werden könnte. Um die Zugriffszeiten auf die einzelnen Datenstrukturen zu messen wird die Methode `clock` aus dem `time`-Paket [67, S. 438] verwendet. Das Vorgehen wird hier beispielhaft für die SciPy lil-Matrix

Quellcode
dir: /sonstige_codes/profile
file: profile.py

aufgezeigt. In Listing 5.2 wird in Z. 1 zunächst eine $n \times n$-lil-Matrix initialisiert. Anschließend wird der Startzeitpunkt des Experiments bestimmt (Z. 2). Danach wird die Matrix mit einer durch a festgelegten Anzahl an numerischen Elementen befüllt (Z. 3 – 5). Abschließend wird die für die zu untersuchende Zeit in Z. 6 errechnet. Analog wie in den Zeilen 3 – 6 wird für den Lesezugriff verfahren. Nach dem in Listing 5.2 vorgestelltem Lösungsansatz werden alle vier Datenstrukturen untersucht.

Listing 5.1: Initialisierung und Datenspeicherung für die Untersuchung des Laufzeitverhaltens verschiedener Datenstrukturen (die eigentliche Messung wird beispielhaft in Listing 5.2 aufgeführt).

```
1  # This script measures the access time for several data structures.
2  # Version 1.0 17.06.2016 Stephan Adolf
3
4  import numpy as np
5  from scipy.sparse import lil_matrix
6  from time import clock
7  from sys import argv
8
9  n = int(argv[1]) # read param n from shell
10 start = int(n/10)
11 step = int(n/10)
12
13 with open("compare_write_%i.txt"%n, "w") as text_file1:
14   with open("compare_read_%i.txt"%n, "w") as text_file2:
15     with open("compare_rw_%i.txt"%n, "w") as text_file3:
16       for a in range(start, n, step):
17         Zeitmessung s. Listing 5.2
18         print(str(a) + " " + str(time_w_A) + " " + str(time_w_B) + " "
               + str(time_w_C) + " " + str(time_w_D), file=text_file1)
19         print(str(a) + " " + str(time_r_A) + " " + str(time_r_B) + " "
               + str(time_r_C) + " " + str(time_r_D), file=text_file2)
20         print(str(a) + " " + str(time_w_A-time_r_A) + " " + str(
               time_w_B-time_r_B) + " " + str(time_w_C-time_r_C) + " " +
               str(time_w_D-time_r_D), file=text_file3)
```

Listing 5.2: Zeitmessung am Beispiel des Schreibens einer lil-Matrix bis zur Belegung a.

```
A = lil_matrix((n ,n))
time_w_A = clock() # measure writing
for i in range(a):
  for j in range(a):
    A[i, j] = 42
time_w_A = clock() - time_w_A
```

Die Messwerte werden mit matplotlib [39], [66, Kapitel 5.3.1] und [73, Kapitel 7] unter Python 2.7 mit Hilfe der Skripte `plot.py` und `myplot.py` dargestellt.

Quellcode
dir: /sonstige_codes/profile
file: plot.py

In Abbildung 5.3 wird die benötigte Zeit zum Schreiben der vier Datenstrukturen gegen den Füllgrad bei einer Matrixgröße $n = 100$ aufgetragen. Der Grafik ist zu entnehmen, dass die Zugriffszeiten für alle Datenstrukturen bei zunehmendem Füllgrad größer werden. Dieser Umstand ist zu erwarten gewesen, da bei einem erhöhten Füllgrad mehr Elemente beschrieben werden müssen. Des Weiteren ist zu erkennen, dass das dicht besetzte NumPy-Array sowie das Dictionary signifikant schlechter als die SciPy Sparse Matrix bzw. die Liste abschneiden. Insbesondere ist festzustellen, dass sich die Liste bzgl. der Schreibzugriffe am Besten als Datenstruktur eignen.

Quellcode
dir: /sonstige_codes
file: myplot.py

Analog wie für die Schreibzugriffe werden die Lesezeiten in Abbildung 5.4 aufgeführt. Allgemein ist festzustellen, dass Lesezugriffe wesentlich schneller ausgeführt werden können als Schreibzugriffe. Ferner schneiden die SciPy Sparse Matrizen insbesondere bei hohen Füllgraden besonders schlecht ab. Die übrigen untersuchten Datenstrukturen weisen ein ähnliches Verhalten auf, wobei die NumPy-Arrays bei hohen Füllgraden etwas schlechter abschneiden als Dictionaries und Listen.

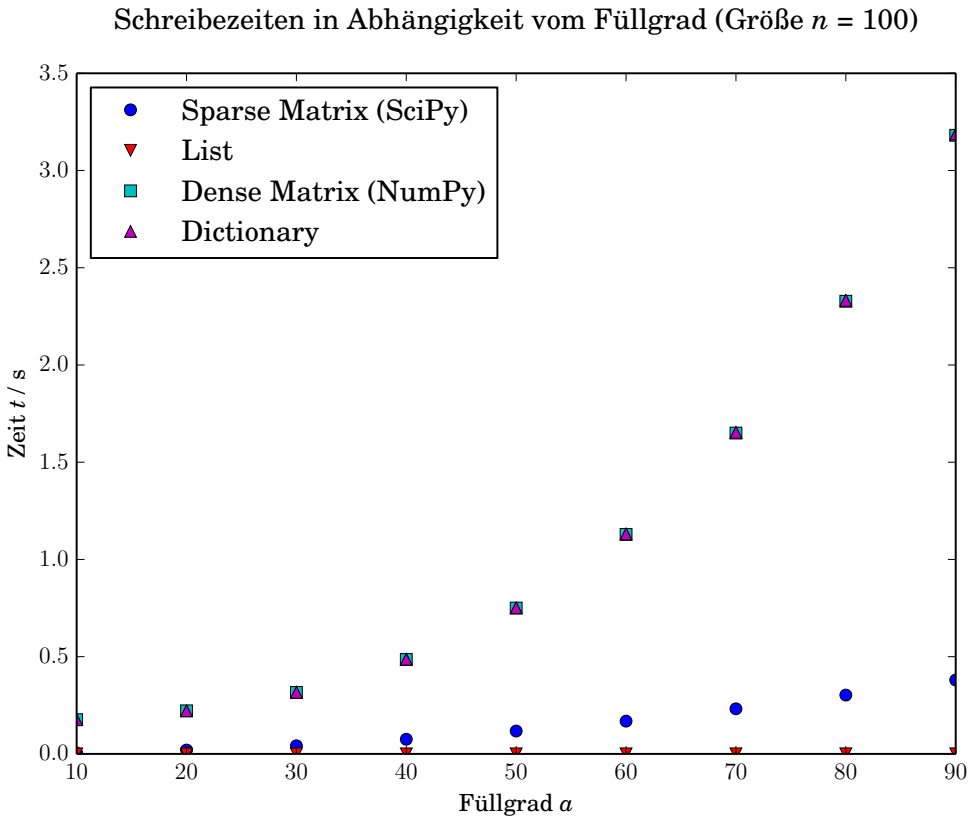


Abbildung 5.3.: Plot der Schreibzeiten gegen den Füllgrad bei einer Matrixgröße $n = 100$.

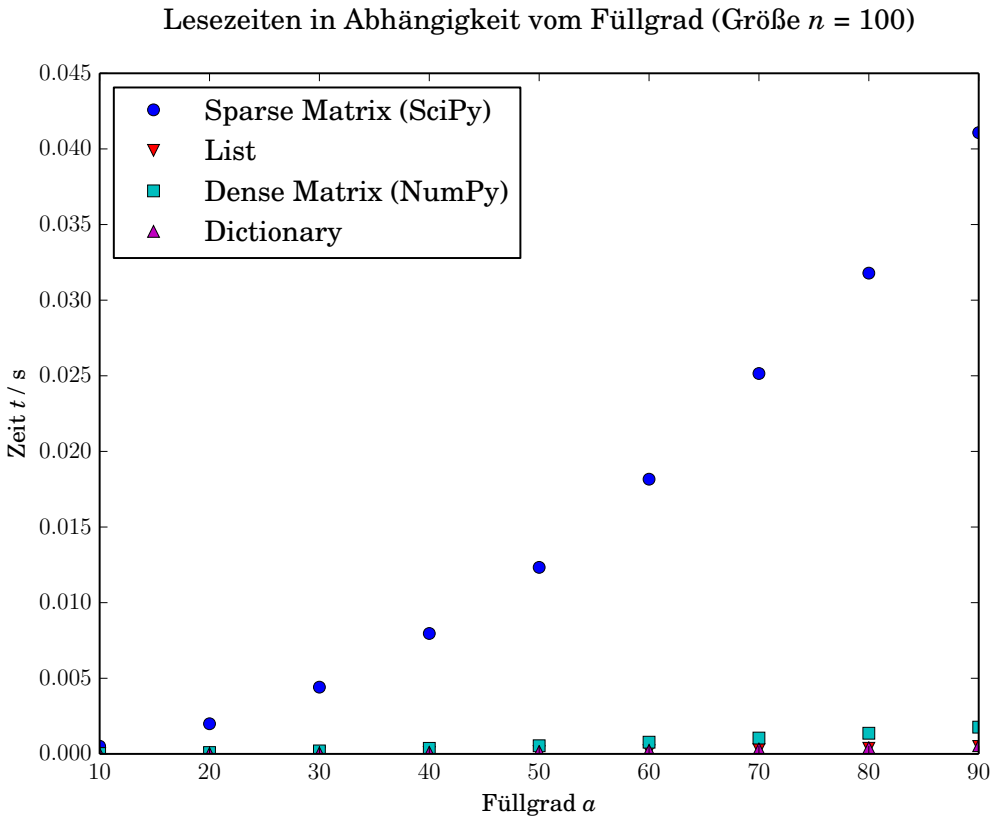


Abbildung 5.4.: Plot der Lesezeiten gegen den Füllgrad bei einer Matrixgröße $n = 100$.

Nach der Untersuchung der Zugriffszeiten erfolgt an dieser Stelle eine Untersuchung der genannten Datenstrukturen bzgl. Speicherplatzbelegung. Die Untersuchung erfolgt anhand einiger Konsolenexperimente, wobei die Speicherbelegung mit Hilfe der Systemüberwachung des verwendeten Betriebssystems (Linux Mint 17.1 „rebecca“) erfasst worden ist. Im Folgenden wurden alle Experimente bei einer Speicherbelegung größer als 5 GiB abgebrochen, da in allen oben vorgestellten Algorithmen die entsprechende Speicherstruktur zweimal benötigt wird, um auf den Zustand $t - 1$ zum Zeitpunkt t zuzugreifen. Entsprechend könnte ein gewöhnlicher Bürocomputer mit einer Arbeitsspeichergröße von 2 GiB $\sim$ 8 GiB die Daten nicht mehr effizient speichern und verarbeiten. Im Folgenden wird von einer quadratischen $n \times n$-Matrix ausgegangen.

Es ist festzustellen, dass aus Sicht der Speicherbelegung die Python Dictionaries besonders schlecht abschneiden, da bei ca. $n = 7000$ die 5 GiB-Schwelle überschritten wird. Die Python Listen weisen ein etwas besseres Speicherverhalten auf als die Dictionaries, überschreiten die genannte Schwelle allerdings bei etwa $n = 7000 - 8000$.

Das NumPy-Array sowie die (voll besetzte) SciPy-Matrix schneiden besser ab, wobei das NumPy-Array deutlich bessere Werte als die SciPy-Matrix bei gleicher Größe erzielt. Daher sind aus Sicht der Speicherbelegung die NumPy-Arrays am besten geeignet.

Aufgrund der bisherigen Betrachtungen ist festzustellen, dass die Python Listen aus Sicht der Rechenzeit am besten geeignet sind, während die NumPy-Arrays aus Sicht der Speicherbelegung am besten geeignet sind. Die NumPy-Arrays weisen jedoch ein besonders schlechtes Speicherzugriffsverhalten auf, wenn über alle Zellen iteriert wird. Gemäß der Anforderungen S3 und S4 ist eine effiziente Simulationsberechnung besonders wichtig. Daher wird hier die Rechenzeit stärker gewichtet als die Speicherbelegung, sodass in dieser Arbeit die Python Listen verwendet werden.

5.2.5. Fazit

Das „einfache Verfahren“ (Alg1) ist bzgl. Speicherplatzausnutzung und Rechenaufwand für dicht besetzte Zellulare Automaten das am Besten geeignete Verfahren. Das liegt daran, dass alle anderen Verfahren Rechenzeit und z. T. Speicherressourcen einsetzten um die Anzahl der Berechnungen zu senken, was bei dicht besetzten Zellularen Automaten obsolet ist.

Das „Indexverfahren“ (Alg2) ist aufgrund des hohen Implementierungsaufwandes sowie der algorithmischen Komplexität für das Projekt schlecht geeignet. Ferner verträgt sich das „Indexverfahren“ etwas schlechter mit den Anforderungen aus Kapitel 4. Daher muss die Idee an dieser Stelle verworfen werden.

Das „Stapelverfahren“ (Alg3) ist sehr gut für dünn besetzte Zellulare Automaten geeignet, da es unter Ausnutzung der Listen in Python leicht implementierbar ist. Das „Matrizenverfahren“ (Alg4) ist nur unter der Voraussetzung, dass die verwendete Programmierumgebung eine geeignete Datenstruktur zur Verfügung stellt, geeignet. Diese Voraussetzung ist auf Grundlage der Analyse des vorherigen Kapitels nur bedingt gegeben. Die Sparse Matrizen des SciPy-Paketes stellen zwar die erforderlichen Funktionalitäten bereit und weisen ein akzeptables Zugriffsverhalten auf, dennoch besteht das Risiko, dass es während der Simulation zu einem Speicherüberlauf kommt, wenn der Zellulare Automat zu stark wächst. Aus diesem Grunde wird in dieser Arbeit das „Stapelverfahren“ für dünn besetzte Zellulare Automaten verwendet.

Abschließend ist zu diskutieren, ob das „Stapelverfahren“ oder das „Kombiver-

fahren“ (Alg5) unter Ausnutzung des „Stapelverfahrens“ und des „einfachen Verfahrens“ genutzt werden. Der Vorteil der ersten Variante besteht darin, dass der Implementierungsaufwand geringer ist. Der Vorteil der zweiten Variante ist darin zu sehen, dass dicht besetzte Zellulare Automaten besser verarbeitet werden können als alleine beim „Stapelverfahren“. Andererseits hängt die Güte des „Kombiverfahrens“ stark von der richtigen Wahl zwischen den beiden Verfahren ab. Aufgrund der Tatsache, dass sowohl das „Stapel-“ als auch das „einfachen Verfahren“ einen relativ geringen Implementierungsaufwand aufweisen, wird in dieser Bachelorarbeit das „Kombiverfahren“ implementiert. Die notwendige Entscheidungsheuristik kann im Laufe der folgenden Arbeit durch experimentelle Untersuchungen entwickelt werden.

5.3. Datenhaltung

Gemäß Anforderung D1 sollen die Simulationsergebnisse in einer Datenbank gespeichert werden. Ferner sollen die Ergebnisse gemäß Anforderung D7 möglichst komfortabel in QGIS aus der Datenbank importiert und visualisiert werden können. Dazu ist es zum Einen erforderlich, dass sich die verwendete Datenbank einfach mit QGIS verknüpfen lässt, zum Anderen ist es erforderlich räumliche Indexe zu nutzen, um effizient auf die Daten zugreifen zu können. Des Weiteren wäre es für die Benutzer (Biologen) hilfreich, wenn es seitens QGIS eine Dokumentation / Nutzungsanleitung der genutzten Datenbank gäbe. Ferner muss die verwendete Datenbank für alle gängigen Plattformen verfügbar sein.

Gemäß des DB Manager Plugins können die Räumlichen Datenbanken PostGIS, SpatiaLite und Oracle Spatial genutzt werden [93], daher werden nur diese Datenbanken im Folgenden diskutiert. Im QGIS Training Manual [95] wird die Benutzung von PostgreSQL/PostGIS in den Modulen 15, 16 und 18 besprochen. Des Weiteren finden sich einige Hinweise zu SpatiaLite (einer räumlichen Erweiterung von SQLite). SpatiaLite [26], PostGIS [76, S. 404] und Oracle Spatial [14, S. 226] stellen einen R-Baum als räumlichen Index zur Verfügung. Oracle Spatial bietet zusätzlich einen Quadtree-Index [14, S. 201]. Gemäß Anforderung D1 soll die verwendete Datenbank die OGC-Standards umsetzen, da sich auch QGIS an diesen Standards hält [14, S. 5]. Das ist für alle drei Datenbanken der Fall [5, S. 307] und [14, S. 33]. Oracle Spatial erfüllt als einzige proprietäre Software Anforderung G7 nicht. In dieser Bachelorarbeit wird PostGIS verwendet, da es nach der obigen Diskussion alle Anforderungen erfüllt und in QGIS am besten dokumentiert ist.

Wie schon oben besprochen werden die Simulationsergebnisse in dieser Arbeit als Rasterdaten in einer PostGIS-Datenbank gespeichert. Entsprechend kann der in Kapitel 2.3.3 vorgestellte R-Baum als räumlicher Index genutzt werden [76, S. 404]. Alternativ können GeoHashes als Index in PostGIS genutzt werden. Dabei ist das Format WGS 84 festgelegt. Ferner sind GeoHashes in PostGIS auf kleine Objekte optimiert [76, S. 404 f.]. Daher wird in dieser Bachelorarbeit der R-Baum als räumlicher Index vorgezogen.

Zusätzlich zu der besprochenen räumlichen Komponente weisen die zu speichernden Daten eine zeitliche Dimension auf. Die in Kapitel 2 angesprochene Temporal Table Extension [122] führt zwar eine Versionierung auf temporalen Daten ein, bietet jedoch keine Indexierung. Die Versionierung bringt für diese Arbeit keinen Vorteil, da die Zeit nicht über die Transaktionszeit determiniert ist. Daher kann die Zeit als Anzahl an Tagen seit dem Beginn der Simulation gespeichert werden.

Ferner ist es wichtig alle Daten möglichst effizient aus der Datenbank auslesen zu können. Dazu bietet PostgreSQL zwei technische Lösungen an. Die erste Lösung besteht darin einen Index über mehrere Spalten zu nutzen [112, Kapitel 11.3]. Bei der zweiten Lösung werden verschiedene Indexe kombiniert [112, Kapitel 11.5]. Für eine Untersuchung der beiden Lösungen muss zwischen zwei verschiedenen Anfragen unterschieden werden. Bei Anfrage A wird die Raumregion ξ zum Zeitpunkt t abgefragt. Bei Anfrage B der ganze Zellraum zum Zeitpunkt t abgefragt. Für Anfrage A ist laut PostgreSQL-Dokumentation [112, S. 345] die erste Lösung besser als die zweite geeignet, da die Anfrage i. A. schneller verarbeitet werden kann. Problematisch ist jedoch, dass die Anfrage B mit der ersten Lösung nicht effizient verarbeitet werden kann. Dahingegen können beide Anfragen mit der zweiten Lösung effizient verarbeitet werden. Die zweite Lösung wird aufgrund der höheren Flexibilität in dieser Arbeit genutzt.

Neben dem bisher besprochenen Datenzugriff ist die Dateneingabe von Bedeutung, um die Simulation durchführen zu können. Dabei müssen die in Anforderung D3 angegebenen Daten eingegeben werden. Während der Anfangszustand nur vor dem Beginn der Simulation einmalig eingegeben werden muss, ist zu entscheiden, ob Daten wie bspw. die Temperatur global gesetzt werden sollen. Alternativ könnten die Daten für jeden Simulationstag (ggf. auch für jede Zelle) neu gesetzt werden. Dafür die Daten global zu setzen spricht, dass die Lösung sehr einfach zu implementieren ist. Dagegen spricht jedoch, dass das Werkzeug bzgl. eventueller Erweiterungen stark eingeschränkt wird. Daher wird in dieser Bachelorarbeit ein Zwischenweg gewählt,

der es erlaubt die Dateneingabe in der Zukunft verhältnismäßig leicht zu erweitern. Dabei werden die Daten zu jedem Zeitpunkt für jede zu simulierende Zelle von einem „Datenmanager“ abgefragt. In einer ersten Version werden dabei jedoch nur globale Daten verarbeitet, was für den Simulationskern jedoch nicht ersichtlich ist.

5.4. Architektur

In diesem Abschnitt wird der Architekturentwurf vorgestellt und diskutiert. Abbildung 5.5 zeigt ein Flussdiagramm des zu entwickelnden Werkzeugs. Im Flussdiagramm wird zwischen der Initialisierung und der eigentlichen Simulation unterschieden. Die Simulation wird für die beiden verschiedenen Verfahren jeweils durchlaufen. Tatsächlich unterscheiden sich die beiden Verfahren nur im in Abbildung 5.5 gelb unterlegten Simulationsschritt. Daher können beide Verfahren auf einer gemeinsamen Codebasis aufsetzen.

Ein grober Architekturentwurf ist in Abbildung 5.6 zu finden. Das System besteht aus den vier Einheiten Controller, Input, Output sowie den Simultionskern. Der Controller bildet dabei zum Einen die Schnittstelle nach „außen“ und zum Anderen die Steuerungseinheit der Systeminitialisierung sowie der eigentlichen Simulation. „Außen“ bezeichnet dabei Programme (Skripte, GUI’s oder Ähnliches), welche das hier entwickelte System als Backend nutzen. Die Inputeinheit wird benötigt, um für die Simulation benötigte Daten sowie ggf. den Initialzustand des Zellularen Automaten einzulesen. Der Output dient der persistenten Speicherung der Simulations(zwischen)ergebnisse. Der Simulationskern bildet das Herzstück dieser Architektur und implementiert den Zellularen Automaten sowie das MALCAM-Modell.

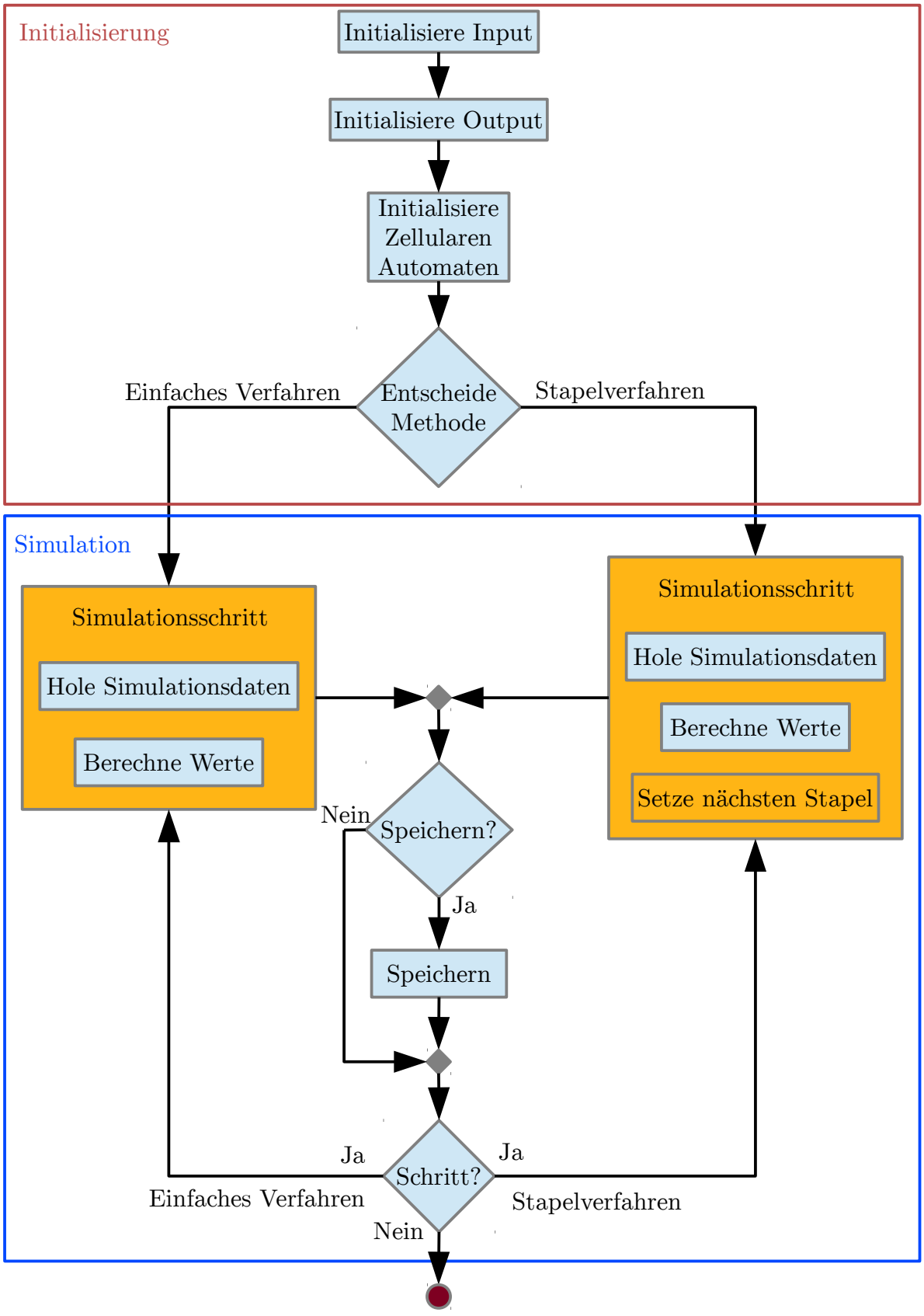


Abbildung 5.5.: Flussdiagramm des zu entwickelnden Werkzeugs. Die beiden Verfahren unterscheiden sich in den gelb markierten Aktionen.

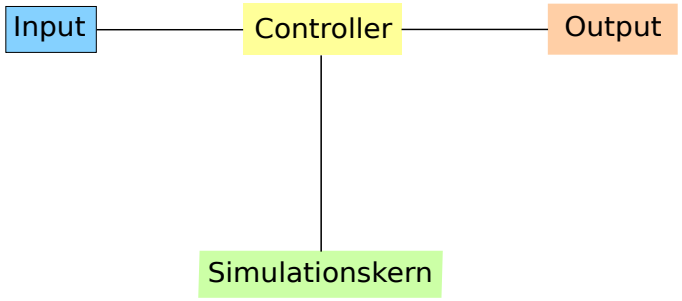


Abbildung 5.6.: Visualisierung des groben Architekturentwurfs.

Wie in Abbildung 5.7 zu erkennen ist, bildet die Klasse `CombiCA` den Kern der Simulation und implementiert das „Kombiverfahren“ (Alg5). Die abstrakte Klasse `AbstractCA` dient als Schnittstelle zwischen dem Controller und der Simulationsklasse. Der Vorteil dieses Vorgehens liegt darin, dass zu einem späteren Zeitpunkt weitere Simulationsklassen unabhängig von `CombiCA` implementiert werden können, welche bspw. nicht auf dem einfachen hier verwendeten Modell basieren. `CombiCA` entscheidet mit Hilfe der Methode `decideMethod` während der Initialisierung und dynamisch im Verlauf der Simulation, welches der implementierten Verfahren verwendet werden soll. Für dicht besetzte Zellulare Automaten steht die Klasse `EasyCA` zur Verfügung, welche das „einfache Verfahren“ (Alg1) implementiert. Das „Stapelverfahren“ (Alg3) wird von der Klasse `StackCA` implementiert. Die beiden Klassen `EasyCA` und `StackCA` erben von `AbstractMatrix`, die die Matrizen, welche die Zellularen Automaten repräsentieren, implementieren. Dadurch müssen die Matrizen sowie der Kern der Simulationslogik nicht zweimal implementiert werden. Das Setzen des Simulationsstapelspeichers für den nächsten Simulationsschritt im `StackCA` erfolgt durch Überschreiben der `_updateValuesOfCell` Funktion.

Die Anbindung der Simulationsregeln erfolgt in der abstrakten Klasse `AbstractCA`, da eine Übergangsfunktion für alle Zellularen Automaten definiert werden muss. Die Klasse `MALCAM` implementiert den gleichnamigen Regelsatz und erbt von der Klasse `AbstractRule`. Durch dieses Vorgehen lässt sich das entwickelte Werkzeug sehr einfach um weitere Regelsätze erweitern. Die Felder `name` und `cite` der Oberklasse `AbstractRule` müssen von den Erben während der Initialisierung mit sinnvollen Werten gesetzt werden. Beide Felder könnten zu einem späteren Zeitpunkt genutzt werden um (verschiedene) Regelsätze in einer GUI unterscheiden zu können. Zusätzlich zum `MALCAM` Regelsatz ist in Abbildung 5.7 eine Klasse `TestRule` zu erkennen, wie deren Name andeutet dient diese Regel lediglich dem Vereinfachen von Tests. Dieser Regelsatz implementiert *keine* gängige Regel zur Berechnung des

Ausbreitungsverhaltens von Mücken und ist somit nicht für die produktive Nutzung vorgesehen. Auf die Implementierung eines Regelparsers analog wie bei Klich [53] wird verzichtet, da eine Erweiterbarkeit um neue Regelsätze aufgrund der gewählten Architektur gegeben ist und sich bei der bisherigen Nutzung des Werkzeugs kein Bedarf herausgestellt hat.

Die Datenausgabe erfolgt mittels der Klasse `DataSet`. Dabei handelt es sich um eine nicht leere Menge an „Ausgabeobjekten“. Ein „Ausgabeobjekt“ ist dabei ein Erbe der Klasse `AbstractData`. In dieser Arbeit wird eine Anbindung an die freie Datenbank PostGIS bereitgestellt (Klasse `DBData`). Als weitere Ausgabemöglichkeit zur Speicherung in CSV-Dateien wird hier die Klasse `SimpleData` implementiert. `SimpleData` dient primär der Vereinfachung des Testens, kann jedoch u. U. im Produktiveinsatz genutzt werden. Die Nutzung der abstrakten Klasse `AbstractData` dient der Erweiterbarkeit des Werkzeugs. Neben der Ausgabe soll es möglich sein, den Zustand eines Zellularen Automaten zu einem beliebigen Zeitpunkt t über die Datenschnittstelle zu laden, um eine unterbrochene Simulation fortführen zu können.

Die Dateneingabe erfolgt über die abstrakte Klasse `AbstractInput`, welche von den Eingabe Klasse(n) implementiert werden muss. `AbstractInput` stellt die Methoden `getInitialState()` für den Startzustand des Zellularen Automaten, sowie `getDataOfCell(...)` für die Simulationsdaten einer Zelle zum Zeitpunkt t zur Verfügung. Dieses Vorgehen bietet die Möglichkeit das Werkzeug bzgl. der Dateneingabe zu einem späteren Zeitpunkt zu erweitern, ohne den Simulationskern verändern zu müssen. Ein denkbares Szenario wäre der Einsatz von Insektiziden ab einem bestimmten Zeitpunkt der Simulation, welcher bspw. die Mortalitätsrate der erwachsenen Mücken erhöht. Des Weiteren ließen sich Temperaturschwankungen mit einbeziehen. In dieser Arbeit werden die angesprochenen Erweiterungen nicht betrachtet. Die Dateneingabe erfolgt hier ausschließlich über die Klasse `SimpleInput`, welche den Startzustand des Zellularen Automaten aus einer CSV-Datei einliest. Die Simulationsparameter sind in der Klasse `SimpleInput` global definiert. Dadurch wird es möglich die Simulationsparameter schnell manuell und reproduzierbar anzupassen, sodass das Testen des Werkzeugs vereinfacht wird. Das Auslesen von Daten aus einer Datenbank bzw. GUI ist nicht Teil dieser Arbeit, wäre jedoch aufgrund der gewählten Architektur technisch umsetzbar.

Im Folgenden werden einige Überlegungen zu Koordinaten angestellt. Dazu werden hier einige Begrifflichkeiten definiert. In dieser Arbeit bezeichnen *interne Koordi-*

naten die vom Simulationskern verwendeten Koordinaten. Diese entsprechen aus technischen Gründen den Listenindexen aus Python, so dass für die internen Koordinaten einer Zelle (x, y) mit $x \in \{0, \ldots, n-1\}$ und $y \in \{0, \ldots, m-1\}$ gilt, wobei n der größte Zellindex auf der Abszisse und m der größte Zellindex auf der Ordinate sind. Die Zelle $(0, 0)$ sei die linke untere Zelle des Zellularen Automaten. *Äußere Koordinaten* bezeichnen Koordinaten der realen Welt (bspw. in Metern), welche außerhalb dieses Werkzeugs gelten. In PostGIS können Rasterdaten zusätzlich zu den oben definierten äußeren Koordinaten über Zellindexe, in dieser Arbeit als *virtuelle Koordinaten* bezeichnet, indexiert werden. Für eine Zelle in virtuellen Koordinaten gilt $(\tilde{x}, \tilde{y})$ mit $\tilde{x} \in \{1, \ldots, n\}$ und $\tilde{y} \in \{1, \ldots, m\}$, wobei n der größte Zellindex auf der Abszisse und m der größte Zellindex auf der Ordinate sind. In dieser Arbeit werden für PostGIS ausschließlich virtuelle Koordinaten verwendet, um eine Koordinatensystem unabhängige Implementierung zu gewährleisten. Es bleibt dem Nutzer überlassen ein geeignetes (längentreues) Koordinatensystem als äußere Koordinaten in PostGIS zu wählen.

Der Simulationskern des Werkzeugs rechnet mit internen Koordinaten. Es wird angenommen, dass die Zellen des Zellularen Automaten jeweils die gleiche Höhe und Breite aufweisen und gitterartig in einem zweidimensionalem Universum angeordnet sind. Der Vorteil dieses Vorgehens besteht in der vereinfachten Implementierung des Simulationskerns, da keine Information über die äußeren Koordinaten gespeichert werden müssen. Das bedeutet, dass bei der Ein- und Ausgabe der Daten ggf. eine Koordinatentransformation durchgeführt werden muss. Dazu dient das Interface `CoordinateTransformer`, welches von abgeleiteten Klassen implementiert werden muss. Das Interface sieht eine Funktion vor, welches äußere- bzw. virtuelle Koordinaten auf innere Koordinaten abbildet (`calcInnerCoordinate`), `calcOuterCoordinate` ist entsprechend die Umkehrfunktion. In dieser Version werden als `CoordinateTransformer` ein `TestTransformer` sowie der `DBTransformer` implementiert. Dabei bildet der `DBTransformer` auf die oben definierten virtuellen Koordinaten der Datenbank ab. Für PostGIS Rasterdaten kann ein Zugriff auf die Zellen analog zu den inneren Koordinaten erfolgen, jedoch beginnt der Index mit dem Wert 1. In Python hingegen beginnt der Index mit 0. Daher rechnet der `DBTransformer` zwischen diesen beiden Indexsystemen um. Sind die äußeren Koordinaten identisch zu den inneren Koordinaten, kann der `TestTransformer` genutzt werden. Das Transformerkonzept ist während der Implementierung entwickelt worden, um die Koordinatentransformation unabhängig von den Ein- und Ausga-

beschnittstellen durchführen zu können. Dadurch kann Coderedundanz vermieden werden.

Der `SimulationController` hat die Aufgabe die einzelnen Teile zu initialisieren und den Simulationsablauf inklusive Datenein- und ausgabe zu steuern. Des Weiteren bietet der `SimulationController` die benötigten Schnittstellen, um das Werkzeug zu einem späteren Zeitpunkt in einer anderen Umgebung (z. B. einer GUI) zu nutzen. In dieser Arbeit erfolgt die Nutzung ausschließlich über ein Python-Skript, welches es ermöglicht das Werkzeug zu testen.

Die Simulationsalgorithmen werden an dieser Stelle kurz diskutiert. Wie in Abbildung 5.8 zu erkennen ist, liest das „einfache Verfahren“ (Alg1) für alle Zellen die Simulationsdaten der aktuell betrachteten Zelle sowie deren Nachbarn ein und berechnet die neuen Werte (Anzahl an weiblichen Mücken und Anzahl an weiblichen Mückenlarven). Anschließend wird die Anzahl an Tagen um eins erhöht und ggf. wird der aktuelle Zustand abgespeichert. Dieses Vorgehen wird wiederholt, solange noch Simulationsschritte zu verarbeiten sind. Das „Stapelverfahren“ (Alg3) unterscheidet sich vom „einfachen Verfahren“ wie in Abbildung 5.9 zu erkennen ist lediglich dadurch, dass es nur die in der Warteschlange gespeicherten Zellindexe neu berechnet. Anschließend wird der Simulationsstapelspeicher für den nächsten Zeitschritt aufgebaut.

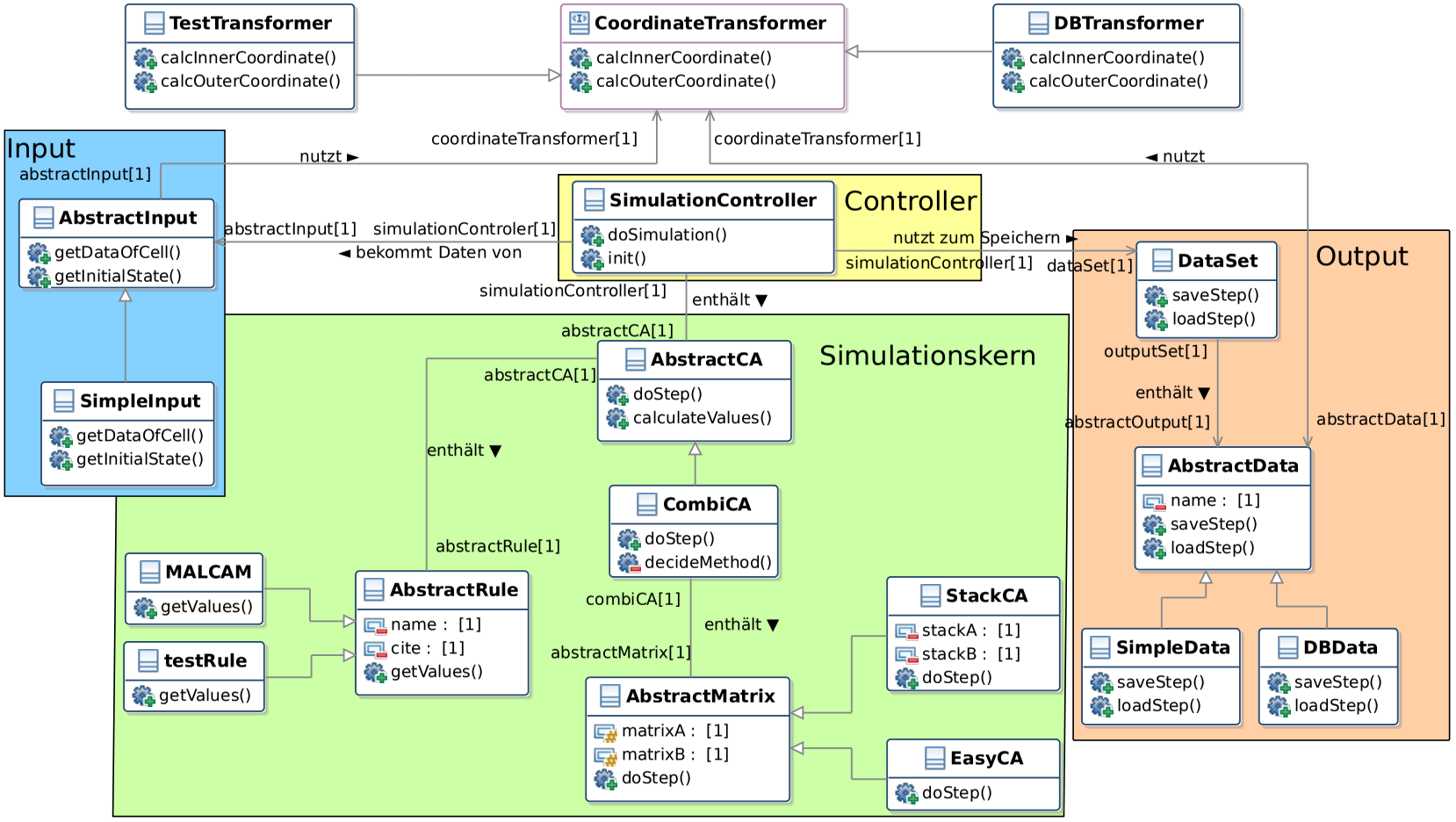


Abbildung 5.7.: Entwurfsklassendiagramm des Werkzeugs (die Attribute und Methoden der Klassen werden nicht in Gänze aufgeführt).

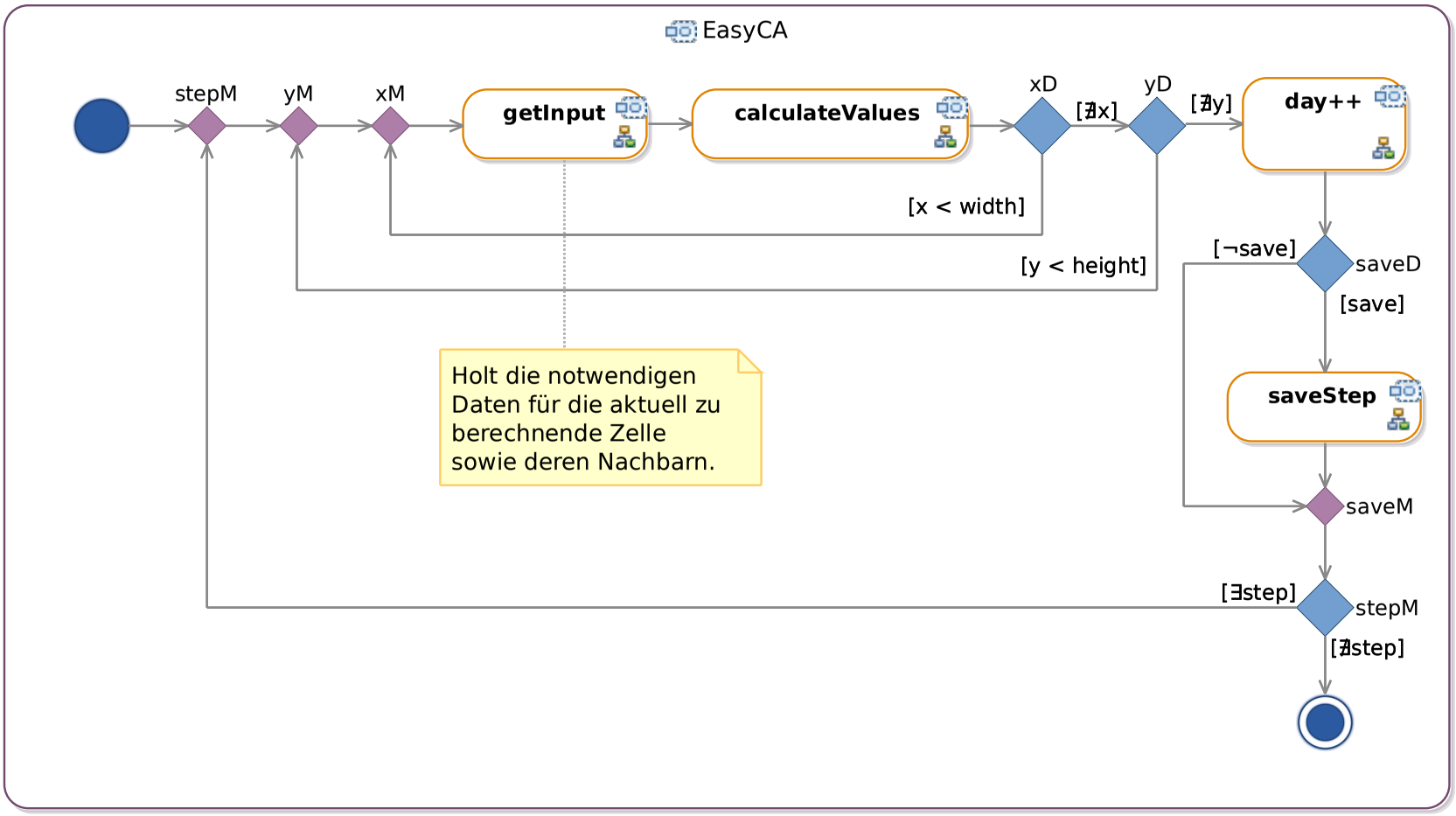


Abbildung 5.8.: Entwurfsaktivitätsdiagramm des „einfachen Verfahrens“ (Alg1).

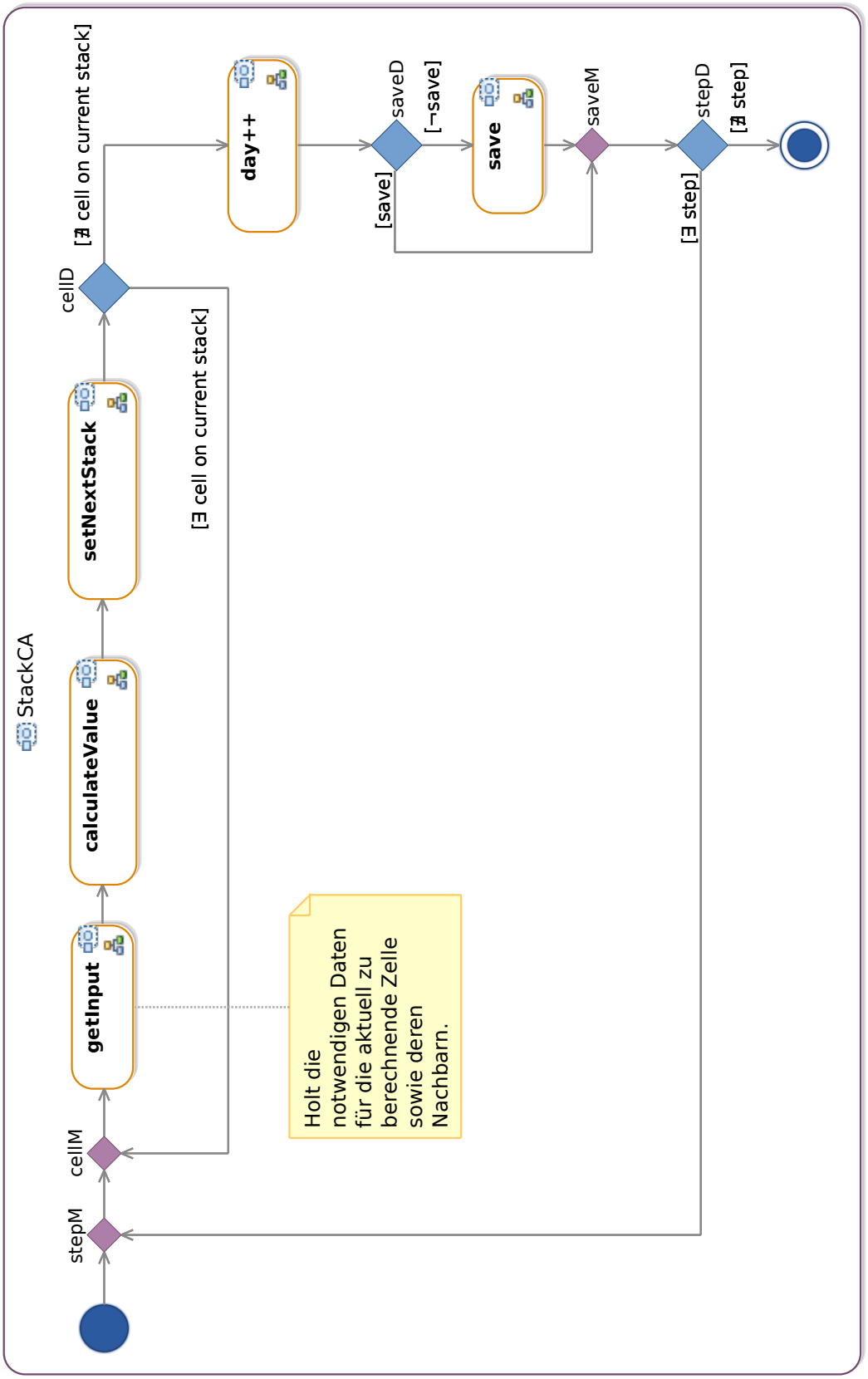


Abbildung 5.9.: Entwurfsaktivitätsdiagramm des „Stapelverfahrens“ (Alg3).

Kapitel
6

Implementierung

Dieses Kapitel behandelt Implementierungsaspekte und -entscheidungen. Entsprechend werden Anmerkungen gegeben, wie Implementierungsalternativen möglichst einfach umgesetzt werden könnten.
Das Vorgehen während der Entwicklung des Werkzeugs wird im ersten Abschnitt vorgestellt. Im zweiten Abschnitt werden die wichtigsten Schnittstellenaufrufe zu den einzelnen Klassen vorgestellt. Der dritten Abschnitt zeigt exemplarisch auf, wie die Datenformate für die interne- und externe Kommunikation des Werkzeugs definiert sind. Der vierte Abschnitt stellt die Implementierung des MALCAM-Modells, wie es theoretisch in Kapitel 3.3 besprochen wurde, vor. Der letzte Abschnitt zeigt auf, wie die Datenbankanbindung realisiert worden ist. Des Weiteren gibt der letzte Abschnitt einige Hinweise, wie die Datenbank konfiguriert sein muss.

Zunächst werden einige Konventionen aufgeführt, die in diesem Kapitel Verwendung finden. Zusätzlich zu den üblichen Konventionen für Klassen- und Funktionennamen werden hier einige weitere Konventionen benötigt. Die im Entwurfskapitel modellierten Interfaces und abstrakten Klassen lassen sich im Gegensatz zu anderen objektorientierten Programmiersprachen, wie bspw. Java, in Python nicht eins zu eins umsetzen. Per Konvention werden für modellierte Interfaces Python-Klassen verwendet, welche von `object` erben und keine aus implementierten Funktionen beinhalten. Abstrakte Klassen beinhalten in dieser Arbeit Funktionen, welche nicht aus implementiert worden sind. Im Gegensatz zu Interfaces enthalten sie jedoch zusätzlich Attribute und Klassenfunktionen. Einige Beispiele für Interfaces, abstrakte Klassen sowie einige Anmerkungen zum Zugriff von Klassenfunktionen und -attributen finden sich in Anhang B.2.

Im Folgenden werden in Sequenzdiagrammen einige Funktionalitäten wie gETINPUT als Platzhalter für andere Sequenzdiagramme verwendet. Ferner werden bei Sequenzdiagrammen in dieser Arbeit, sofern nicht anders angegeben, stets die abstrakten Oberklassen angegeben. Die abstrakten Klassen sind als Platzhalter für die erbenden Klassen zu verstehen. Einzige Ausnahme bildet der `CombiCA`, welcher von `AbstractCA` abgeleitet ist. Sollten zu einem späteren Zeitpunkt Zellulare Automaten implementiert werden, welche nicht auf dem hier verwendeten einfachen Modell basieren, so stimmen die internen Funktionenaufrufe u. U. nicht mit den hier für `CombiCA` dargestellten überein. Ferner wird als Kurzbezeichnung für for-Schleifen der ∀-Quantor genutzt.

Das Werkzeug nutzt für unterschiedliche Funktionalitäten konstante String-Schlüsselworte. Um eine redundante Definition dieser Schlüsselworte zu vermeiden, werden diese im Paket `constants.py` definiert. Ferner müssen in unterschiedlichen Modulen des Werkzeugs leere Zellulare Automaten initialisiert werden. Dafür wird im Paket `zeroCA.py` eine gleichnamige Funktion definiert.

Es sei angemerkt, dass für die Entwicklung dieser Arbeit eine virtuelle Maschine auf Basis von Oracle VirtualBox [79] aufgesetzt worden ist. Eine kurze Dokumentation der verwendeten Software findet sich in Anhang A, die virtuelle Maschine ist auf der beigelegten DVD (s. S. 160) zu finden. Der virtuellen Maschine werden 2 GiB RAM (Random-Access Memory), ein CPU-Kern (Central Processing Unit) sowie 8 GiB Festplattenspeicher zugewiesen. Als Betriebssystem wird Bodhi Linux

[10] verwendet. Weitere für dieses Kapitel relevante Werkzeuge sind Python [92], PostgreSQL [112] / PostGIS [99] und Eclipse [18].

6.1. Vorgehen

In diesem Abschnitt wird das Vorgehen während der Softwareentwicklung beschrieben. Abbildung 6.1 zeigt, dass das Vorgehen im Wesentlichen einem evolutionärem Wasserfallmodell entspricht. Nach der Festlegung der Anforderungen wird der Entwurf definiert. Während die Anforderungen nicht mehr verändert werden, sind Teile des Entwurfs während der weiteren Entwicklung angepasst worden, wobei jedoch ein Großteil der ersten Version erhalten geblieben ist. Nach Definition des Entwurfs ist das Werkzeug in drei Abschnitten entwickelt worden:

1. Entwicklung des Simulationskerns, bestehend aus dem Zellularem Automaten, der Testregel, der Dateneingabe, der CSV-Ausgabe und des Controllers.
2. Entwicklung der Datenbankanbindung.
3. Implementierung des MALCAM-Modells.

Nach dem ersten Abschnitt ist der Entwurf angepasst worden, um das Nutzen der Regel für den Simulationsschritt besser erweiterbar zu gestalten. Ferner sind einige Umbenennungen von Klassen und Funktionen vorgenommen worden, um deren Semantik im Entwurf besser zu verdeutlichen. Nach der Implementierung eines der drei oben genannten Abschnitte wird ein erster Skripttest durchgeführt, um zu überprüfen, ob das System funktionsfähig ist. Durch dieses an die testgetriebene Softwareentwicklung angelehnte Vorgehen werden Implementierungsfehler möglichst früh aufgedeckt, ohne Unittests entwickeln zu müssen. Nach diesen Skripttests schließt sich die Phase des Refactorings an, in der Verbesserungen am Code vorgenommen werden, welche jedoch nicht die Semantik verändern. Dabei handelt es sich bspw. um das Auslagern von Funktionalitäten in neue Funktionen oder Klassen, Umbenennungen von Attributen und Funktionen sowie die Optimierung der Kommentare. Danach schließt sich die Phase der Unittests an, in der automatisierte Tests entwickelt werden, um die Qualität der Software sicherzustellen. Das Benchmarking wird nach Fertigstellung der Software durchgeführt und dient zum Einen der Qualitätsprüfung bzgl. der Recheneffizienz zum Anderen die Heuristik für die Wahl des zu verwendenden Verfahrens abzuleiten. Die Knoten Implementierung

und Refactoring in Abbildung 6.1 entsprechen der Implementierung des klassischen Wasserfallmodells. Analoges gilt für die Tests.

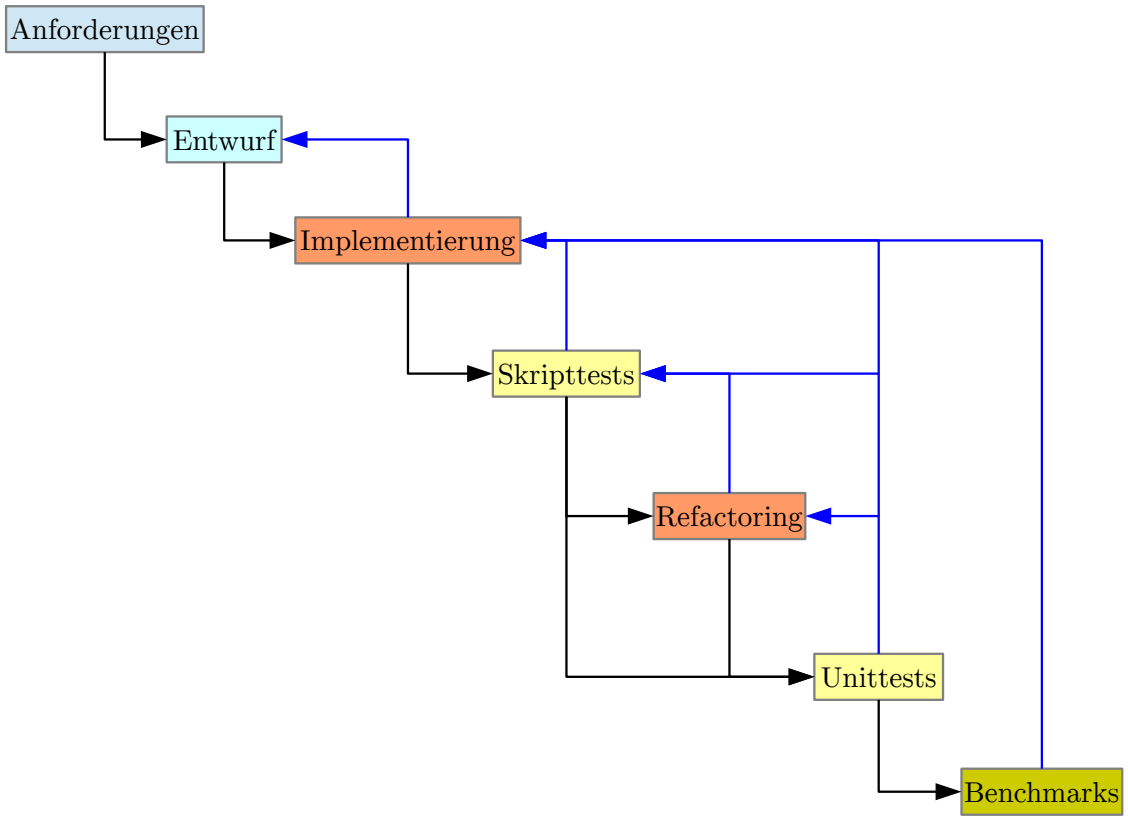


Abbildung 6.1.: Visualisierung des gewählten Vorgehens. Die blau eingezeichneten Pfeile bezeichnen einen Rücksprung zu einem vorherigem Abschnitt des Vorgehensmodells.

6.2. Funktionenaufrufe während der Simulation

In diesem Abschnitt wird beschrieben, wie die Schnittstellen zu den einzelnen Klassen (s. Kapitel Entwurf Abbildung 5.7) während der Simulation aufgerufen werden.

Die Simulation wird durch den Aufruf der Funktion `doSimulation` des `SimulationControllers` von außen (bspw. von einem Skript oder einer GUI) gestartet. Als Parameter werden die Anzahl der zu simulierenden Tage und das Intervall, in dem die Daten gespeichert werden, übergeben. Wie in Abbildung 6.2 zu erkennen ist, wird zu Beginn der Simulation der Initialzustand gespeichert, sofern während der Initialisierung das Attribut `__enableSaving` nicht auf `False` gesetzt worden ist. Anschließend beginnt die eigentliche Simulation. Dabei wird für jeden durchzuführenden

Simulationstag die Funktion `doStep` der Klasse `CombiCA` aufgerufen. `CombiCA` ruft je nachdem welches Verfahren für die Simulation genutzt werden soll die `doStep` Funktion von `EasyCA` bzw. `StackCA` auf.

In der Implementierung dieser Arbeit wird die Entscheidung, welches Verfahren genutzt werden soll während der Initialisierung und ggf. während der Simulation getroffen. Während der Initialisierung kann der Nutzer den Parameter `methodToUse` übergeben und damit die zu nutzende Methode selber vorgeben. In diesem Fall wird während der Simulation nicht geprüft, ob das Verfahren gewechselt werden muss.

Die geerbte `_performCalculation` Funktionen von `EasyCA` und `StackCA` unterscheiden sich darin, welche Zellen für die Simulation ausgewählt werden. In Abbildung 6.2 werden die beiden unterschiedlichen Verfahren mittels ∀ cells subsumiert. Tatsächlich handelt es sich beim `EasyCA` um eine Doppelschleife über alle Zellen und beim `StackCA` um den Stapelmechanismus aus Abschnitt 5.2.1 (Alg3). Nach der Durchführung des Simulationsschrittes wird der Zellulare Automat ggf. gespeichert. Die Darstellung in Abbildung 6.3(b) ist an dieser Stelle etwas vereinfacht worden, um das Diagramm übersichtlicher zu halten. Tatsächlich wird nur gespeichert, wenn das Attribut `__enableSaving` nicht auf `False` gesetzt worden ist und der aktuelle Schritt gespeichert werden soll (bzw. der letzte Simulationsschritt ist).

In Abbildung 6.2 sind die Funktionalitäten zum Holen der für die Simulation benötigten Eingabedaten (gETINPUT) sowie zur Speicherung der Daten (sAVEDATA) abgekürzt worden. Eine detailliertere Version für gETINPUT ist in Abbildung 6.3(a) zu finden. Wird die Funktion `getDataOfCells` mit einer nichtleeren Menge an Zellindexen im `EasyCA` oder `StackCA` aufgerufen, so werden für jede übergebene Zelle der alte Zustand mittels `getOldCell` sowie die eigentlichen Simulationsdaten mittels `getDataOfCell` besorgt. Der Aufruf von `getDataOfCell` ist in Abbildung 6.3(a) zur Verbesserung der Übersichlichkeit etwas vereinfacht worden. Tatsächlich wird in `AbstractMatrix` eine interne `getDataOfCell` Funktion aufgerufen, welche die Anfrage zum `CombiCA` weiterleitet. Diese Designentscheidung ist getroffen worden, um bei möglichen späteren Erweiterungen das Holen der Simulationsdaten einer Zelle über eine Funktion realisieren zu können. `CombiCA` leitet die Datenanfrage weiter an den `SimulationController`, welcher Zugriff auf das eigentliche Dateneingabemodul hat (in diesem Fall `SimpleInput`) und von diesem die angeforderten Daten holen kann. Abschließend werden die Daten für alle abgefragten Zellen in der Funktion `getDataOfCells` aggregiert und als Liste zurückgegeben. Es sei an-

gemerkt, dass die Funktionen `getDataOfCells`, `getOldCell` und `getDataOfCell` in den Klassen `EasyCA` und `StackCA` von deren gemeinsamen Oberklasse geerbt `AbstractCA` werden.

Die in Abbildung 6.3(b) dargestellte Speicherfunktionalität wird im `SimulationController` während der Simulation aufgerufen. Dabei handelt es sich um eine Weiterleitung des Zellularen Automaten sowie des zu speichernden Simulationstages über den `DataSet` an `SimpleData` bzw. `DBData`, welche die eigentliche Speicherung vornehmen. `DataSet` ermöglicht es die Simulationsergebnisse gleichzeitig in verschiedenen Formaten zu speichern und dient daher der Funktionserweiterung des Werkzeugs.

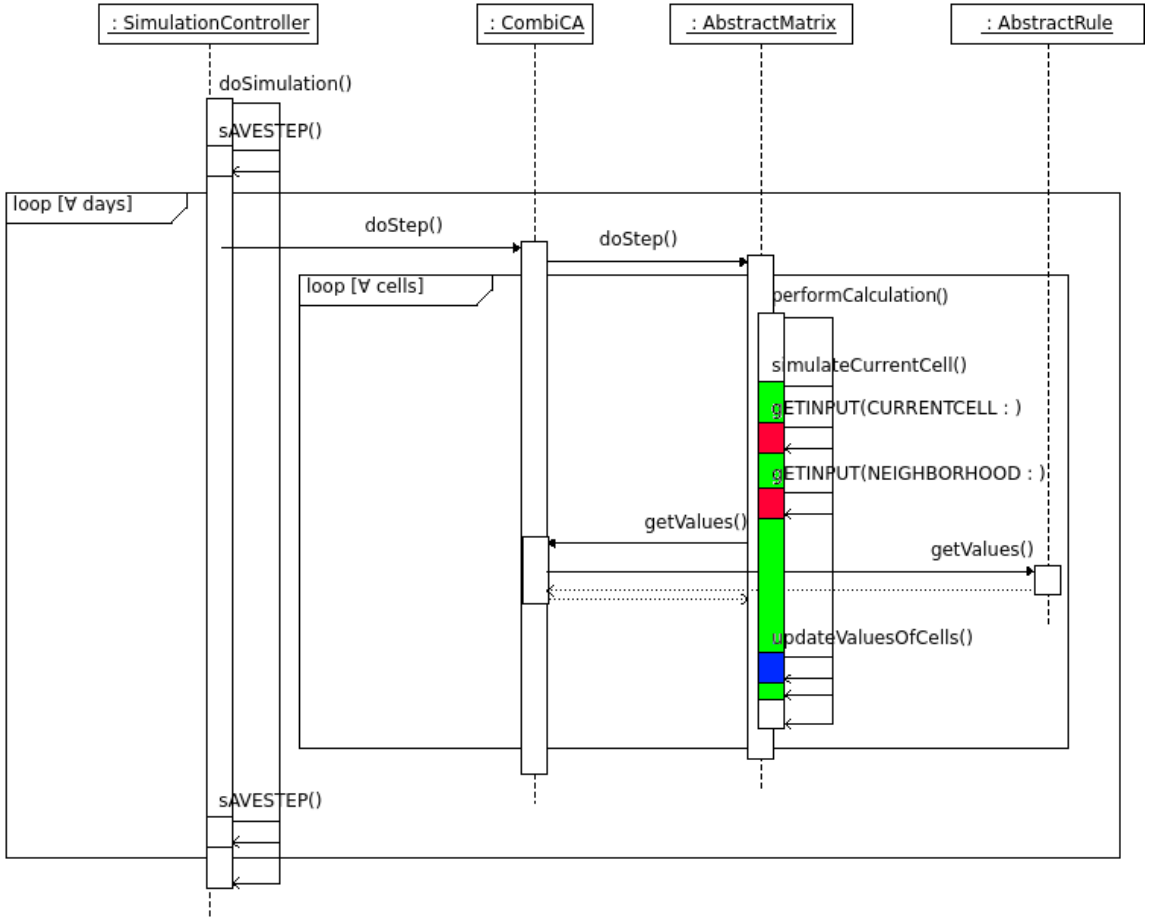
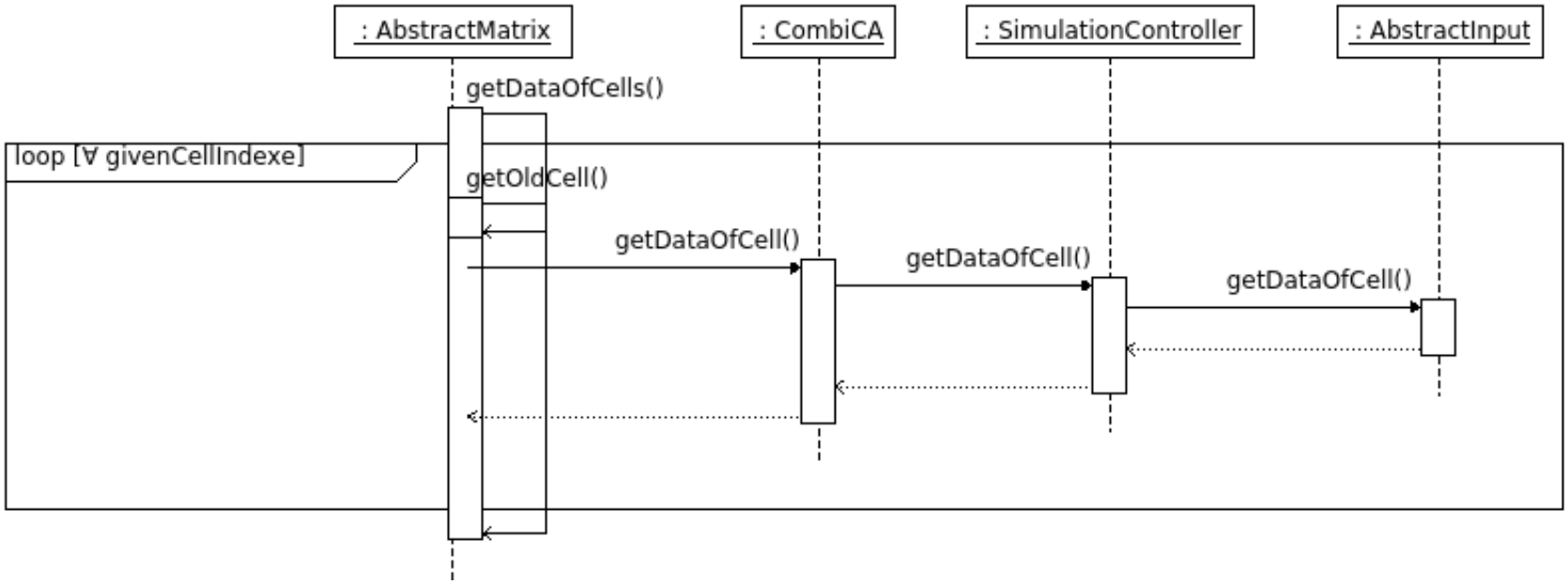
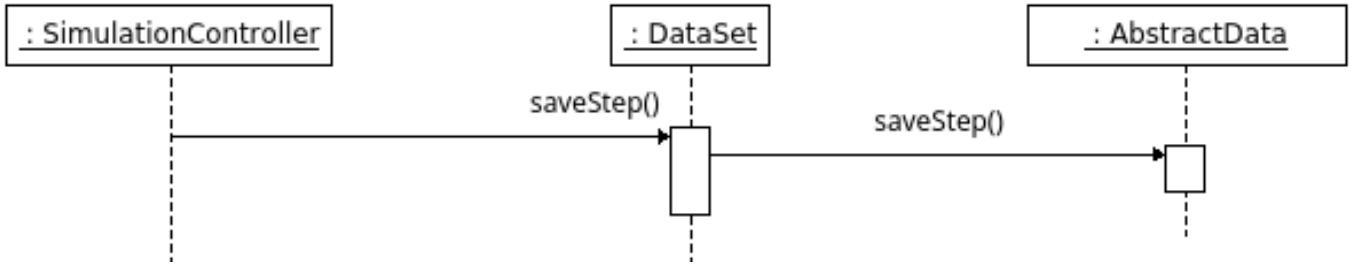


Abbildung 6.2.: Sequenzdiagramm der Simulation (die Funktionsparameter werden hier nicht aufgeführt). Die Funktionen gETINPUT(...) (bzw. sAVEDATA(...)) beziehen sich auf die in Abbildung 6.3(a) (bzw. Abbildung 6.3(b)) dargestellten Vorgänge und dienen lediglich als Abkürzungen.



(a) Sequenzdiagramm von gETINPUT.



(b) Sequenzdiagramm von sAVEDATA.

Abbildung 6.3.: Sequenzdiagramm von gETINPUT und sAVEDATA (die Funktionsparameter werden hier nicht aufgeführt).

6.3. Datenformate

6.3.1. Initialisierung

Die Initialisierung des Werkzeugs erfolgt über den `SimulationController`. Den `init*` Funktionen werden unter anderem `*Args` Parameter übergeben. Dabei handelt es sich um Python Dictionaries, sodass der Funktion eine verschiedene Anzahl benötigter Parameter übergeben werden können. Der Grund für dieses Vorgehen liegt in einer vereinfachten Erweiterbarkeit der jeweiligen Funktionalitäten. Um `SimpleData` zu initialisieren, wird bspw. ein Verzeichnis während der Initialisierung übergeben, in dem die CSV-Ausgabe gespeichert werden soll. Der Parameter „Verzeichnis“ wird für eine Datenbankanbindung jedoch nicht benötigt. Um die Initialisierung der Datenausgabe über eine Funktion im `SimulationController` zu realisieren, müsste ohne die Nutzung von Dictionaries für jede neue Ausgabeklasse die Signatur der Funktion `initData` im `SimulationController` angepasst werden (entsprechend natürlich auch alle Aufrufe dieser Funktion). Dadurch würde die Erweiterbarkeit der Software sehr stark leiden. Durch die Nutzung von Dictionaries muss die Signatur der Funktion `initData` nicht verändert werden, sodass eine Erweiterung der Software leicht geschehen kann. Der Grund liegt in der Funktionsweise von Python Dictionaries, welche es erlauben eine Zuordnung von Schlüsseln zu Werten zu definieren [121, S. 221]. Das Funktionsprinzip wird hier am Beispiel von `initData` aufgezeigt, wobei hier ein `DBData` Objekt initialisiert werden soll. Die anderen Initialisierungsfunktionen sind nach dem selben Schema aufgebaut, eine entsprechende Beschreibung ist dem Quellcode bzw. dessen Dokumentation zu entnehmen. Für die Übergabe des Parameterdictionaries wird der Parameter `dataArgs` verwendet. Eine Übersicht der geforderten Einträge für die Initialisierung von `DBData` findet sich in Tabelle 6.1. Dictionaryeinträge haben in Python die Form `key:value`. In dieser Arbeit werden die `key`-Einträge als Stringkonstanten in dem Paket `constants.py` bereitgestellt. Dadurch kann zum Einen eine redundante Definition vermieden werden und zum Anderen wird die Fehlerwahrscheinlichkeit gesenkt.

Tabelle 6.1.: Auflistung der Schlüssel (key, sowie die dazugehörige key-Konstante aus dem `constants.py`-Paketes) sowie deren Bedeutung für `dataArgs` zur Initialisierung eines `DBData` Objekes. Die Parameter für die PostGIS Datenbank sind in [98, Kapitel 9.3] dokumentiert.

key	**key-Konstante**	**Bedeutung**
name	`NAMEARGS`	Name der Tabelle.
dbname	`DBNAMEARGS`	Name der Datenbank
user	`USERARGS`	Name des Datenbankennutzers.
host	`HOSTARGS`	Host der Datenbank (z. B. localhost für den lokalen Rechner).
password	`PASSWORDARGS`	Passwort des Datenbankennutzers.
newTable	`NEWTABLEARGS`	Soll eine neue Tabelle erstellt werden?
width	`WIDTHARGS`	Anzahl der Zellen auf der Horizontalen.
height	`HEIGHTARGS`	Anzahl der Zellen auf der Vertikalen.
upperleftx	`UPPERLEFTXARGS`	Abszissen Koordinate der linken oberen Zelle.
upperlefty	`UPPERLEFTYARGS`	Ordinaten Koordinate der linken oberen Zelle.
scalex	`SCALEXARGS`	Breite einer Zelle.
scaley	`SCALEYARGS`	Höhe einer Zelle.
skewx	`SKEWXARGS`	Rotation der Zellen um die Abszisse.
skewy	`SKEWYARGS`	Rotation der Zellen um die Ordinate.
srid	`SRIDARGS`	„Spatial Reference ID“ Definition des Koordinatensystems, wenn `unknown` gesetzt wird, ist das Koordinatensystem unbekannt.

6.3.2. Simulation

Während der Simulation müssen zwischen den einzelnen Komponenten des Werkzeugs Daten ausgetauscht werden. Dazu werden stets Python Listen verwendet, da diese schnell zugegriffen werden können. Um die für die Simulation benötigten Daten einer Zelle zu laden, erfolgt eine `getDataOfCell(day, cell)` Anfrage an `SimpleInput`, wobei `cell` den Index der Zelle repräsentiert. Der Rückgabewert ist eine Liste welche bspw. die Temperatur, Mortalitätsrate und Entwicklungsrate der abgefragten Zelle enthält. Um die neuen Werte einer Zelle zu berechnen, wird die `calculateValues(cell, neighborhood)` aufgerufen. Dabei repräsentiert `cell` den alten Zustand sowie die Simulationsdaten der aktuellen Zelle und `neighborhood` ist eine Liste, welche die Daten und Zustände der Nachbarzellen enthält. Der Rückgabe-

wert ist eine Liste von Listen, welche für die zu betrachtende Zelle und deren Nachbarn die Zellkoordinaten, sowie das Δ der Mücken(-larven)anzahl angibt. Das Aktualisieren der Zellen ist in Listing 6.1 dargestellt. In der Funktion `_updateValuesOfCell` (s. Z. 6) erfolgt das Aktualisieren einer Zelle. Die übergebene Zelle ist eine Python Liste mit den Einträgen Abszissenkoordinate, Ordinatenkoordinate und einer Unterliste, welche die Werte der Mücken und Larven enthält. Das mit dem Parameter übergebene Δ der Mücken(-larven)anzahl wird auf den aktuell gespeicherten Zustand addiert (s. Z. 9 bzw. 11). Wie im Entwurf 5.2.1 definiert werden abwechselnd zwei verschiedene Matrizen (`_matrixA` / `_matrixB`) beschrieben (s. Z. 8 - 11). Das Attribut `__useA` in Z. 7 speichert, welche der beiden Matrizen im aktuellen Simulationsschritt beschrieben wird.

Es sei angemerkt, dass der `StackCA` die Funktion `updateValuesOfCell` überschreibt, um während der Simulation den Simulationsstapel (ohne Zellduplikate) für den nächsten Simulationsschritt aufzubauen.

Listing 6.1: Gekürzte Version der Klasse `AbstractMatrix`.

```
1  class AbstractMatrix(object):
2      def _updateValuesOfCells(self, cells):
3          for cell in cells:
4              self._updateValuesOfCell(cell)
5
6      def _updateValuesOfCell(self, cell):
7          for i in range(len(cell[2])):
8              if self.__useA:
9                  self._matrixA[cell[0]][cell[1]][i] += cell[2][i]
10             else:
11                 self._matrixB[cell[0]][cell[1]][i] += cell[2][i]
```

6.4. MALCAM Implementierung

Die Gleichungen 3.3, 3.4, 3.5, 3.6 und 3.7 des MACAM-Modells von Linard et al. [68] werden hier noch einmal in parametrisierter Form inklusive der Funktionsnamen der Implementierung aufgeführt. Das Aufteilen des Modells auf meh-

Quellcode
dir: /mosquito-simulation/src
file: malcam.py

rere Funktionen bietet einige Vorteile: Zum Einen wird die Erweiterbarkeit vereinfacht, soll bspw. die Berechnung der Anzahl an weiblichen Mücken verändert werden, so muss in einer abgeleiteten Klasse nur die betreffende Funktion überschrieben werden. Ferner kann die Übersichtlichkeit der Klasse mit vielen kleinen Funktionen erhöht und Redundanz vermieden werden. Des Weiteren können kleine Funktionen leichter (auch automatisiert) getestet werden.

Berechnung des Gonotrophischen Zyklus (`__calculateGonotrophicCycle`):

$$u = \left(\frac{\alpha_1}{\vartheta - \alpha_2} \right) + 1. \tag{6.1}$$

Berechnung der Entwicklungsrate der Larven (`__developmentRateLarvae`):

$$d = \alpha_6 \cdot \left(\exp\left[\alpha_5 \cdot \{\vartheta - \alpha_4\}\right] - \exp\left[\underbrace{\alpha_5 \cdot \{\alpha_3 - \alpha_4\}}_{\texttt{\_\_factor756}} - \left\{ \frac{\alpha_3 - \vartheta}{\alpha_7} \right\} \right] \right). \tag{6.2}$$

Aufgrund der Tatsache, dass α_3 bis α_5 konstante Werte aufweisen, können während der Simulation Berechnungen durch Einführung des konstanten Attributs `__factor756` eingespart werden.

In den nachfolgenden Gleichungen wird an mehreren Stellen $\#t_s \cdot P = 1$ Tag berechnet. Um in dem Programm einige Rechnungen einzusparen, wird der konstante Faktor `__scaledLengthTimeStep` im Quellcode eingeführt. Um die Lesbarkeit der nachfolgenden Gleichungen zu gewährleisten wird dieser Faktor mit $\mathfrak{A}$ abgekürzt.
Berechnung der Anzahl an Larven (`__calculateLarvae`):

$$l_x^t = l_x^{t-1} + \left(\underbrace{\left[a_x^{t-1} \cdot \left\{ \frac{\mathfrak{A}}{u} \right\} \cdot r \right]}_{\texttt{\_\_hatchedLarvae}} - \underbrace{\left[l_x^{t-1} \cdot \mathfrak{A} \cdot d \right]}_{\texttt{\_\_larvaeAdults}} \right) \cdot \underbrace{\left(1 - \frac{l_x^{t-1}}{l_{max}} \right)}_{\texttt{\_\_larvaeLimiter}}. \tag{6.3}$$

Die Funktion `__hatchedLarvae` berechnet die Anzahl an geschlüpften Larven. Die Anzahl der erwachsen gewordenen Larven wird mit der Funktion `__larvaeAdults` berechnet. Die `__larvaeLimiter` Funktion sorgt für ein logistisches Wachstum und limitiert die Anzahl an Larven pro Zelle.

Berechnung der Anzahl an weibliche Mücken (`__calculateAdults`):

$$a_x^t = a_x^{t-1} + \underbrace{\left(l_x^{t-1} \cdot \mathfrak{A} \cdot d\right)}_{\texttt{\_\_larvaeAdults}} - \underbrace{\left(a_x^{t-1} \cdot \mathfrak{A} \cdot m\right)}_{\texttt{\_\_adultsM}}. \tag{6.4}$$

Berechnung der Anzahl an weiblicher Mücken (`__calculateAdultsMove`), welche die Zelle verlassen:

$$a_M = a \cdot \frac{\mathfrak{A}}{u}. \tag{6.5}$$

Variablen

u gonotrophischer Zyklus

ϑ Temperatur, $[\vartheta] = °\text{C}$

l_x^t Anzahl an weiblichen Mückenlarven in der Zelle x zum Zeitpunkt t

l_{max} maximale Anzahl an weiblichen Mückenlarven in einer Zelle

a_x^t Anzahl an erwachsenen weiblichen Mücken in Zelle x zum Zeitpunkt t

P Länge eines Zeitschritts, hier: $P = \frac{1}{3}$

Wie schon oben erwähnt korrelliert P mit $\#t_s$, sodass stets $\#t_s \cdot P = 1$ Tag gilt.

d Entwicklungsrate der Larven

r Reproduktionsrate der Mücken

m Sterberate der erwachsenen Mücken

a_M Anzahl der erwachsenen Mücken, die nach neuer Blutnahrung suchen und dafür die Zelle x verlassen

a Anzahl an Mücken in einer Zelle x

Parameter (Werte entnommen aus [68])

$\#t_s = 3$ kann als die Anzahl an Zeitschritten pro Tag interpretiert werden, sodass $\#t_s \cdot P = 1$ Tag.

$\alpha_1 = 36.5$ Skalierung des gonotrophischen Zyklus, $[\alpha_1] = °C$

$\alpha_2 = 9.9$ Minimaltemperatur für Suche nach Blutnahrung, $[\alpha_2] = °C$

$\alpha_3 = 35$, $[\alpha_3] = °C$

$\alpha_4 = 10$ Minimaltemperatur für Larvenentwicklung, $[\alpha_4] = °C$

$\alpha_5 = 0.162$ Empirische Konstante, $[\alpha_5] = °C^{-1}$

$\alpha_6 = 0.021$ Empirische Konstante

$\alpha_7 = 5.007$ Empirische Konstante, $[\alpha_7] = °C$

Die Implementierung des MALCAM-Modells erfolgt wie oben aufgeführt in mehreren Funktionen. Dadurch lassen sich Teile des Modells einfach ändern. Soll bspw. die Berechnung der Larven verändert werden, so kann eine Klasse von `MALCAM` abgeleitet werden, wobei die Funktion `__calculateLarvae` überschrieben werden muss. Analog ließe sich die von Klich [56] implementierte Dispersion durch überschreiben der Funktion `__calculateNeighborCells` umsetzen. Bei Klich wird eine Brutstättenqualität zwischen 0 und 1 für diese Zwecke genutzt. Dieser Faktor wird in der Implementierung dieses Werkzeugs für jede Zelle über die Dateneingabe an `MALCAM` übergeben (dort jedoch nicht genutzt) und könnte zu diesem Zwecke verwendet werden.

Die oben eingeführten Parameter geben die konstanten numerischen Werte des MALCAM-Modells wieder. Standardmäßig enthalten diese Parameter die selben Werte wie in [68], können in diesem Werkzeug jedoch bei Bedarf verändert werden.

Im Folgenden wird überprüft, ob es für korrekte Eingabewerte zu Speicherüberläufen kommen kann. Laut [68, S. 165] gilt für die Eingabewerte:

Anzahl Larven $[0, 400\,000]$ Linard et al. legen die maximale Anzahl an Mücken pro Zelle auf 400 000 fest. U. U. könnte es später interessant sein, dieses Limit auf Grundlage der Zellgröße sowie der Habitatqulität zu bestimmen.

Anzahl Mücken $[0, ??]$

Temperatur $[14\,°C, 28\,°C]$

Reproduktionsrate 3.25

Mortalitätsrate 0.21

Eine Betrachtung von Gleichung (6.1) zeigt, dass der gonotrophische Zyklus im Intervall $[3, 10]$ liegt, solange die Temperatur im definierten Intervall ist (Abbildung C.1 im Anhang C bestätigt diese Überlegung[1]). Der Nenner $\vartheta - \alpha_2 \to \vartheta - 9.9$ zeigt eine Singularität bei der Temperatur $\vartheta = 9.9\,°\mathrm{C}$ an (vgl. Abbildung C.2). Für den Grenzwert gilt $\lim_{\vartheta \searrow 9.9} u = \infty$. Das kann im Programm zu einem Speicherüberlauf führen. Der gonotrophische Zyklus nimmt für $\vartheta = 10\,°\mathrm{C}$ einen Wert von 366 Tagen an. Das bedeutet, dass zwischen zwei Blutmahlzeiten ein Jahr Zeit liegt. Ferner wird der gonotrophische Zyklus für Temperaturen kleiner als 9.9 °C negativ. Daher ist anzunehmen, dass für Temperaturen kleiner als 10 °C keine Entwicklung der Mücken zu beobachten ist. Daher werden alle Temperaturen kleiner als 10 °C auf 10 °C gesetzt.

Für die Entwicklungsrate der Larven (Gleichung (6.2)) ist festzustellen, dass beide Exponenten für alle definierten Temperaturen positive Werte ergeben. Da der zweite Exponent stets kleiner als der erste Exponent ist, ist die Entwicklungsrate stets positiv. Der Vorfaktor $\alpha_6 = 0.021$ sorgt dafür, dass die Entwicklungsrate kleiner als 0.1 ist. Abbildung C.3 bestätigt diese Analyse. Das Definitionsintervall für die Temperatur eingehalten werden, da gemäß Abbildung C.4 für die Entwicklungsrate der Larven mit $\lim_{\theta \to \infty} d \to -\infty$ gilt.

Gleichung (6.3) für die Anzahl an Mückenlarven entspricht einem logistischem Wachstum mit maximaler Larvenzahl 400 000 [120, S. 35]. Das stellt sicher, dass es während der Simulation pro Zelle niemals mehr als 400 000 Larven geben kann (vgl. Abbildungen C.5 und C.6), solange der Anfangszustand der Larven pro Zelle kleiner als 400 000 ist. Gemäß Abbildung C.7 ist festzustellen, dass für unzulässig kleine Temperaturen unterhalb von 9.9 °C negative Larvenanzahlen berechnet werden können. Das Programm fängt negative Mücken- sowie Larvenanzahlen ab und setzt die entsprechenden Werte dann auf 0.

Die Anzahl an weiblichen Mücken in Gleichung (6.4) kann nicht negativ werden, solange die Mortalitätsrate kleiner als 1 ist. Diese Bedingung wird im MALCAM-Modell zwar nicht gefordert, ist jedoch plausibel, da bei einer Mortalitätsrate größer 1 mehr als 100 % aller weiblichen Mücken sterben müssten, was offensichtlich ein Widerspruch ist. Aufgrund der vorherigen Analyse ist die Entwicklungsrate der Mückenlarve stets kleiner als 0.1. Da auch die Anzahl der weiblichen Mückenlarven

[1] Abbildungen auf die in diesem Abschnitt verwiesen werden, dienen als Unterstützung der jeweiligen Aussagen und befinden sich daher im Anhang C.

pro Zelle beschränkt ist (default 400 000), ist auch die Anzahl an weiblichen Mückenlarven die pro Zeitschritt erwachsen wird endlich (default max. 40 000). Unter der Annahme, dass pro Zeitschritt Mücken sterben, strebt jede Simulation (unter Vernachlässigung der Dispersion) für unendlich viele Simulationsschritte gegen einen maximalen Wert. Die Abbildungen C.8 und C.9 bestätigen diese Analyse. Für sehr kleine Mortalitätsraten könnten diese maximalen Mückenanzahlen jedoch so groß sein, dass es zu einem Speicherüberlauf kommen könnte. Selbiges gilt für den (unrealistischen) Fall, dass die Mortalitätsrate Werte kleiner gleich 0 annimmt.

Die Anzahl an weiblichen Mücken, welche eine Zelle verlassen, ist kleiner als die Anzahl an vorhandenen Mücken, solange der gonotrophische Zyklus größer als 1 ist (gemäß der vorherigen Analysen ist das für alle definierten Temperaturen der Fall).

Die vorherige Analyse des MALCAM-Modells zeigt, dass unrealistische Werte wie bspw. negative Larvenanzahlen für bestimmte Konfigurationen berechnet werden können. Daher erfolgt die Umsetzung im Werkzeug so, dass unrealistische Ergebnisse und Speicherüberläufe vermieden werden. Daher werden negative Mücken- und Larvenanzahlen auf 0 gesetzt. Temperaturen kleiner als 10 °C werden auf 10 °C gesetzt.

6.5. Datenbankanbindung

Grundlegende theoretische Überlegungen zu Rasterdaten in räumlichen Datenbanken sowie Räumlichen Indexen werden hier nicht diskutiert, sondern in Kapitel 2 behandelt. Um die in der Klasse `DBData` implementierte Anbindung an eine PostgreSQL / PostGIS-Datenbank nutzen zu können, müssen auf dem verwendeten Rechner einige Voraussetzungen erfüllt sein. Im Folgenden wird von einer lokal installierten Datenbank ausgegangen, die Betrachtungen gelten analog auch für eine auf einem externen Server installierte Datenbank.

Die mittels `dbname` übergebene Datenbank (z. B. `test` auf der virtuellen Maschine) muss vor der Nutzung der Software angelegt werden. Des Weiteren muss die PostgreSQL Erweiterung PostGIS geladen werden (bspw. mittels SQL: `CREATE EXTENSION postgis;`). Ferner muss `DBData` während der Initialisierung der Name eines Datenbankennutzers übergeben werden. Es muss sichergestellt wer-

Quellcode
dir: /mosquito-simulation/src
file: dbData.py

den, dass dieser Nutzer die notwendigen Schreib- und Leserechte auf der Datenbank hat.

6.5.1. Durchführen eines Datenbankzugriffes

Um Datenbankzugriffe durchführen zu können, wird die Datenbankanbindung `psycopg2`[2] [119] verwendet. Alle Datenbankzugriffe aus dem Werkzeug erfolgen nach folgendem Schema:

Connect Öffnet eine Verbindung zur Datenbank. In Listing 6.2 Z. 2 werden der Name der Datenbank, der Nutzername, das Passwort sowie der Hostname übergeben. Im Falle einer lokalen Datenbank muss für den Hostnamen `localhost` angegeben werden.

Anschließend muss der Cursor geholt werden (s. Z. 4).

Do something usefull SQL-Anfragen können mittels `cur.execute(...)` ausgeführt werden (s. Z. 7).

Commit gibt die im vorherigen Schritt durchgeführten Änderungen frei (s. Z. 10). Dadurch kann das Datenbankmanagementsystem die Änderungen in die Datenbank einpflegen. Dieser Schritt kann bei rein lesenden Zugriffen übersprungen werden, da keine Änderungen in die Datenbank eingepflegt werden müssen.

Analyze Verbose sammelt Statistiken über den Tabelleninhalt. Diese Informationen werden bei späteren Anfragen an die Datenbank genutzt, um zu entscheiden, wie die Anfrage verarbeitet wird [76, S. 381] (s. Z. 13).

Close Connection Schließt die Verbindung (s. Z. 16).

Listing 6.2: Exemplarische Durchführung eines Datenbankzugriffes.

```
1 # open connection
2 conn = db.connect("""dbname=%s user=%s host=%s password=%s""" % (
      dbname, user, host, password))
3 # fetch the cursor
4 cur = conn.cursor()
5
```

[2] `Psycopg2` wird in dieser Arbeit als alias `db` importiert: `import psycopg2 as db`.

```
6  # do something usefull
7  cur.execute("""-- do something usefull""")
8
9  # commit the changes (skip this step, if you just read data from the db)
10 conn.commit()
11
12 # analyse the table content (skip this step, if you just read data from the
       db)
13 cur.execute("""ANALYZE VERBOSE %s;""" % (tablename))
14
15 # close the connection
16 conn.close()
```

In den folgenden Abschnitten werden ausschließlich die SQL-Codes aufgeführt, welche den Kommentar „do something usefull“ aus Z. 7 Listing 6.2 sinnvoll ersetzen.

6.5.2. Erstellen einer Tabelle

Für die Erstellung einer Tabelle sind zwei unterschiedliche Möglichkeiten vorgesehen. Bei der ersten Option wird die zu nutzende Tabelle von dem Werkzeug während der Initialisierung erstellt. Dazu wird der Parameter `newTable = True` der `__init__`-Funktion übergeben. Die zu erzeugende Tabelle darf vor Initialisierung des Werkzeugs nicht in der angegebenen Datenbank existieren. Die zweite Option geht von einer schon existierenden Tabelle aus, dafür muss der `__init__`-Funktion der Parameter `newTable = False` übergeben werden.

Das Erstellen einer neuen Tabelle in PostgreSQL / PostGIS wird am Listing 6.3 erläutert[3]. In Z. 2 f. wird eine neue Tabelle Namens „myraster“ angelegt. Der Primärschlüssel der Tabelle ist der Simulationstag vom Typ `smallint`. Das räumliche Raster (`rast`, Typ `raster`) darf dabei nicht `NULL` sein. Wie in Abschnitt 5.3 werden ein Index über die räumliche Spalte (s. Z. 7) und ein Index über die Tagesspalte (s. Z. 9) definiert [98, S. 68], [112, S. 1475 ff.] und [82, S. 182].

Listing 6.3: Erstellung einer neuen Tabelle `myraster` in PostgreSQL / PostGIS inklusive Indexe auf den Raster- und Tagesspalten [98, S. 68], [112, S. 1475 ff.] und [82, S. 182].

[3] Die SQL-Codes werden zur Verbesserten Lesbarkeit so aufgeführt, wie diese ein Nutzer über eine Konsole eingeben würde. In der Implementierung dieses Werkzeugs werden die konkreten Werte über Parameter eingegeben (s. Quellcode von `DBData`).

```
1  -- Create a table with columns day and rast.
2  CREATE TABLE myraster(day smallint,
3                        rast raster NOT NULL,
4                        PRIMARY KEY(day));
5
6  -- Create a new index on the raster column.
7  CREATE INDEX myraster_rast_st_convexhull_idx ON myraster
       USING gist(ST_ConvexHull(rast));
8
9  -- Create a new index on the day column.
10 CREATE UNIQUE INDEX myraster_day_idx ON myraster (day);
```

6.5.3. Speichern eines Simulationstages

Für die Speicherung eines Simulationstages wird, sofern nicht vorhanden, ein neuer Eintrag in der Datenbank wie in Listing 6.4 angelegt [98, S. 420 ff. & 426], andernfalls wird der erste Schritt übersprungen. Dabei bezeichnet `tableName` (Z. 1) die zu nutzende Tabelle und `day` (Z. 2) den neu anzulegenden Simulationstag. Die Semantik der übrigen Parameter (`width`, ..., `srid`) entspricht der aus Tabelle 6.1 und wird hier nicht erneut aufgeführt. Dabei wird ein leeres `width` × `height` Raster (Z. 4) mit zwei Bändern angelegt (Z. 3, 7 ff.). Die beiden Bänder werden mit dem Wert `0` initialisiert und repräsentieren die Anzahl an weiblichen Mücken bzw. die Anzahl an weiblichen Mückenlarven. Soll das Werkzeug derart erweitert werden, dass der Zustand einer Zelle bspw. auch die Anzahl an männlichen Mücken enthält, so muss das `ARRAY` in Z. 7 ff. um einen weiteren Eintrag erweitert werden.

Listing 6.4: Anlegen eines neuen Simulationstages [98, S. 420 ff. & 426].

```
1  INSERT INTO tableName
2    VALUES (day,
3      ST_AddBand(
4        ST_MakeEmptyRaster(
5          width,height,upperleftx,upperlefty,scalex,scaley,skewx,skewy,
               srid
6        ),
7        ARRAY[
```

```
8         ROW(1, '32BUI', 0, NULL),
9         ROW(2, '32BUI', 0, NULL)
10      ]::addbandarg[]
11    )
12  );
```

Die zu speichernden Werte werden mit der `UPDATE`-Funktion des in Listing 6.5 aufgeführtem SQL-Code für jede Zelle in die Tabelle geladen [98, S. 464]. Der `ST_SetValue`-Funktion (Z. 2) wird der Name der zu verwendenden Spalte (`rast`), das zu beschreibende Band (`band`), die virtuellen Koordinaten (`x_tilde` und `y_tilde`) sowie der zu updatende Wert (`value`) übergeben. Die `WHERE` Klauses (Z. 3) spezifiziert den zu aktualisierenden Tag. Die in Listing 6.6 aufgeführte Funktionalität wird für jede Zelle des Zellularen Automaten zweimal aufgerufen.

Listing 6.5: Laden eines Wertes einer Zelle in das korrespondierende Band der Datenbank [98, S. 464].

```
1 UPDATE tableName
2   SET rast = ST_SetValue(rast, band, x_tilde, y_tilde, value)
3   WHERE tableName.day = day;
```

6.5.4. Laden eines Simulationstages

Ein kompletter Simulationstag kann aus der Datenbank geladen werden, indem unter Kenntnis der Größe des Zellularen Automaten jede Zelle mittels eines `SELECT FROM WHERE`-Statements geladen wird (s. Listing 6.6) [98, S. 464]. Für die zu ladenden Einträge werden aliase (`adults` und `larvae`) verwendet (s. Z. 1). Um die so geladenen Ergebnisse in Python weiter zu verarbeiten, wird die Datenbankantwort analog zu Listing 6.7 verarbeitet. In Z. 1 wird der Cursor geholt, wobei in `answer` die Anfrageergebnisse im Stringformat als Tupel gespeichert sind. Mittels Z. 2 werden die Daten in eine für diese Arbeit besser weiterverarbeitbare Listenform konvertiert. Die aufgeführten Operationen werden für alle Zellen wiederholt.

Die in dieser Arbeit verwendeten Indexe werden vom Query-Optimizer verwendet, wenn `ANALYZE VERBOSE` nach dem Speichern eines Simulationstages durchgeführt worden ist. Das durch den Dienst autovacuum ausgeführte `ANALYZE` [82, S. 182] hat sich als nicht zielführend herausgestellt, da der Query-Optimizer dann nur einen der definierten Indexe verwendet hat.

Listing 6.6: Laden eines Wertes einer Zelle in das korrespondierende Band der Datenbank [98, S. 464].

```
SELECT ST_Value(rast, 1,x_tilde, y_tilde) AS adults, ST_Value(
   rast, 2, x_tilde, y_tilde) AS larvae
  FROM tableName
  WHERE tableName.day = day
```

Listing 6.7: Verarbeitung der Ergebnisse aus Listing 6.6.

```
answer = cur.fetchall()
result = [int(answer[0][0]), int(answer[0][1])]
```

Kapitel 7

Tests

In diesem Kapitel werden die durchgeführten Tests besprochen. Im ersten Abschnitt werden die Unittests der einzelnen Komponenten exemplarisch dargestellt. Der zweite Abschnitt beschäftigt sich mit einem automatisierten Integrationstest (welcher streng genommen ein Unittest ist). Im dritten Abschnitt wird aufgezeigt, wie mit Hilfe von Python Skripten das Werkzeug gestestet worden ist. Der vierte Abschnitt zeigt auf, dass die in der PostgreSQL / PostGIS-Datenbank gespeicherten Rasterdaten in QGIS graphisch dargestellt werden können. Im letzten Abschnitt wird ein Simulationstest mit einem großen Zellularen Automaten über einen längeren Zeitraum durchgeführt, um zu demonstrieren, dass dieses Werkzeug unter realistischen Bedingungen funktioniert.
In dieser Arbeit wird für das automatisierte Testen Python Unittest verwendet. Die einzelnen Unittests werden in einer Testsuite zusammengeführt, um alle verfügbaren automatischen Tests „auf einmal“ ausführen zu können. Der automatisierte Integrationstest im zweiten Abschnitt wird ebenfalls von der Testsuite aufgerufen.

7.1. Unittests

Quellcode
dir: /mosquito-simulation/test
file: testSimpleData.py

Mit Hilfe von Unittests werden die wichtigsten Funktionalitäten der unterschiedlichen Module getestet [89, The Python Library Reference S. 1269 ff.]. Das Vorgehen wird am Beispiel von `TestSimpleData` in Listing 7.1 demonstriert. Die Klasse `TestSimpleData` erbt von `unittest.TestCase`. Die Vorbereitung des Tests wird mit Hilfe der Funktion `setUp` (Z. 7 ff.) durchgeführt. In Z. 9 wird der `TestTransformer` initialisiert. Dieser wird benötigt, um `SimpleData` zu testen. Eigentlich sollten bei Unittests die zu testenden Module unabhängig von anderen Modulen getestet werden. Problematisch ist jedoch, dass `SimpleData` nicht unabhängig von der Koordinatentransformerfunktionalität getestet werden kann. Alternativ könnte das Verhalten von `TestTransformer` mittels Mock-Objekten oder Hilfsklassen simuliert werden. Es ist dabei jedoch zu bedenken, dass auch in Testfällen Fehler auftreten können und sich der Implementierungsaufwand unverhältnismäßig erhöht. Aufgrund dessen wird hier der `TestTransformer` verwendet und davon ausgegangen, dass dieser nach passieren des zugehörigen Unittests aufgrund der trivialen Funktionalität auch hier korrekt funktioniert. Um während des Tests CSV-Dateien lesen und schreiben zu können muss auf dem verwendeten System ein entsprechendes Verzeichnis angegeben werden. Auf der beigefügten virtuellen Maschine wird `/home/bodhi/workspace/mosquito-simulation/test` genutzt. Dieses Verzeichnis wird in der Variable `DIRECTORY` im Modul `setupTests` gespeichert und muss ggf. auf anderen Systemen angepasst werden. Die Funktion `getComparator5x4` (aus dem `helper` Modul) in Z. 14 liefert einen vordefinierten Zellularen Automaten, der zu Testzwecken genutzt wird. Das Modul `helper` sammelt Funktionalitäten, welche für verschiedene Unittests benötigt werden.

Die eigentlichen Funktionalitätstests erfolgen in Funktionen, deren Name mit `test` beginnen muss, damit `unittest` diese Funktionen automatisch ausführt. Der Test `testSaveStep` speichert den Zellularen Automaten mit Hilfe der `saveStep`-Funktion von `SimpleData`. Anschließend lädt die Hilfsfunktion `loadCA` (Modul `helper`) den Zellularen Automaten aus der gespeicherten Datei und prüft die Korrektheit mit Funktion `listsEqualTest` (Modul `helper`). Die `self.assertTrue` Funktion wird von Unittest geerbt und wird hier genutzt um den Rückgabeparameter der `listsEqualTest` Funktion auf `True` zu testen. Sollte der Rückgabeparameter `False` sein, hat der Test versagt und es wird eine Fehlermeldung ausgegeben. Der Test `testLoad-`

`Step` funktioniert analog.

Zum Abschluss der Tests wird die geschriebene CSV-Datei gelöscht (Z. 16 f.).

Listing 7.1: Gekürzte Version von TestSimpleData.

```
1  from os import remove
2  import unittest
3  from helper import listsEqualTest, getComparator5x4, loadCA, saveCA
4  from simpleData import SimpleData
5  from testTransformer import TestTransformer
6  from setupTests import DIRECTORY
7  class TestSimpleData(unittest.TestCase):
8      def setUp(self):
9          self.__name = "testOut"
10         self.__coordinateTransformer = TestTransformer()
11         self.__directory = DIRECTORY
12         self.__simpleData = SimpleData(self.__name,
13                                        self.__coordinateTransformer,
14                                        self.__directory)
15         self.__ca = getComparator5x4()
16
17     def tearDown(self): # Finish the test: Remove the testOut.csv file .
18         remove("%s/%s.csv" % (self.__directory, self.__name)) #
               remove the CSV file
19
20     def testSaveStep(self): # Test the saveStep function with self.__ca.
21         self.__simpleData.saveStep(self.__ca, 0)
22         equal = listsEqualTest(self.__ca, loadCA(self.__directory,
               self.__name, 0, 5, 4))
23         self.assertTrue(equal, "testSaveStep failed")
24
25     def testLoadStep(self): # Test the loadStep function with self.__ca.
26         saveCA(self.__ca, 0, self.__directory, self.__name) # save the
               CA
27         equal = listsEqualTest(self.__simpleData.loadStep(0, 5, 4),
               self.__ca)
28         self.assertTrue(equal, "testLoadStep failed")
```

Es sei angemerkt, dass für den Test von `DBData` eine PostgreSQL / PostGIS Datenbank vorhanden sein muss. Die beigefügte Virtuelle Maschine enthält eine Datenbank `test`, welche für diese Testzwecke genutzt werden kann. Während des Tests legt der Datenbankennutzer `postgres` die Tabelle `testraster` an.

7.2. Integrationstest

Mit Hilfe der oben aufgeführten Unittests können Module wie bspw. `SimpleData` oder `MALCAM` gut auf ihre eigenständige Funktionsweise geprüft werden. Fehler, welche beim Zusammenspiel verschiedener Komponenten auftreten, lassen sich mit diesen Tests nicht finden [107, S. 219]. Daher wird das oben vorgestellte Unittestkonzept in `TestIntegration` „missbraucht“, um einen vollständigen Test des entworfenen Systems automatisiert durchzuführen. Für die Dateneingabe wird `SimpleInput` genutzt, die Speicherung erfolgt mittels `SimpleData` und als Regelsatz wird `TestRule` verwendet. Es werden vier verschiedene Konfigurationen getestet, wobei jeweils geprüft wird, ob die Initialisierung sowie der erste Simulationsschritt korrekt verarbeitet worden sind:

`testEasyCAWithoutDispersion` führt den Test mit dem einfachen Verfahren ohne Dispersion durch.

`testEasyCAWithDispersion` führt den Test mit dem einfachen Verfahren mit Dispersion durch.

`testStackCAWithoutDispersion` führt den Test mit dem ohne Dispersion durch.

`testStackCAWithDispersion` führt den Test mit dem Stapelverfahren mit Dispersion durch.

Der Integrationstest wird auch in der Testsuite aufgerufen. Dadurch ist sichergestellt, dass zusätzlich zu den Unittests einmal das komplette System automatisiert im Zusammenspiel getestet wird.

7.3. Skripttests

Die bisher aufgeführten automatisierten Tests eignen sich sehr gut, um einzelne Komponenten des Werkzeugs automatisch zu testen. Um das gesamte Werkzeug

im Zusammenspiel zu testen sind Unittests nur in begrenztem Ausmaße geeignet. Insbesondere eine „schnelle“ Prüfung während der Entwicklung lässt sich besser durchführen, in dem eine Simulation mit leicht nachvollziehbaren Daten durchgeführt und anschließend visualisiert wird. Zu diesem Zweck könnte der `SimulationController` in einer Konsolensitzung importiert und mit entsprechenden Daten ausgeführt werden. Problematisch dabei ist jedoch, dass derartige Versuche schlecht reproduzierbar und dokumentierbar sind. Daher sind in dieser Arbeit Pythonskripte entwickelt worden, welche das Werkzeug ausführen. Die Simulationsergebnisse werden dabei (zumeist) in CSV-Dateien gespeichert, aus denen mit einem GNU Octave Programm Grafiken generiert worden sind.

Das Skript `testRun.py`, welches sich (wie alle weiteren in diesem Abschnitt aufgeführten Pythonskripte) auf der beigefügten DVD im Verzeichnis `/mosquito-simulation/run/` befindet, diente während der Entwicklung als ein erster Test. Dabei steht neben dem Zusammenspiel der einzelnen Komponenten im Fokus, ob `EasyCA` und `StackCA` wie gewünscht die verschiedenen Zellen neu berechnen. Um die Ergebnisse besser nachvollziehen zu können, wird in diesem Skript ausschließlich die `TestRule` verwendet. Es sind folgende Versuche durchgeführt worden:

Versuch 1 geht von einem 5×5 Zellularem Automaten aus, bei dem sich in jeder Zelle drei erwachsene Mücken sowie vier Larven befinden. Der Test wird mit `EasyCA` und abgeschalteter Dispersion für einen Simulationsschritt durchgeführt. Die gespeicherte CSV-Datei muss nach dem Versuch den Ausgangszustand des Zellularen Automaten sowie das Ergebnis nach dem ersten Simulationsschritt enthalten. Soweit nicht anders angegeben, müssen bei den folgenden Versuchen alle Simulationstage abgespeichert werden. In dieser Konfiguration muss jede Zelle nach dem Schritt 1003 Mücken und 104 Larven enthalten.

Versuch 2 basiert auf einem leeren 5×5 Zellularem Automaten, wobei der `EasyCA` mit eingeschalteter Dispersion für einen Simulationstag genutzt wird. Die vier Eckzellen müssen dabei drei Mücken und Larven enthalten. Alle übrigen Randzellen enthalten jeweils fünf Mücken und Larven. Alle inneren Zellen sind jeweils mit acht Mücken und Larven besetzt. Diese Ergebnisse werden in Anhang C.2 in Abbildung C.10 visualisiert.

Versuch 3 nutzt den selben initialen Zellularen Automaten wie Versuch 1, führt den Simulationsschritt jedoch mit dem `StackCA` und ausgeschalteter Dispersion durch. Das Ergebnis muss mit dem von Versuch 1 übereinstimmen.

Versuche 4 & 5 basieren auf einem 10×10 Zellularem Automaten, bei dem die Zellen mit den Indexen $(0, 6)$, $(4, 4)$ und $(9, 0)$ mit fünf Mücken und Larven belegt sind. Die übrigen Zellen sind leer. Beide Versuche führen jeweils zehn Simulationsschritte mit dem `StackCA` bei eingeschalteter Dispersion durch. Der Unterschied liegt darin, welche Schritte gespeichert werden. Bei Versuch 4 werden alle Schritte gespeichert. Für Versuch 5 hingegen wird der übergebene `saveStep` Parameter der `doSimulation` Funktion des `SimulationControllers` auf vier gesetzt. Daher werden der Initialzustand (Abbildung C.11(a)), die Simulationstage eins (Abbildung C.11(b)), fünf (Abbildung C.11(c)), neun (Abbildung C.11(d)) und zehn (da es der letzte Simulationstag ist, Abbildung C.11(e)) gespeichert. Die Ergebnisse dieser Simulationstage werden in Abbildung C.11 aufgeführt.

Die Datenbankanbindung wird mit dem Skript `testDBRun.py` getestet. Dabei wird Versuch 4 mit einem Versuchstag durchgeführt und in einer Testdatenbank gespeichert. Auf der beigefügten virtuellen Maschine steht für diese Zwecke die Datenbank `test` zur Verfügung. Der Test setzt eine neue Tabelle (`myraster`) auf, welche vor jedem Lauf des Skriptes gelöscht werden muss. Dazu wird das Skript `dbtest.py` ausgeführt, welches die in der Datenbank gespeicherten Daten ausliest und im CSV Format analog zu den vorherigen Versuchen abspeichert. Ferner löscht `dbtest.py` die Tabelle `myraster` aus der Datenbank.

Ein weiterer durchgeführter Skripttest `malcamRun.py` testet das MALCAM-Modell mit Hilfe eines 10×10 Zellularen Automaten, bei dem ausschließlich Zelle $(5, 5)$ initial mit jeweils 1000 Mücken und Larven besetzt ist. Es werden insgesamt 100 Tage simuliert. In Abbildung C.12 werden neben dem Initialzustand Visualisierungen für die Zustände nach 50 bzw. 100 Tagen angegeben.

Der letzte durchgeführte Skripttest basiert auf dem vorherigen Test für das MALCAM-Modell, speichert die Simulationsergebnisse jedoch in einer PostGIS-Datenbank, auf der virtuellen Maschine steht unter der `test`-Datenbank die Tabelle `qgis-test` für diesen Versuch zur Verfügung. Die gespeicherten Daten werden im nachfolgenden Abschnitt auf ihre Kompatibilität mit QGIS geprüft.

7.4. QGIS-Datenvisualisierung

Quellcode
dir: /mosquito-simulation/run
file: qgisDBRun.py

In diesem Abschnitt wird gezeigt, wie die im letzten aufgeführten Skripttest gespeicherten Daten in QGIS importiert und visualisiert werden können. Das Ergebnis ist auf der beigefügten virtuellen Maschine unter `/home/bodhi/firstTest.qgs` abgespeichert worden.

Der erste Schritt besteht darin eine Verbindung zur lokalen PostgreSQL / PostGIS Datenbank aufzubauen. Dazu muss der Dialog „Create a New PostGIS connection“ geöffnet werden. Dieser Dialog kann im rot umrandeten Reiter in Abbildung 7.1 per Rechtsklick „New Connection...“ geöffnet werden. Wie in Abbildung 7.1 zu sehen werden die entsprechenden Felder ausgefüllt. Das Passwort für die Datenbank lautet „bodhi“. Benutzername und Passwort werden in QGIS als Texte gespeichert. Bei sicherheitsrelevanten Daten sollte daher von einer Speicherung abgesehen werden. Anschließend kann der „DB Manager“ mittels „Database“ > „DB Manager“ > „DB Manager“ geöffnet werden. Wie in Abbildung 7.2 zu erkennen ist, wird unter anderem unter „PostGIS“ > „PostGIS“ > „qgistest“ aufgeführt. Um die gespeicherten Daten zu visualisieren kann in Abbildung 7.2 im rot umrandeten Bereich auf „Add to canvas“ geklickt werden. Dadurch wird das gespeicherte Raster, wie in Abbildung 7.3, visualisiert.

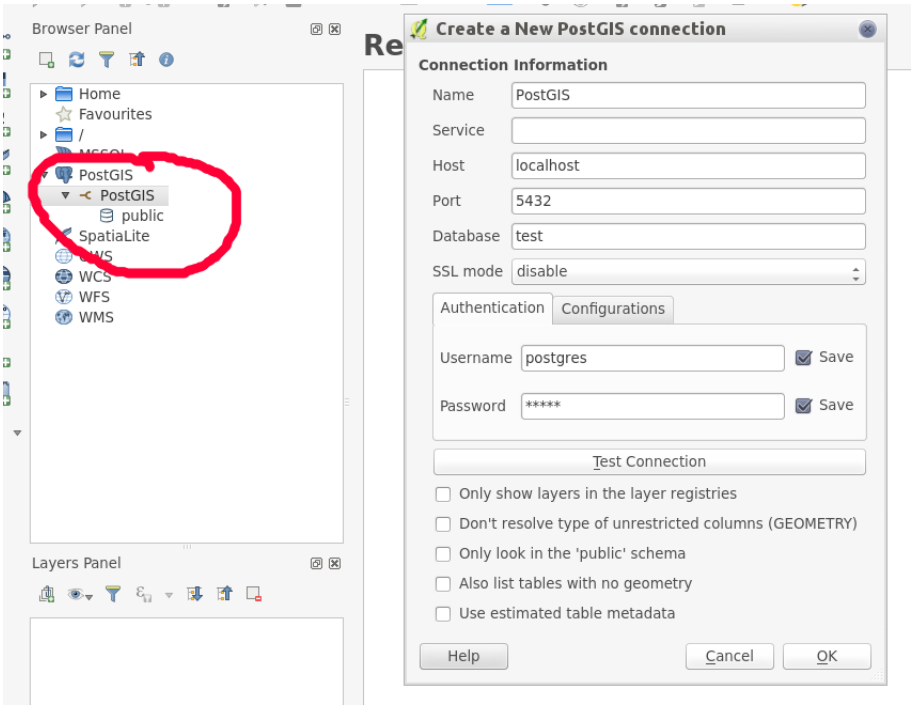


Abbildung 7.1.: Screenshot des Aufbaus einer Datenbankanbindung von QGIS zu PostGIS.

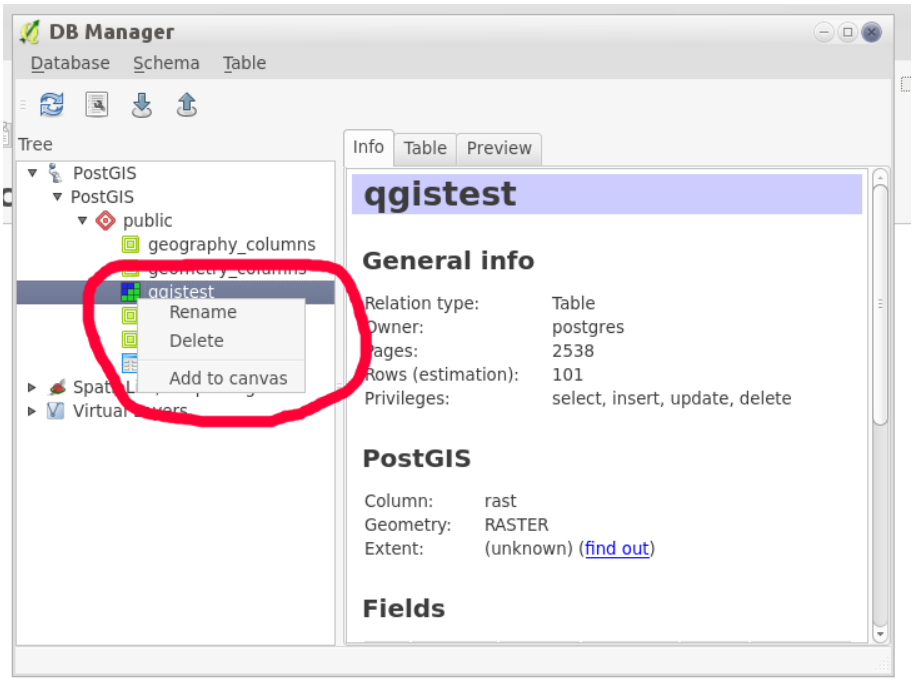


Abbildung 7.2.: Screenshot des Ladens der PostGIS Daten aus der Tabelle `qgistest`.

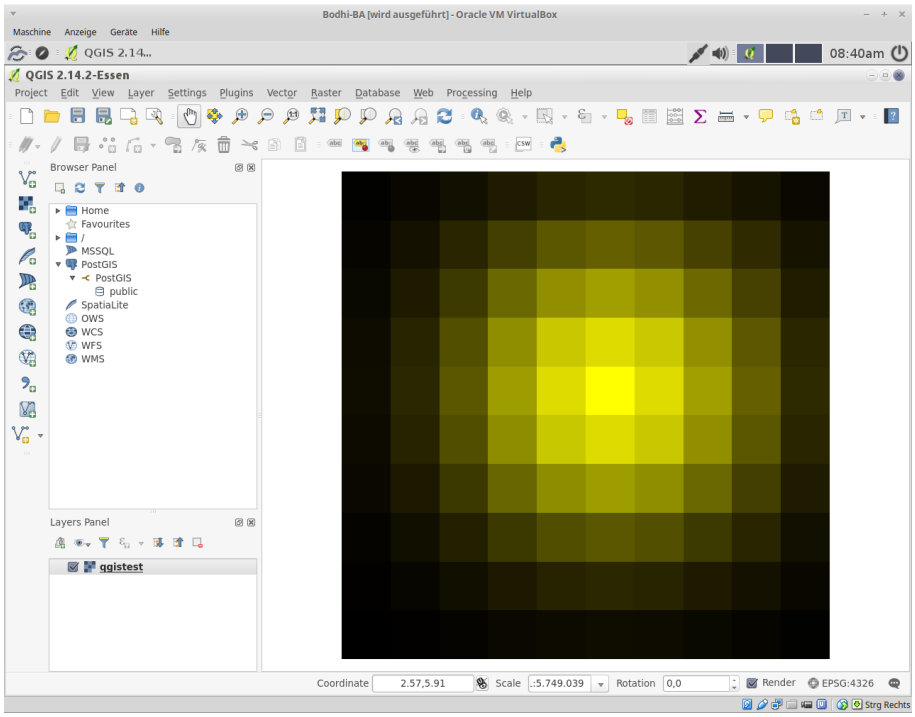


Abbildung 7.3.: Screenshot der Visualisierung des in der PostGIS-Tabelle `qgistest` gespeicherten Rasters.

7.5. Anwendungstest

In diesem Abschnitt wird überprüft, ob das Werkzeug für einen 1000×1000 Zellularen Automaten funktioniert. Der initiale Zellulare Automat ist fast leer, lediglich die Zelle (500, 500) enthält 1000 Mücken(-larven). Es werden mit Hilfe des Skripts `applicationTest.py` 2000 Tage simuliert und die Simulationsergebnisse werden in einer CSV-Datei abgespeichert. Die Ergebnisse einiger Simulationstage werden im Anhang Abbildung C.13 dargestellt. Es ist zu erkennen, dass sich die Mücken konzentrisch um den Infektionsherd ausbreiten. Das liegt daran, dass es keine Vorzugsrichtung gibt, in die sich die Mücken ausbreiten.

Quellcode
dir: /mosquito-simulation/run
file: applicationTest.py

Kapitel 8

Evaluation

In diesem Kapitel wird die Performanz des Werkzeugs untersucht. Für alle folgenden Untersuchungen wird die im Kapitel 6 vorgestellte virtuelle Maschine mit 2 GiB RAM und einem CPU-Kern verwendet. Im ersten Abschnitt wird untersucht, wie viele Zellen maximal unter den gegebenen Bedingungen ausgewertet werden können. Im zweiten Abschnitt werden die beiden Verfahren bzgl. ihrer Performanz in Abhängigkeit vom Füllgrad des Zellularen Automaten betrachtet. Daraus wird die Heuristik abgeleitet, welche über das zu verwendende Verfahren entscheidet. Im dritten Abschnitt werden die Speicher- und Ladegeschwindigkeiten der Datenbank im Vergleich zu CSV-Dateien untersucht. Im letzten Abschnitt wird die Erweiterbarkeit des Werkzeugs untersucht und mit der Software von Klich verglichen.

8.1. Maximale Automaten Größe

Quellcode
dir: /mosquito-simulation/run
file: benchmarkCAEasyCA.py

In diesem Abschnitt wird zum Einen bestimmt, wie lange die Berechnung eines Simulationsschrittes in Abhängigkeit von der Größe des Zellularen Automaten im worst case dauert und zum Anderen, welche Automatengröße maximal auf der verwendeten virtuellen Maschine simuliert werden kann. Die Untersuchung ist unabhängig vom Füllgrad des Zellularen Automaten. Dabei wird der `EasyCA` sowie als Regelsatz das MALCAM-Modell genutzt. Die Simulationsergebnisse werden nicht gespeichert, um die Ergebnisse nicht zu verfälschen. Für diesen Versuch wird das Skript `benchmarkCAEasyCA.py` verwendet. Das Ergebnis ist in Abbildung 8.1 visualisiert worden. Bis zu einer Automatengröße von ca. 10 Millionen Zellen steigt die benötigte Rechenzeit für einen Simulationsschritt näherungsweise linear an. Das liegt daran, dass die Daten des Werkzeugs den Hauptspeicher nicht gänzlich füllen. Das bedeutet, dass der Hauptspeicherzugriff die geschwindigkeitsbestimmende Operation ist. Werden größere Zellulare Automaten simuliert, ist damit zu rechnen, dass die Zeit für einen Zeitschritt deutlich stärker ansteigt, da nicht alle Daten im Hauptspeicher gehalten werden können. Das Betriebssystem lagert dann Hauptspeicherseiten auf den Sekundärspeicher aus und lädt diese bei Bedarf wieder in den Hauptspeicher. Da Sekundärspeicherzugriffe i. A. wesentlich langsamer als Hauptspeicherzugriffe sind [14, S. 175 f.], wird die Rechenzeit verstärkt durch den Auslagerungsprozess von Hauptspeicherseiten bestimmt. In Abbildung 8.1 stimmen die gemessenen Werte für 11 - und 12 Millionen Zellen gut mit dieser theoretischen Betrachtung überein. Der gemessene Wert für 13 Millionen Zellen liegt bei etwa 400 Sekunden, zu erwarten wären jedoch mehr als 470 Sekunden. Der Grund könnte an nicht trivialen Seiteneffenkten liegen, welche sich aus der Hauptspeicherzuteilung seitens des verwendeten Betriebssystems sowie aus der Verwendung des Garbage Collectors des Python Laufzeitsystems ergeben.

Abschließend ist festzuhalten, dass der Rechenaufwand im Falle des `EasyCA's` unterhalb der durch den Hauptspeicher gesetzten Grenze für die Automatengröße linear zur Größe des Zellularen Automaten anwächst.

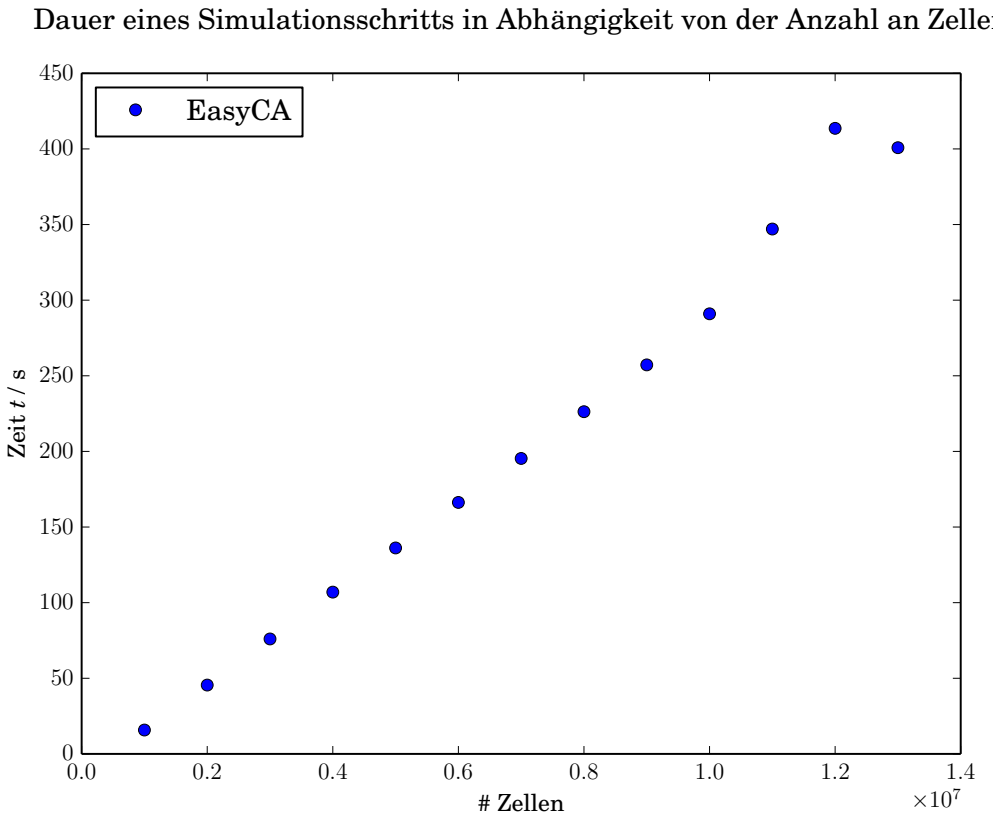


Abbildung 8.1.: Dauer eines Simulationsschritts im worst case in Abhängigkeit von der Anzahl an Zellen.

8.2. Effizienz der beiden verschiedenen Verfahren und Entwicklung der Heuristik

In diesem Abschnitt wird untersucht, wie effizient die beiden implementierten Verfahren sind. Das „Stapelverfahren“ (Alg3) wird mit dem „einfachen Verfahren“ (Alg1) verglichen. Für diese Untersuchung werden Zellulare Automaten mit einer Größe von 700 000 bis 2 000 000 Zellen verwendet. Für jede Größe wird die Rechenzeit für einen Simulationsschritt in Abhängigkeit von dem Füllgrad des Automaten gemessen. Dazu wird das Skript `benchmarkCAFillingDegree.py` verwendet. Exemplarisch wird hier eine Visualisierung der Simulation eines Zellularen Automaten mit 1 000 000 Zellen in Abbildung 8.2 aufgeführt. In der genannten Abbildung ist zu erkennen, dass die Simulationszeit eines Tages für den `EasyCA` bei allen Füllgraden

Quellcode
dir: /mosquito-simulation/run
file: benchmarkCAFillingDegree.py

bei etwa 31 Sekunden liegt. Dieser Befund entspricht der Erwartung, da die Anzahl der neu zu berechnenden Zellen unabhängig vom Füllgrad des Automaten ist. Im Fall des `StackCA's` ist festzustellen, dass die Rechenzeit in Abhängigkeit vom Füllgrad näherungsweise linear zunimmt. Das lässt sich dadurch erklären, dass die Anzahl an neu zu berechnenden Zellen stark abhängig vom Füllgrad des Zellularen Automaten ist. Die leichte Streuung der gemessenen Werte ist darauf zurückzuführen, dass auf einem realen Betriebssystem immer mehrere Prozesse aktiv sind, welche aufgrund des Schedulings zu statistischen Schwankungen der Messwerte führen [38, S. 426 ff.]. Analoges gilt für den Garbage Collector des Python-Laufzeitsystems, welcher während der Programmausführung aktiv wird [69, S. 356]. Abbildung 8.2 ist zu entnehmen, dass bei weniger als etwa 800 000 gefüllten Zellen, das entspricht einem Füllgrad von etwa 80 %, der `StackCA` performanter ist als der `EasyCA`. Bei einem höheren Füllgrad ist der `EasyCA` aufgrund des Overheads für die Stapelverwaltung des `StackCA's` besser geeignet.

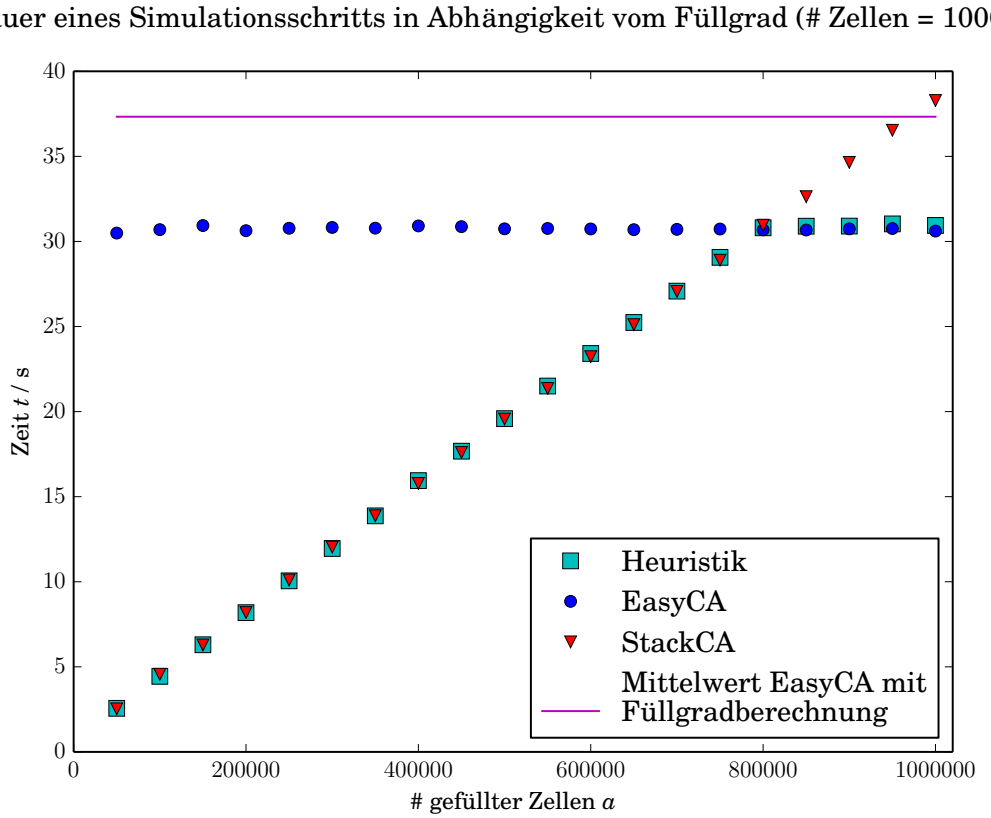


Abbildung 8.2.: Dauer eines Simulationsschrittes des `EasyCA's`, des `StackCA's` und unter Verwendung der Heuristik in Abhängigkeit der Anzahl an gefüllten Zellen. Zum Vergleich wird die mittlere Rechenzeit des `EasyCA's` mit Füllgradberechnung aufgeführt.

Die angesprochene Heuristik wird aus den verschiedenen Versuchen bei unterschiedlichen Automatengrößen abgeleitet. Dabei wird jeweils der Füllgrad bestimmt, oberhalb dem der `EasyCA` dem `StackCA` überlegen ist. Dazu werden für die unterschiedlich großen Zellularen Automaten jeweils 20 Durchläufe für die beiden Verfahren bei unterschiedlichen Füllgraden durchgeführt.

Es wird der prozentuale Füllgrad am Schnittpunkt p_{fill}^{SP} für den `EasyCA` und `StackCA` bestimmten Steigungen und Interzepten berechnet. Die Werte für den prozentualen Füllgrad am Schnittpunkt der Regressionsgeraden liegen im Intervall von 78.83 % bis 80.44 %. Das bedeutet, dass das arithmetische Mittel bei 79.63 % liegt und die Standardabweichung von 0.40 % sehr klein ist. Die Rechenzeit des `StackCA's` steigt um zwei Größenordnungen stärker als die des `EasyCA's`. Das bedeutet, dass die Rechenzeit beim `StackCA` wie zu erwarten war vom Füllgrad abhängig ist. Für den `EasyCA` ist eine solche Abhängigkeit nicht feststellbar. Des Weiteren ist zu er-

kennen, dass der Interzept, d. h. der konstante Anteil der Regressionsanalyse, mit der Größe des Zellularen Automaten ansteigt. Daraus lässt sich schließen, dass die Simulationen zweier verschieden großer Zellularer Automaten bei gleichem Füllgrad unterschiedlich lange dauern. Je größer der Zellulare Automat ist, desto länger dauert die Berechnung eines Simulationstages, da mehr Zellen neu berechnet werden müssen. Diese Beobachtung gilt für beide Verfahren.

Abschließend lässt sich feststellen, dass bis zu einem Füllgrad von 79.63 % das „Stapelverfahren“ dem „einfachen Verfahren“ in dieser Implementierung überlegen ist. Daher wird diese Schranke als Heuristik im Werkzeug verwendet, um festzulegen, welches Verfahren für die Simulation genutzt wird. Eine Überprüfung der Heuristik ist unter gleichen Bedingungen wie der Versuch aus Abbildung 8.2 durchgeführt worden. Wie in der dortigen Abbildung zu erkennen ist, stimmen die Messwerte unterhalb der Schwelle gut mit denen des `StackCA's` und oberhalb der Schwelle gut mit dem des `EasyCA's` überein. Es ist untersucht worden, wie sich das System bzgl. der Performanz verhält, wenn nach jedem Schritt geprüft wird, welches Verfahren verwendet werden soll. Dazu ist ein Zähler implementiert worden, welcher während der Simulation die Anzahl an belegten Zellen misst. Dadurch ist es jedoch zu einer signifikanten Verschlechterung der Performanz gekommen. Zu Vergleichszwecken wird in Abbildung 8.2 die mittlere Zeit des `EasyCA's` als margenta farbene Linie eingezeichnet. Daher wird in diesem Werkzeug alle zehn Simulationsschritte geprüft, ob das Verfahren gewechselt werden muss. Der Füllgrad ergibt sich für den `StackCA` aus der Größe des Simulationsstacks und wird für den `EasyCA` explizit neu berechnet. Das Werkzeug von Klich arbeitet ähnlich wie der `EasyCA`. Aufgrund der besseren Performanz des `StackCA's` ist für viele Automatenkonfigurationen mit einer besseren Performanz als bei Klich zu rechnen.

8.3. Speicher- und Ladegeschwindigkeit der Datenbank gegenüber CSV-Dateien

In diesem Abschnitt werden die Speicher- und Ladegeschwindigkeiten der Datenbankausgabe sowie der CSV-Ausgabe in Abhängigkeit vom Simulationstag verglichen. Wie in Abbildung 8.3 zu erkennen ist, unterscheiden sich die gemessenen Ladegeschwindigkeiten für beide Verfahren bei einem Zellularen Automaten mit 10 000 Zellen erheblich. Während das Speichern der CSV-Datei quasi instantan erfolgt, scheint das Speichern bei der Datenbank etwa 0.7 Sekunden zu dauern. In beiden

Fällen ist die Bearbeitungszeit näherungsweise unabhängig vom Simulationstag. Das liegt daran, dass das Pythonprogramm in beiden Fällen nur die Kommunikation mit dem Betriebssystem (bei der CSV-Datei) bzw. der Datenbank messen kann. Der Grund ist, dass die eigentliche Speicherung in einem anderen Thread bzw. Prozess stattfindet. Daher lässt sich über die eigentliche Speicherzeit nur schwer eine valide Aussage treffen. Eine Speicherung ausschließlich in eine CSV-Datei bei ansonsten gleichen Bedingungen wie der bisherige Versuch hat gezeigt, dass die Speicherung über die Datenbankanbindung mit sehr hoher Wahrscheinlichkeit deutlich langsamer ist als bei der CSV-Datei.

Für das Laden eines Simulationstages (vgl. Abbildung 8.4) ist festzustellen, dass die Ladezeit im Falle der CSV-Datei linear mit dem Simulationstag zunimmt. Für die Datenbank ist hingegen eine konstante Zugriffszeit von etwa 0.3 Sekunden zu beobachten. Ferner schneidet die Datenbank bei mehr als 20 Simulationstagen besser ab als die CSV-Datei. Diese Befunde lassen sich darauf zurückführen, dass in der Datenbank Indexe genutzt werden können, um die gesuchten Daten zu finden. Im Fall der CSV-Datei muss die gesamte Datei geladen und linear durchsucht werden.

Ein Vergleich der hier dargestellten Ergebnisse mit anderen Zellenanzahlen zeigt, dass die Speicher- und Ladezeiten auch mit der Zellenanzahl zunehmen, da mehr Daten gespeichert bzw. geladen werden müssen.

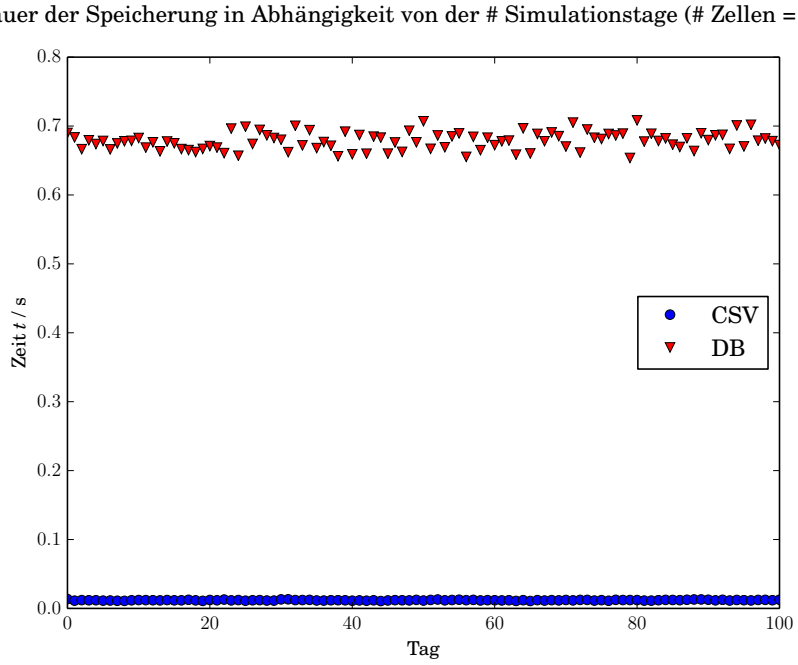


Abbildung 8.3.: Speicherdauer eines Zellularen Automaten mit 10 000 Zellen in Abhängigkeit von der Anzahl an Simulationstagen für `SimpleData` und `DBData`.

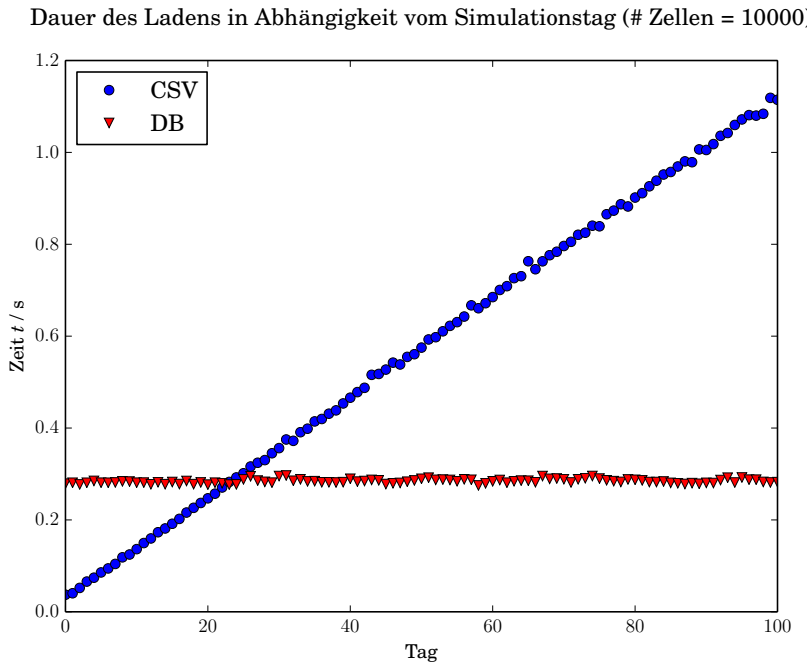


Abbildung 8.4.: Ladedauer eines Zellularen Automaten mit 10 000 Zellen in Abhängigkeit vom Simulationstag für `SimpleData` und `DBData`.

Abschließend ist festzustellen, dass das Speichern von Daten in eine Datenbank aufgrund der oben gemachten Beobachtungen u. U. wesentlich länger dauert als bei der Nutzung von CSV-Dateien. Dennoch bietet die Datenbanken einige Vorteile gegenüber CSV-Dateien, zum Einen können, wie oben gezeigt, große Datenmengen in vielen Fällen signifikant schneller geladen werden. Zum Anderen bietet die Nutzung einer Datenbank vielfältige technische Möglichkeiten, wie bspw. unterschiedliche implementierte Koordinatensysteme, welche bei der CSV-Ausgabe ggf. zusätzlich im Werkzeug (oder in Drittsoftware) implementiert werden müssten. Ferner bietet QGIS eine Datenbankanbindung an, wodurch die Daten komfortabler in das Geoinformationssystem geladen werden können als CSV-Dateien.

8.4. Erweiterbarkeit des Werkzeugs

In diesem Abschnitt wird die Erweiterbarkeit des Werkzeugs im Vergleich zu der Arbeit von Klich auf Basis von Kapitel 3.4.1 betrachtet. Hierzu werden die im Entwurf definierten Komponenten Input, Output und Simulationskern betrachtet.

Für die Dateneingabe (Komponente Input) ist festzustellen, dass eine Erweiterung durch Entwicklung einer neuen Klasse, welche von `AbstractInput` erbt möglich ist. Ggf. muss der Controller angepasst werden, an den übrigen Teilen (insbesondere dem Simulationskern) sind keine Änderungen erforderlich. In der Arbeit von Klich ist eine solche Schnittstelle nicht vorhanden. Die Dateneingabe erfolgt ausschließlich über die Nutzereingabe sowie aus Karteninformationen über QGIS. Daraus folgt zum Einen, dass beim Werkzeug von Klich externe Daten nur mit relativ hohem Aufwand genutzt werden können und zum Anderen, dass das Werkzeug an die Nutzung von QGIS gebunden ist.

Bei der Datenausgabe (Komponente Output) gilt ähnliches wie im Falle der Dateneingabe, auch hier ist eine einfache Erweiterung um weitere Ausgabeformate durch Entwicklung einer neuen Klasse (abgeleitet von `AbstractData`) gegeben. Bei Klich erfolgt die Datenspeicherung teilweise direkt im Simulationskern. Dadurch müssen Klassen, welche für die Simulation vorgesehen sind, bei einer Erweiterung um weitere Datenausgabeformate geändert werden.

Der Simulationskern dieses Werkzeug kann um weitere Zellulare Automatenmodell erweitert werden, indem von `AbstractCA` geerbt wird, ähnliches gilt auch für die Implementierung von Klich. Sollte es in Zukunft erforderlich sein weitere Simulationsverfahren (wie bspw. `StackCA`) zu implementieren, so kann als Oberklasse

`AbstractMatrix` verwendet werden. Bei Klich ist eine Unterscheidung in verschiedene Simulationsverfahren nicht vorgesehen. Die Erweiterbarkeit um neue Regelsätze ist in diesem Werkzeug mit der Klasse `AbstractRule` gegeben. Bei Klich ist diese Möglichkeit auch vorhanden. Zusätzlich gibt es einen Regelparser, wodurch der Nutzer Regeln in die GUI eingeben und auswerten lassen kann. Einen solchen Regelparser gibt es in dieser Implementierung nicht, da sich bei der bisherigen Nutzung des Werkzeugs von Klich kein Bedarf herausgestellt hat.

Kapitel 9

Zusammenfassung und Ausblick

9.1. Zusammenfassung

Für die Simulation des Ausbreitungsverhalten gibt es ein Werkzeug von Klich [49] welches auf dem Zellularen Automaten-Modell basiert. Jedoch weist das Werkzeug einige Schwächen bzgl. der Architektur und der Rechengeschwindigkeit auf, sodass hier eine Neuentwicklung eines Simulationskerns vorgenommen worden ist. Das hier entwickelte Werkzeug setzt wie in Tabelle 9.1 aufgelistet die Anforderungen aus Kapitel 4.2 um, wobei insbesondere eine leicht erweiterbare Architektur entwickelt worden ist. Dabei ist der eigentliche Simulationskern unabhängig von Drittsoftware lauffähig. Die beiden mit „0“ bewerteten Anforderungen D3 und A2 beziehen sich auf die Dateneingabe. Die Programmierschnittstelle ermöglicht es die Dateneingabe zu erweitern, jedoch sind in `SimpleInput` die Möglichkeiten für eine effektive Nutzung des Werkzeugs beschränkt (s. u. Ausblick Dateneingabe).

Tabelle 9.1.: Bewertung der Anforderungen an das Werkzeug.

Anmerkung: „+“: wird erfüllt, „-“: wird nicht erfüllt, „0“: weder + noch -. Die aufgeführten Abkürzungen beziehen sich auf Kapitel 4.2.

Generelle A.		**A. an die Datenhaltung**		**A. an den Simulationskern**		**A. aus Anwendungssicht**	
A1	+	D1	+	S1	+	G1	+
A2	0	D2	+	S2	+	G2	+
A3	+	D3	0	S3	+	G3	+

A4	+	D4	+	S4	+	G4	+
		D5	+	S5	+	G5	+
		D6	+	S6	+	G6	+
		D7	+	S7	+	G7	+
$\sum$	**+3**	$\sum$	**+6**	$\sum$	**+7**	$\sum$	**+7**

Des Weiteren ist das Werkzeug ausführlich getestet und auf die Performanzeigenschaften untersucht worden. Aufgrund dessen, dass das Werkzeug von Klich ähnlich wie der `EasyCA` arbeitet und der `StackCA` für viele Füllgrade performantere Ergebnisse erzielt, ist durch diese Arbeit ein Verbesserung bzgl. der Rechenzeit im Vergleich zu Klich erzielt worden.

Die Nutzung der Datenbank hat sich insbesondere beim Lesen von großen Zellularen Automaten und vielen Simulationstagen als performanter im Vergleich zu der Nutzung von CSV-Dateien herausgestellt. Des Weiteren bietet die Datenbank eine Anbindung an QGIS und stell weitere Funktionalitäten zur Verfügung.

9.2. Ausblick

In dieser Arbeit lag der Fokus auf der Verbesserung der Rechengeschwindigkeit bei dünn besetzten Zellularen Automaten, sowie der persistenten Speicherung der Simulationsergebnisse in einer räumlichen Datenbank. Daraus ergeben sich einige Fragen, welche in weiteren Arbeiten untersucht werden können:

Dateneingabe In der aktuellen Version des Werkzeugs werden die für die Simulation relevanten Daten wie bspw. die Temperatur als konstante Werte über `SimpleInput` zur Verfügung gestellt. Für eine realistischere Simulation ist jedoch anzunehmen, dass nicht alle Zellen die gleiche Temperatur aufweisen und die Temperatur nicht für alle Simulationstage gleich sein muss. Daher könnte eine zukünftige Arbeit untersuchen, wie bspw. gemessenen Temperaturen oder Ergebnisse aus Klimasimulationen für dieses Werkzeug nutzbar gemacht werden können. Dabei könnten Techniken aus dem Datamining sowie Methoden aus der Statistik genutzt werden, um für die einzelnen Zellen zu den unterschiedlichen Simulationstagen die notwendigen Daten bereitzustellen.

MALCAM-Modell Die Implementierung des MALCAM-Modells hält sich sehr eng an der Publikation von Linard et al. Die Dispersion in dieser Implementierung

unterscheidet sich von der aus dem Werkzeug von Klich, da die dort vorgenommenen Abweichungen vom MALCAM-Modell nicht auf ihre Korrektheit nachvollzogen werden konnten. Es könnte in Zusammenarbeit mit der Biologie geprüft werden, ob die bei Klich zu findende Dispersion im Sinne einer realistischen Mückenausbreitung korrekt ist. Des Weiteren erscheint es dem Autor nicht plausibel, die maximale Anzahl an Larven pro Zelle unabhängig von der Zellgröße sowie der Habitatqualität auf einen konstanten Wert zu setzen.

Weitere Regelsätze Das Werkzeug könnte um weitere Regelsätze ergänzt werden, um verschiedene Mückenarten simulieren zu können. Für diese Regelsätze sollte überprüft werden, ob die hier abgeleitete Heuristik zur optimalen Wahl der Simulationsverfahren angepasst werden muss.

Fehlerrechnung Sowohl gemessene- und simulierte Eingabedaten als auch die verwendeten Modelle sind fehlerbehaftet. Aus Entwicklersicht könnte eine Fehlerrechnung ähnlich wie sie im Laborumfeld üblich ist genutzt werden, um die Güte der Simulationsergebnisse abzuschätzen. Problematisch ist jedoch, dass die Modellfehler nur sehr schwer abgeschätzt werden können und möglicherweise größere Auswirkungen auf die Simulationsergebnisse haben als Fehler der Eingabgedaten. Daher könnte in Zusammenarbeit mit der Biologie untersucht werden, ob eine explizite Fehlerrechnung in diesem Werkzeug sinnvoll ist.

Visualisierung und Analyse der Simulationsergebnisse Dabei kann untersucht werden, welche Drittwerkzeuge für eine optimale Visualisierung und Analyse der Simulationsergebnisse genutzt werden können.

GUI Das Werkzeug von Klich implementiert eine interaktive Steuerung der Simulation sowie eine Visualisierung der Ergebnisse. In Zusammenhang mit dem vorherigen Punkt könnte überlegt werden, ob eine Visualisierung in einem externen (freien) Werkzeug wie QGIS besser ist, als in einer (neuen) GUI. Entsprechend müsste ggf. nur noch eine GUI für die Steuerung dieses Werkzeugs entwickelt werden. Dabei könnte auch die Reintegration dieses Werkzeugs in QGIS oder eine andere Drittsoftware ins Auge gefasst werden.

Anhang
A

Verwendete Software

In diesem Anhang werden Angaben zur verwendeten Software gemacht. Dabei werden die Bedingungen (z.B. Programmpakete unter GNU Linux) aufgeführt, unter denen die aufgeführten Programme lauffähig sind. Es handelt sich hierbei *nicht* um eine vollständige Installationsanleitung. Gleichwohl werden an einigen Stellen Anmerkungen gemacht, wenn Besonderheiten aufgetreten sind und das Problem nicht mit für das jeweilige Betriebssystem übliche „Standardwissen“ gelöst werden kann, um die Angaben leichter reproduzierbar zu machen.

Eine exportierte Version der virtuellen Maschine inklusive allen verwendeten Programmen befindet sich auf der beigelegten DVD s. S. 160.

A.1. Oracle VirtualBox

Oracle VirtualBox [79] ist eine freie Virtualisierungssoftware unter der GNU General Public License (GPL). VirtualBox ist für gängige Betriebssysteme (z.B. GNU Linux, Windows, Mac) verfügbar. Es können verschiedenste Gastsysteme installiert werden. In dieser Bachelorarbeit wurde Bodhi Linux verwendet.

A.2. Bodhi Linux

Bodhi Linux [10] ist eine leichtgewichtigte GNU/Linux-Distribution, welche auf Ubuntu basiert. Diese Distrubution wird hier verwendet, da viele der im Internet für Ubuntu, aufgrund dessen Popularität, verfügbaren Hilfen genutzt werden können, ohne jedoch eine vollständige (u. U. stärker Ressourcen nutzende) Ubunut-Version installieren zu müssen.

Um die weiteren Installationsarbeiten zu vereinfachen ist Synaptic [60] installiert worden. Dabei handelt es sich um eine freie GUI für den Debian Package Manager (DPKG), welcher standardmäßig von vielen Linux-Distributionen (u. a. Ubuntu und Bodhi Linux) verwendet wird [1].

Der Benutzername, sowie die (Root-)Passwörter lauten `bodhi`.

A.3. QGIS

QGIS ist ein freies Geoinformationssystem (GIS), welches die Basis des MosquitoCA-Plugins bildet. QGIS lässt sich auf vielen gängigen Plattformen (u. a. GNU/Linux und Windows) installieren [95].

Die Installation unter Bodhi Linux erfolgt aus den Ubuntu PPA's (Personal Package Archive), um die aktuelle QGIS-Version nutzen zu können. Die Installation wird in [115] beschrieben. Um eventuellen Änderungen der Webseite vorzubeugen wird in Abbildung A.1 der relevante Teil der Webseite als Screenshot aufgeführt.

PPA

Von QGIS werden auf deren Homepage verschiedene Paketquellen für die Installation angegeben. Die Paketquelle ist manuell in die Datei sources.list hinzuzufügen (siehe PPA). Im Folgende wird die aktuelle und stabile Version (2.14.x Essen) von QGIS für die Ubuntu Version 14.04 (Codename "trusty") beschrieben.

```
sudo sh -c 'echo "deb http://qgis.org/debian trusty main" >> /etc/apt/sources.list'
```

danach wird zur weiteren Verwendung der Quelle der Signaturschlüssel eingespielt. Die Schlüssel ID erhält man auf der auf der Launchpad-Seite des PPA. Sollte man den Schlüssel nicht einspielen wird bei einer Systemaktualisierung (apt-get update) die SchlüsselID in der GPG-Fehlermeldung hinter NO_PUBKEY ebenfalls angezeigt.

```
wget -O - http://qgis.org/downloads/qgis-2015.gpg.key | gpg --import
gpg --fingerprint 3FF5FFCAD71472C4
gpg --export --armor 3FF5FFCAD71472C4 | sudo apt-key add -
```

Alternativ dazu kann man den Schlüssel auch in einem Schritt (ohne manuelle Schlüsselverifikation) einspielen

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key 3FF5FFCAD71472C4
```

Hinweis!

Zusätzliche Fremdquellen können das System gefährden.

Nach dem Aktualisieren der Paketquellen erfolgt die Installation wie oben angegeben. Alternativ zum stabilen PPA kann auch die Entwickler-Version hilfreich sein. Diese steht sowohl für 12.04 als auch 14.04 zur Verfügung.

Abbildung A.1.: Installationsanweisung von QGIS-Essen in Ubuntu [115, Version vom 20.05.2016].

A.4. Python

Wie in Kapitel 5.1 diskutiert wird das Werkzeug in Python 3.5.1 [92] (am 06.12.2015 veröffentlicht) entwickelt. Es werden Installationsmedien für verschiedene Plattformen bereitgestellt. Die Installationsdetails werden mit dem Installationsmedium in einer README-Datei mitgeliefert. Daher wird hier die Installation nicht beschrieben.

A.4.1. Pip

Pip [88] ist ein Paketverwaltungsprogramm für Python, welches genutzt werden kann um weitere Pythonpakete zu installieren. Um Pip zu installieren kann die Datei `get-pip.py` aus [86] heruntergeladen und analog wie in der Dokumentation [87] ausgeführt werden.

A.4.2. Psycopg2

Psycopg2 [119] ist der am häufigsten genutzte freie Treiber für PostgreSQL. Es sind noch weitere für PostgreSQL geeignete Treiber verfügbar (kleine Übersichten finden sich in [90] und [84]). Ohne hier eine vollständige Diskussion der Gründe anzuführen, ist die Wahl auf Psycopg2 gefallen, da dieser Treiber für alle gängigen Plattformen verfügbar ist und sowohl die Python API als auch die SQL-Standards in für diese Arbeit ausreichender Weise implementiert. Die Installation des Treibers kann mittels des oben beschriebenen Pip Paketverwaltungsprogramms erfolgen.

A.4.3. Weitere Pakete

Die nachfolgenden Pakete sind standardmäßig in Python installiert und werden für diese Arbeit verwendet:

csv Speichern und öffnen von CSV-Dateien.

math (exp) Berechnung der Exponenten im MALCAM-Modell.

unittest Durchführung der automatisierten Tests (insbesondere die Testsuite).

os Löschen von CVS-Dateien während der Tests.

time (clock) Zeitmessung während der Benchmark-Tests.

A.5. PostgreSQL

PostgreSQL ist eine freie objektrelationale Datenbank, welche für viele gängige Plattformen verfügbar ist [112]. Die aktuelle Version 9.5.3 kann von [23] heruntergeladen und installiert werden. Zur Installation auf Bodhi Linux werden zunächst mit root-Rechten die Ausführungsrechte der heruntergeladenen run-Datei im Terminal gesetzt `sudo chmod +x postgresql-9.5.3-1-linux.run` [123, S. 178]. Anschließend erfolgt die Installation mittels `sudo ./postgresql-9.5.3-1-linux.run`.

A.5.1. PostGIS

PostGIS ist eine freie Erweiterung von PostgreSQL um geografische Daten und Funktionen [99]. PostGIS 2.2 kann über den mit PostgreSQL mitgelieferten Stack Builder via GUI installiert werden.

A.5.2. pgAdmin III

PgAdmin III [83] ist ein grafisches Datenbankverwaltungswerkzeug für PostgreSQL und wird automatisch mit diesem installiert. PgAdmin III wird in dieser Arbeit zur Verwaltung der Datenbank genutzt.

A.6. Eclipse

Eclipse ist eine freie IDE (Integrated Development Environment), welche auf vielen gängigen Plattformen genutzt werden kann. Eclipse wurde in diesem Projekt verwendet, da es eine vielzahl an Extensions gibt, welche die Arbeit erleichtern. Wichtige hier verwendete Extensions werden unten weiter aufgeführt. Um die aktuelle Version Eclipse Mars [18] unter Bodhi Linux nutzen zu können, müssen gcj-jre und openjdk-8 über Synaptic bzw. das PPA (ppa:openjdk-r/ppa [116]) installiert werden [114]. Die standardmäßig unter Bodhi Linux installierte GNU-Java-Version reicht hierbei nicht aus. Für Eclipse würde das openjdk-7 aus den Paketquellen ausreichen, allerdings ist damit PyDev (s. u.) nicht nutzbar. Um Eclipse nutzen zu können, wird anschließend das Installationsarchiv nach `/home/bodhi` entpackt. Für eine einfachere Nutzung wird ein Startskript in `/home/bodhi/bin` und eine Verknüpfung in den „Favorite Applications“ der Bodhi Linux Desktopumgebung angelegt.

A.6.1. UML Designer

UML Designer ist ein graphisches Eclipse-Plugin, um UML-Diagramme (Unified Modeling Language) zu zeichnen und kann über den Eclipse Marketplace[1] installiert werden [77]. Aus Sicht des Autors ist UML Designer bzgl. Funktionalität und Usability im Vergleich zu anderen UML-Werkzeugen wie z. B. UMLet [4] oder Papyrus [3] das für dieses Projekt am besten geeignete Werkzeug. Es hat sich zu einem späteren Zeitpunkt herausgestellt, dass sich aufgrund eines Programmfehlers keine Sequenzdiagramme erstellen ließen, daher ist für diese Aufgabe abweichend Umbrello UML Modeller [113] (welches sich nicht auf der Virtuellen Maschine befindet) verwendet worden.

[1] Der Eclipse Marketplace [18] bietet die Möglichkeit Eclipse Extensions direkt aus Eclipse automatisch zu installieren, ohne die Erweiterungen extra herunterladen zu müssen.

A.6.2. EGit

EGit ist eine Eclipse Extension zur Verwendung von git [2]. Git ist ein verteiltes Versionskontrollsystem welches zur kollaborativen Entwicklung von Software genutzt werden kann [16].

A.6.3. PyDev

PyDev [12] ist ein Eclipse-Plugin und bietet einige Funktionalitäten wie bspw. einen Debugger, welche die Entwicklung vereinfachen.

A.7. Doxygen

Bei Doxygen handelt es sich um ein GPL lizenziertes freies Dokumentationswerkzeug [33]. Als GUI kann Doxywizard genutzt werden [34]. Als Alternative hätte Sphinx verwendet werden können [13], welches das bei Python-Projekten am häufigsten genutzte Werkzeug ist. Sphinx hat jedoch im Gegensatz zu Doxygen einige Nachteile. Zum Einen gibt es keine GUI, so dass die Nutzung unkomfortabler ist. Zum Anderen bietet Doxygen von Haus aus die Möglichkeit Diagramme automatisch zu erstellen. Des Weiteren kann GraphViz über die GUI eingebunden werden, wodurch diverse Diagramme generiert werden können.

A.8. GNU Octave

GNU Octave [43] ist eine freie Software / Programmiersprache welche zur Lösung numerischer Probleme eingesetzt werden kann. In dieser Arbeit wird GNU Octave eingesetzt, um Ausgaben des entwickelten Simulationsprogramms testweise möglichst schnell und komfortabel grafisch darstellen zu können. Aufgrund einer Inkompabilität mit PyDev konnte GNU Octave nicht in akzeptabler Zeit auf der Virtuellen Maschine installiert werden.

Anhang
B

Codebeispiele

In diesem Kapitel werden einige Codebeispiele aus der eigentlichen Bachelorarbeit ausgelagert. Dabei wird insbesondere auf die Nutzung von Doxygen als Dokumentationswerkzeug, für diese Arbeit getroffene Programmierkonventionen sowie die Organisation der verschiedenen Klassen eingegangen. Abschließend wird der GNU GPL3 Copyright-Hinweis, wie er in jedem Quellcodefile zu finden ist, aufgeführt.

B.1. Nutzung von Doxygen

Die Nutzung von Doxygen wird hier anhand von Listing B.1 erläutert. Das Beispielmodul implementiert keine „nützlichen“ Funktionalitäten und dient lediglich als Minimalbeispiel. Die erste Zeile eines Doxygen-Kommentars beginnt mit `##`. Jede weitere Kommentarzeile beginnt mit `#`. Die Klasse `Example` wird mit diesen Mitteln kommentiert (s. Z. 4 f.). Zusätzlich können einige Schüsselwörter (`@...`) genutzt werden. Die wichtigsten in dieser Bachelorarbeit verwendeten Schlüsselwörter werden im Folgenden aufgeführt. Für die Kommentierung von Paketen wird `@package` genutzt. Zur Kommentierung von Funktionen werden für die übergebenen Parameter `@param` (s. Z. 7), für den Rückgabewert `@return` (s. Z. 8) und für geworfene Exception `@exception` (s. Z. 9) verwendet.

Aus den mit den vorgestellten Mitteln können mit Doxygen automatisch komfortabel lesbare Dokumente in unterschiedlichen Formaten erstellt werden. In dieser Bachelorarbeit werden html- und LaTeX-Dokumente erstellt. Die html-Seiten eignen sich für das Lesen am Computerbildschirm, da wie auf üblichen Internetseiten, „gesurft“ werden kann. Die LaTeX-Dokumente können zu einem PDF-Dokument kompiliert werden, welches sich für den Druck eignet.

Listing B.1: Nutzung von Doxygen anhand eines Beispielmoduls `example`.

```
1  ## @package example is just for demonstration
2
3  ## Example is just an example class.
4  # More comments ...
5  class Example:
6    ## Function to initialize an Example-Object.
7    # @param x the given value for self.__x
8    # @return self.__x which is the value of the given x
9    # @exception the given value of x = 42 is invalid
10   def __init__(self, x):
11     if (x == 42):
12       raise Exception("42 is invalid")
13     else:
14       self.__x = x
15       return self.__x
```

B.2. Konventionen

In diesem Abschnitt werden einige Konventionen definiert, welche sich aus der Nutzung von Python als Programmiersprache ergeben.

B.2.1. Interfaces

Python kennt im Gegensatz zu Java keine Interfaces. Um diesem Konzept möglichst nahe zu kommen, werden in dieser Arbeit Klassen verwendet, welche von `object` erben (Listing B.2 Z. 1), deren Methoden nicht aus implementiert werden. Für diesen Zweck stellt Python das Schlüsselwort `pass` bereit. Aus Sicht des Autors erscheint es jedoch sinnvoller eine Exception mit einer Fehlermeldung auszugeben, welche dem implementierenden Programmierer einen Hinweis auf das Interface gibt (Z. 2 f.). Analog wird bei der `__init__`-Funtion verfahren (Z. 5 f.), so dass der Versuch eine Instanz des Interfaces zu erzeugen mit einer Fehlermeldung quittiert wird.

Listing B.2: Gekürzte und unkommentierte Version von `Input` zur Definition von Interfaces in dieser Arbeit.

```
1 class Input(object):
2     def __init__(self):
3         raise Exception("Input is an interface")
4 
5     def getDataOfCell(self, x, y, t):
6         raise NotImplementedError("getDataOfCell has not yet been
              implemented")
```

B.2.2. Abstrakte Klassen

Für Abstrakte Klassen wird ähnlich verfahren, wie im vorherigen Abschnitt über Interfaces. Abstrakte Klassen unterscheiden sich von Interfaces dadurch, dass sie Attribute aufweisen können und es vollständig implementierte Methoden geben kann (das gilt auch für die `__init__`-Funktion).

B.2.3. Anmerkungen zu Zugriffsrechten bei Klassenfunktionen und -attributen

In Python gibt es als Zugriffsschutz bei Klassenfunktionen nur die Wahl zwischen private (von außen nicht über die übliche `obj.__function() Notation` aufrufbar[1]) und public (auch externer Aufruf). Damit gibt es keine Möglichkeit in einer Oberklasse (oder Abstrakten Klasse) erbenden Klassen Funktionen zur Verfügung zu stellen, welch von dritten Klassen nicht genutzt werden können. Per Konvention sind in dieser Arbeit Funktionen protected, wenn deren Namen mit einem „_“ beginnt und private, wenn deren Namen mit „__“ beginnt.

B.3. Organisation der Klassen

Um das Arbeiten mit dem Quellcode zu vereinfachen, werden alle Klassen nach dem selben Schema organisiert. Beispielhaft wird eine gekürzte Version der Klasse `AbstractMatrix` (Listing B.3) vorgestellt (Klassen- und Funktionenkommentare gemäß Abschnitt B.1 werden hier nicht aufgeführt). Nach der Klassendefinition erfolgt die Implementierung der `__init__`-Funktion (s. Z. 2 f.), die bei der Initialisierung eines Objektes von Python intern aufgerufen wird. Anschließend werden alle öffentlichen Funktionen, welche nicht getter- und setter-Funktionen sind, in einem Block definiert (s. Z. 5 f.). Diese Funktionen werden von der Umwelt aufgerufen und sind zu großen Teilen im Klassendiagramm Abbildung 5.7 modelliert worden. Die nächsten beiden Blöcke bilden die getter- (s. Z. 8 ff.) und setter-Funktion (s. Z. 12 ff.). Beide Blöcke werden nach den Zugriffsrechten public und protected (und ggf. private) sortiert. An die getter- und setter-Blöcke schließen sich alle übrigen Methoden mit protected Zugriffsrechten an (s. Z. 16 ff.), welche nur von der Oberklasse sowie von abgeleiteten Klassen genutzt werden sollen (wie schon oben angedeutet lässt es sich in der Programmiersprache Python nicht verhindern, dass protected Funktionen auch von dritten Klassen genutzt werden). Abschließend werden die internen (privaten) Funktionen aufgeführt (s. Z. 19 f.), welche nicht von außerhalb (auch nicht aus abgeleiteten Klassen) genutzt werden sollen.

[1] Tatsächlich lassen sich „private“ Funktionen über eine alternative Notation von außerhalb aufrufen: `obj._ClassName__function()`.

Listing B.3: Gekürzte und unkommentierte Version von `AbstractMatrix` als Beispiel, wie die Python-Klassen in dieser Arbeit organisiert werden. Zur Veranschaulichung werden einige Funktionen aufgeführt, welche es in der Implementierung nicht gibt.

```
1  class AbstractMatrix(object):
2      # init function
3      def __init__(self, combiCA, numx, numy, initCA):
4
5      # public functions (interfaces for the environment)
6      def doStep(self):
7
8      # getter functions
9      def getCA(self):
10     def _getOldCell(self, cell):
11
12     # setter functions
13     def setCA(self, ca):
14     def _setXYZ(self, xyz):
15
16     # protected functions (can be used from derivative classes)
17     def _toggleUseA(self):
18
19     # private internal functions
20     def __privateFunction(self):
```

B.4. Copyright-Hinweis

Listing B.4: Copyright-Hinweis am Beispiel des Moduls `input.py`, erzeugt mit Hilfe von Anjuta [64].

```
1  # -*- Mode: Python; coding: utf-8; indent-tabs-mode: t; c-basic-offset: 4;
       tab-width: 4 -*-
2  #
3  # input.py
4  # Copyright (C) 2016 Stephan Adolf <Stephan.Adolf@uni-oldenburg.de>
```

```
5  #
6  # input.py is free software: you can redistribute it and/or modify it
7  # under the terms of the GNU General Public License as published by the
8  # Free Software Foundation, either version 3 of the License, or
9  # (at your option) any later version.
10 #
11 # input.py is distributed in the hope that it will be useful, but
12 # WITHOUT ANY WARRANTY; without even the implied warranty of
13 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
14 # See the GNU General Public License for more details.
15 #
16 # You should have received a copy of the GNU General Public License along
17 # with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Anhang C

Plots

In diesem Anhang werden einige Visualisierungen aufgeführt, welche nicht im Hauptteil zu finden sind.

C.1. Plots des MALCAM-Modells

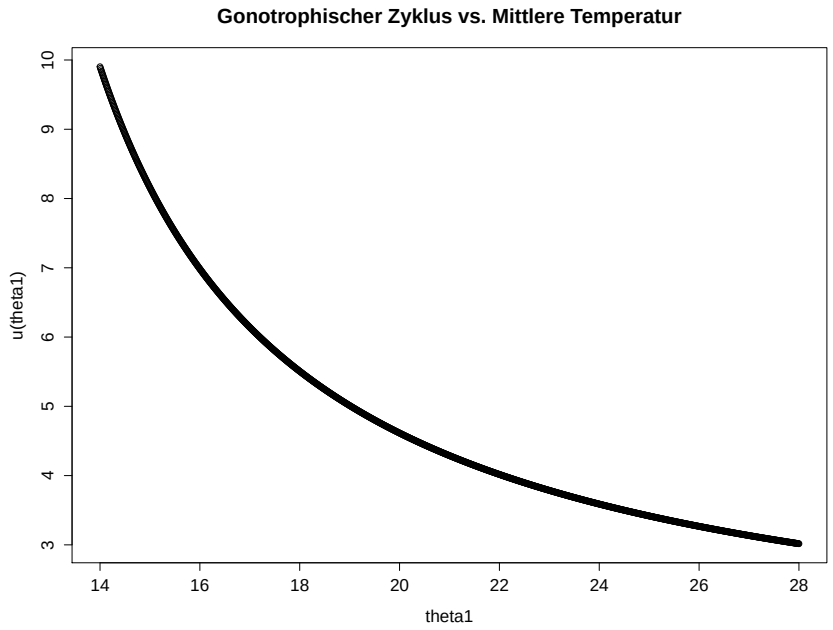


Abbildung C.1.: Dauer des Gonotrophischen Zyklus u(theta) in Abhängigkeit von der Temperatur theta dargestellt im Temperaturintervall [14 °C, 28 °C].

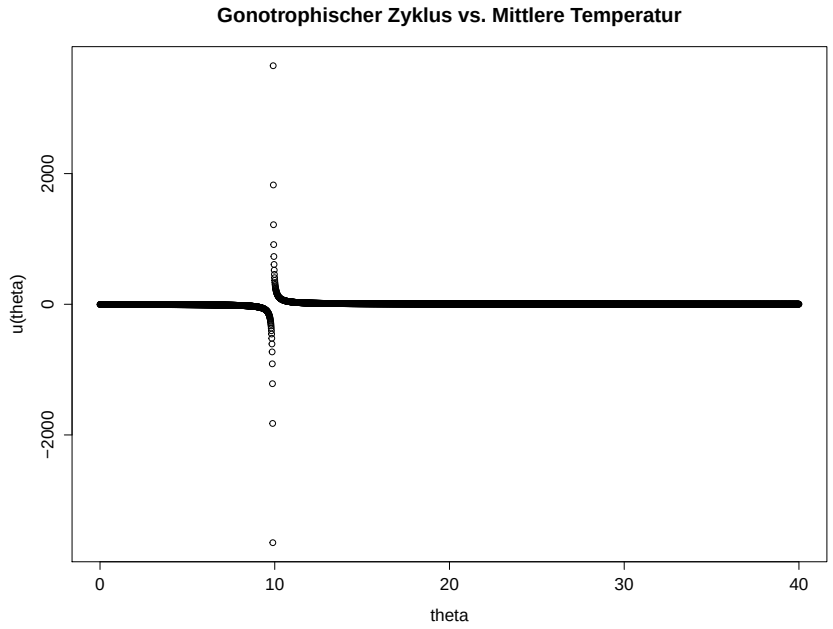


Abbildung C.2.: Dauer des Gonotrophischen Zyklus u(theta) in Abhängigkeit von der Temperatur theta dargestellt im Temperaturintervall $[0\,°C, 40\,°C]$.

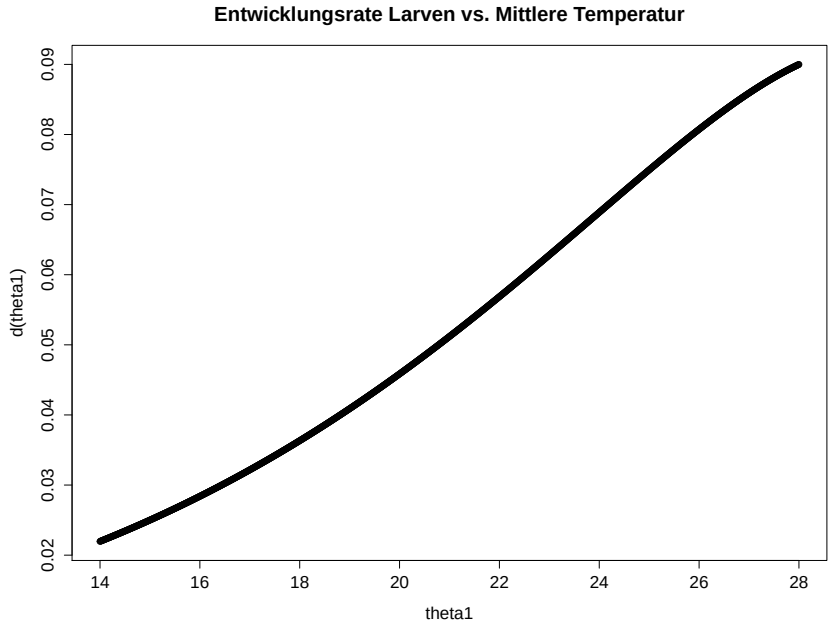


Abbildung C.3.: Größe der Entwicklungsrate der Larven d(theta) in Abhängigkeit von der Temperatur theta dargestellt im Temperaturintervall $[14\,°C, 28\,°C]$.

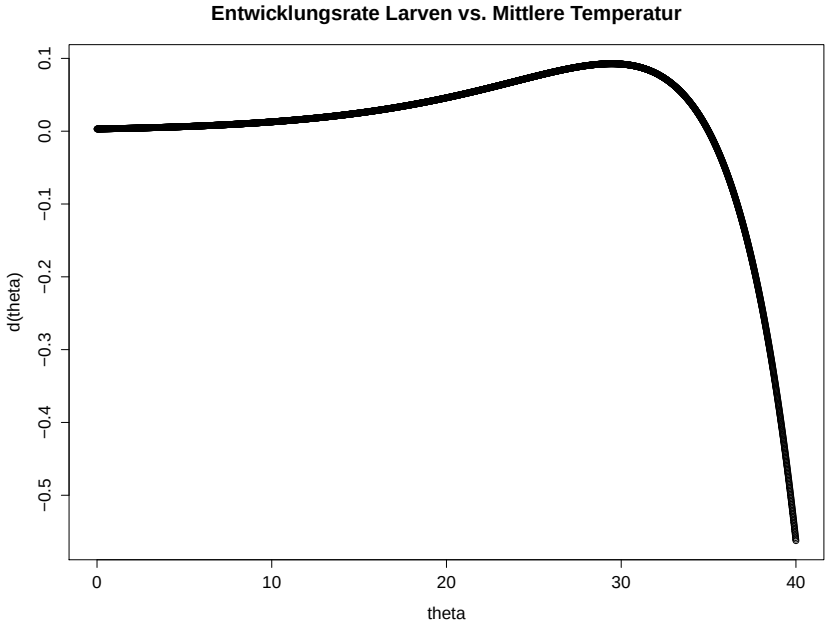


Abbildung C.4.: Größe der Entwicklungsrate der Larven d(theta) dargestellt in Abhängigkeit von der Temperatur theta im Temperaturintervall $[0\,°C, 40\,°C]$.

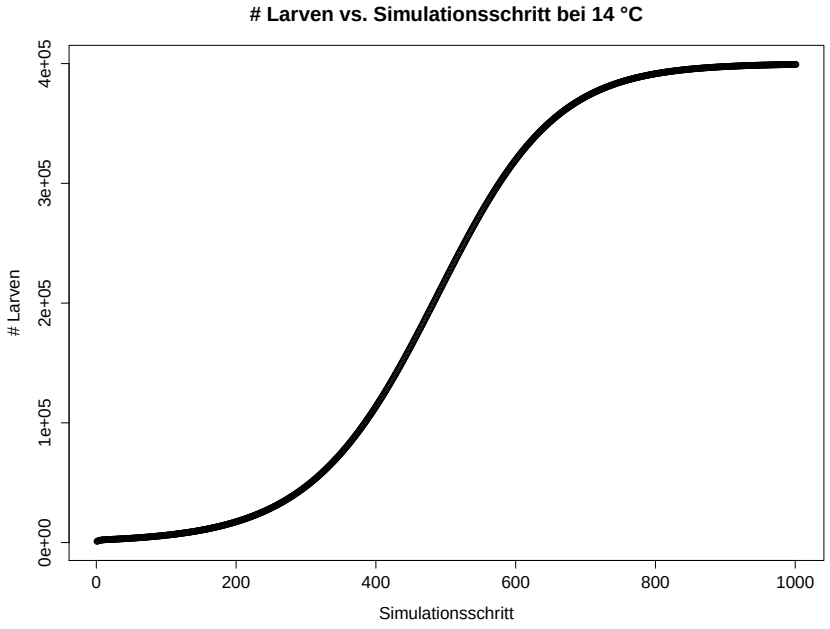


Abbildung C.5.: Anzahl an Larven (# Larven) dargestellt in Abhängigkeit vom Simulationsschritt für eine Temperatur von $14\,°C$.

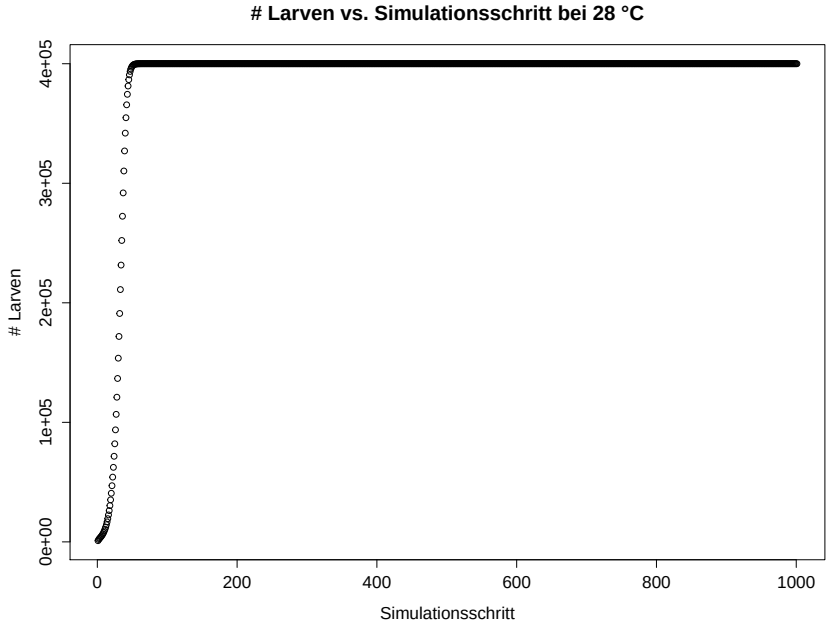


Abbildung C.6.: Anzahl an Larven (# Larven) dargestellt in Abhängigkeit vom Simulationsschritt für eine Temperatur von 28 °C.

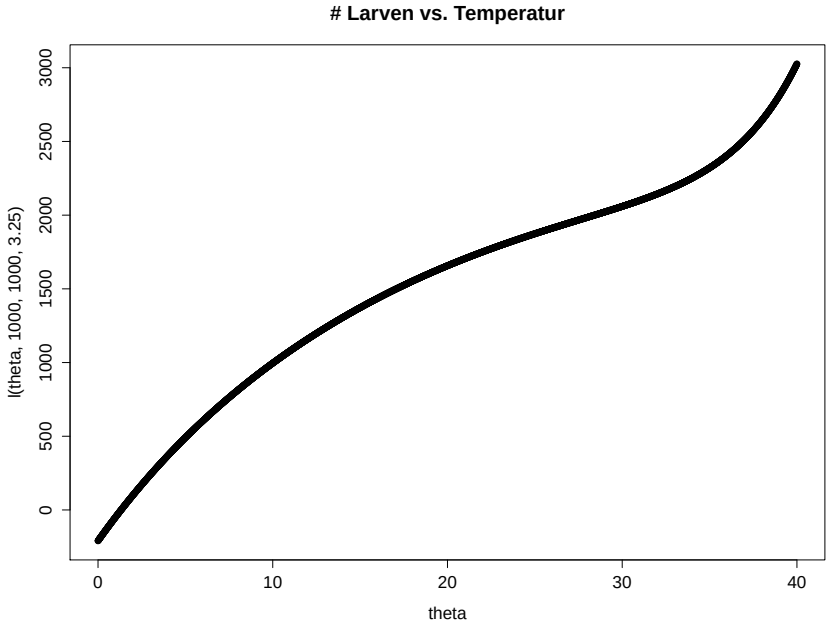


Abbildung C.7.: Anzahl der Larven (l(theta, 1000, 1000, 3.25)) dargestellt in Abhängigkeit von der Temperatur theta im Temperaturintervall $[0\,°C, 40\,°C]$.

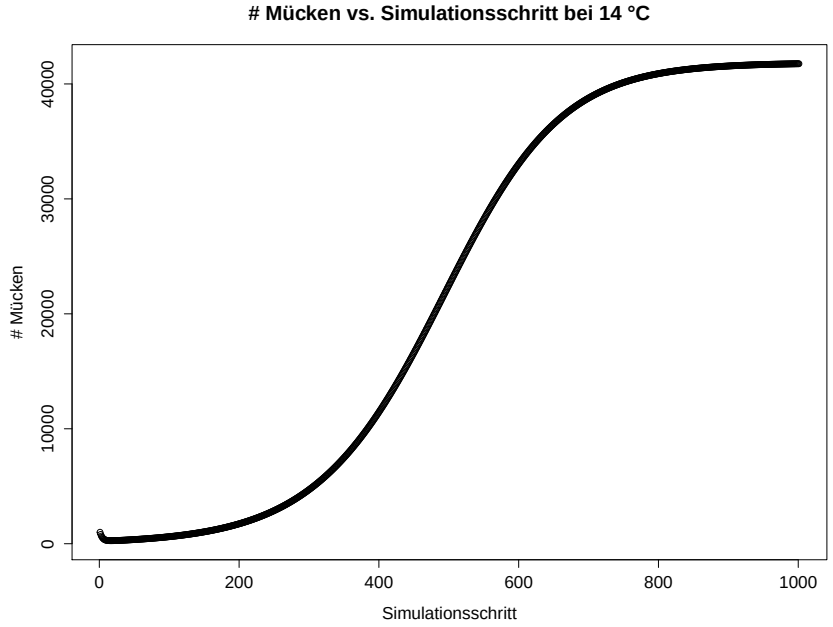


Abbildung C.8.: Anzahl an Mücken (# Mücken) dargestellt in Abhängigkeit vom Simulationsschritt für eine Temperatur von 14 °C.

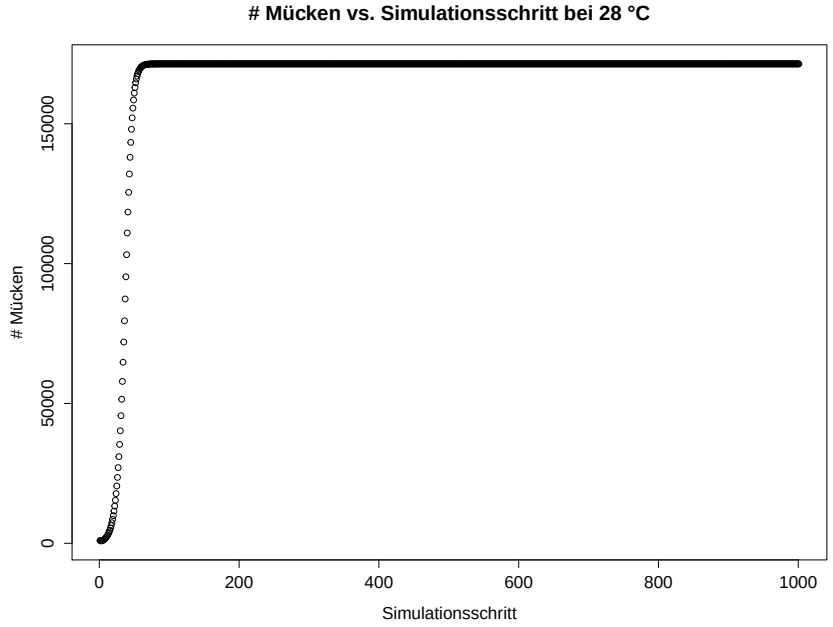


Abbildung C.9.: Anzahl an Mücken (# Mücken) dargestellt in Abhängigkeit vom Simulationsschritt für eine Temperatur von 28 °C.

C.2. Testplots der Skripttests

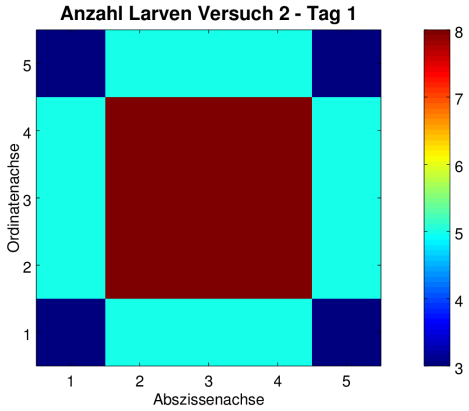
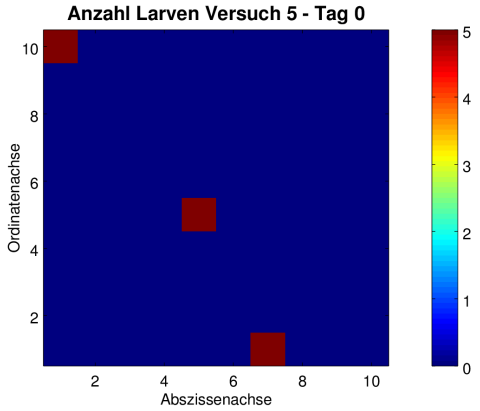
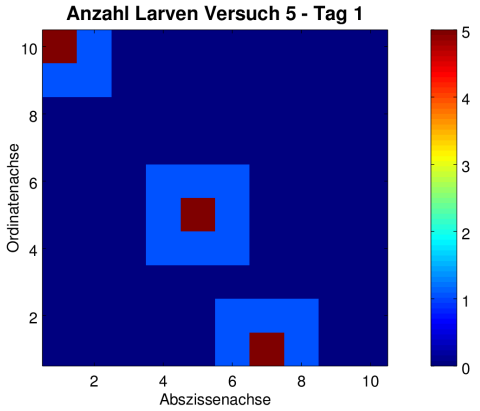


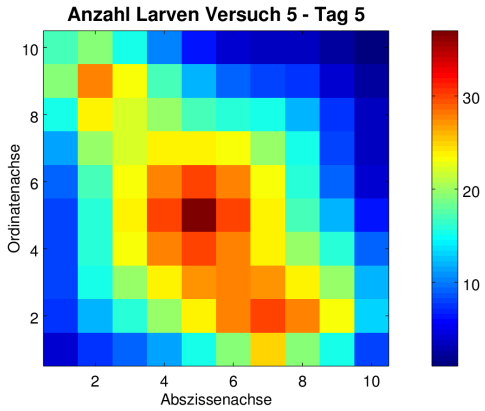
Abbildung C.10.: Visualisierungen der Anzahl der Larven (der Plot für die erwachsenen Mücken ist wegen der verwendeten Regel gleich) aus Versuch 2 der Skripttests. Im Test wurden `EasyCA` und `TestRule` (eingeschaltete Dispersion) verwendet.



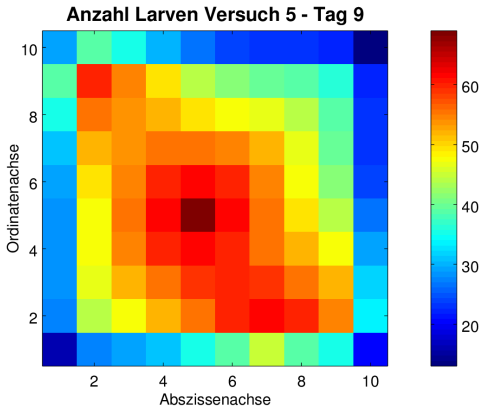
(a) Initialzustand.



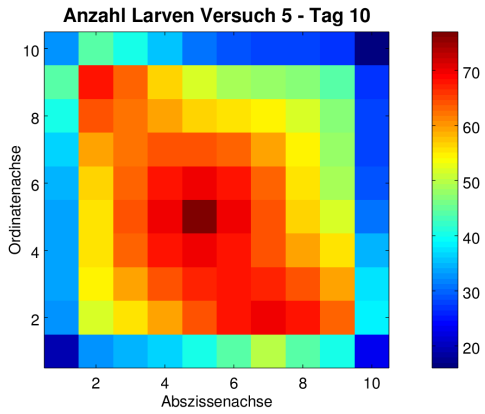
(b) Tag 1.



(c) Tag 5.

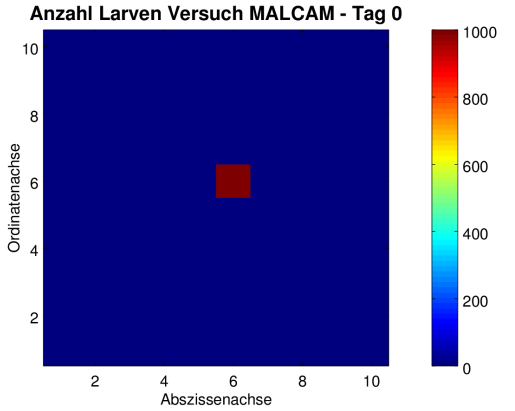


(d) Tag 9.

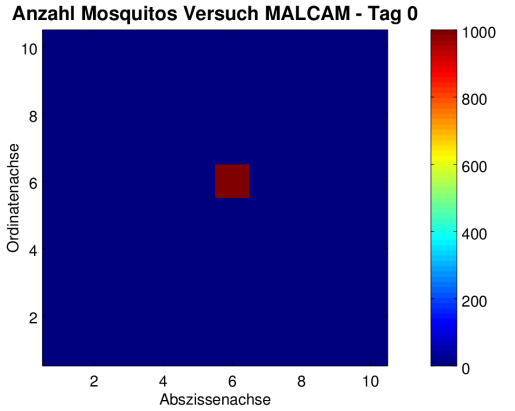


(e) Tag 10.

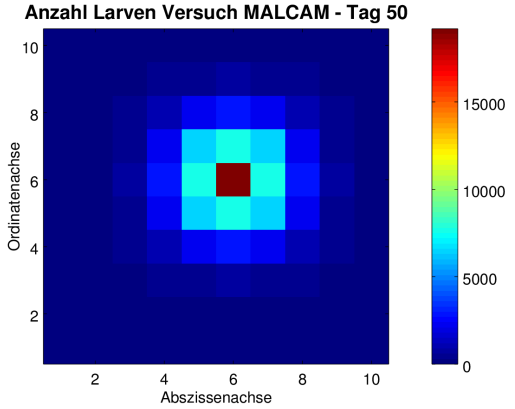
Abbildung C.11.: Visualisierungen der Anzahl der Larven (die Plots für die erwachsenen Mücken sind wegen der verwendeten Regel gleich) aus Versuch 5 der Skripttests. Im Test wurden `EasyCA` und `TestRule` (eingeschaltete Dispersion) verwendet.



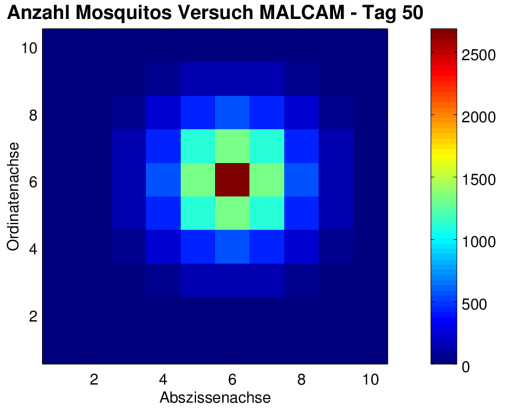
(a) Initialzustand Larven.



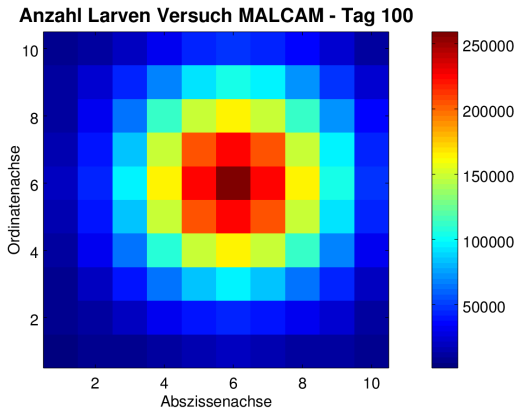
(b) Initialzustand Mücken.



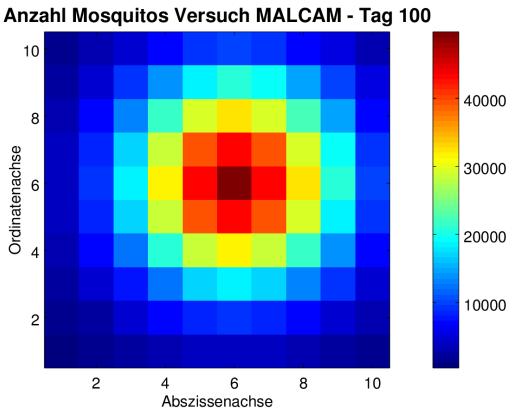
(c) Tag 50 Larven.



(d) Tag 50 Mücken.

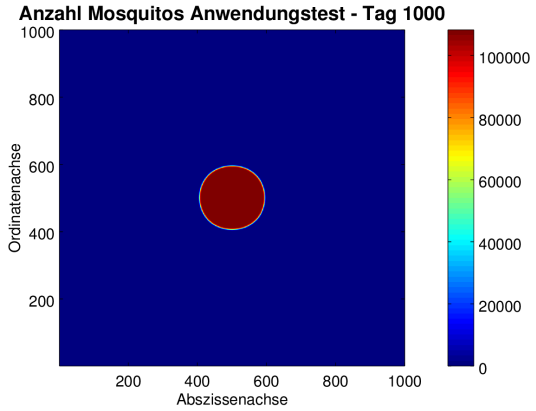


(e) Tag 100 Larven.

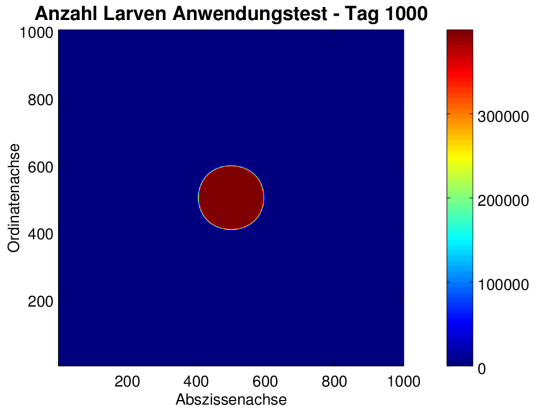


(f) Tag 100 Mücken.

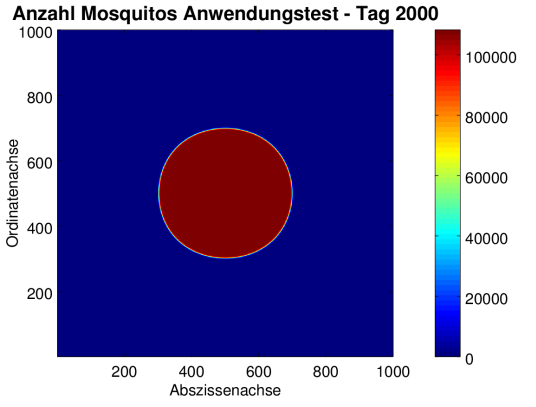
Abbildung C.12.: Visualisierungen der Anzahl der Larven und Mücken aus dem `malcamRun.py` Skripttest für den Initialzustand sowie nach 50 bzw. 100 Simulationstagen. Im Test wurden `StackCA` und das MALCAM-Modell verwendet.



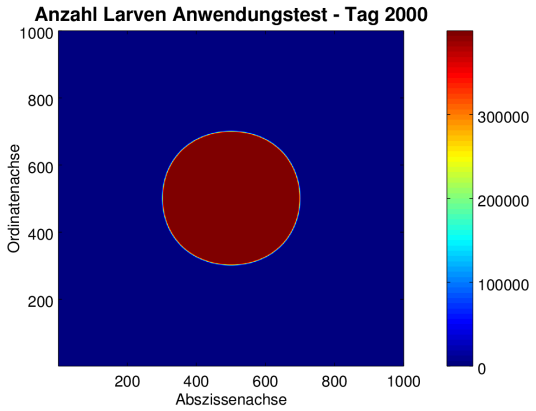
(a) Mücken nach 1000 Tagen.



(b) Larven nach 1000 Tagen.



(c) Mücken nach 2000 Tagen.



(d) Larven nach 2000 Tagen.

Abbildung C.13.: Visualisierung der Simulationsergebnisse eines 1000×1000 Zellularen Automaten aus dem `functionalityTest.py` Skripttest. Im Test wurde das MALCAM-Modell verwendet.

Anhang
D

Anmerkungen zum auf der DVD befindlichen Werkzeug

Der Quellcode des Werkzeugs wird auf der DVD s. S. 160 im Verzeichnis `mosquito-simulation` mitgeliefert. Für eine reibungslose Nutzung müssen einige Dinge beachtet werden. Zunächst müssen Python, PostgreSQL / PostGIS und psycopg2 installiert sein. Für die Datenbank ist zu beachten, dass die Erweiterung PostGIS geladen worden ist (bspw. über pgAdmin oder `CREATE EXTENSION postgis`).

Damit der Test `TestDBData` ausgeführt werden kann, ist es erforderlich dem Test den Namen der Datenbank, den Host der Datenbank, den Nutzernamen sowie dessen Passwort mitzuteilen. Dazu müssen die entsprechenden Variablen in der Datei `setupTests.py` mit korrekten Werten zu setzen. Des Weiteren schreiben / lesen einige Tests CSV-Dateien im `/test`-Verzeichnis. Daher muss die Variable `DIRECTORY` in der genannten Datei auf das `/test`-Verzeichnis verweisen.

Sollen die Skripttests im `/run`-Verzeichnis ausgeführt werden, so werden einige Verzeichnisse für die Dateneingabe sowie die Datenausgabe benötigt. Ein Beispiel für diesen Verzeichnisbaum ist im Verzeichnis `/tree` zu finden. Es bleibt dem Nutzer überlassen diesen Verzeichnisbaum an einen geeigneten Ort zu kopieren[1]. In der Datei `setupRun.py` muss die Variable `DIRECTORY` auf den Pfad zu diesem Verzeichnisbaum speichern. Analog wie bei den Tests ist in der hier genannten Datei mit den Datenbankeinträgen zu verfahren.

Für die Angabe der Pfade ist unter Windows zu beachten, dass „/“ und nicht „\“ verwendet wird. Entsprechend führt die unter Windows übliche Notation `DIRECTORY = "C:\meinverzeichnis\tree"` zu einer Fehlermeldung. Korrekt ist die unter Unix übliche Notation `DIRECTORY = "C:/meinverzeichnis/tree"`.

[1] Soll das Projekt in einem (git-)Repository gespeichert werden, so ist aufgrund der Größe des Verzeichnisbaums dringend zu empfehlen diesen nicht im Repository zu speichern.

Literatur

[1] Alioth.debian.org. *Alioth: Dpkg package manager: Project Home*. 2016. URL: https://alioth.debian.org/projects/dpkg (besucht am 20. 05. 2016).

[2] Chris Aniszczyk. *EGit - Git Team Provider*. 2016. URL: https://marketplace.eclipse.org/content/egit-git-team-provider (besucht am 20. 05. 2016).

[3] AtoS, CEA LIST und Samares Engineering. *Papyrus*. 2015. URL: https://eclipse.org/papyrus/ (besucht am 20. 05. 2016).

[4] Martin Auer, T. Tschurtschenthaler und Stefan X Biffl. *UMLet - Free UML Tool for Fast UML Diagrams*. 2016. URL: http://www.umlet.com/ (besucht am 20. 05. 2016).

[5] Norbert Bartelme. *Geoinformatik Modelle Strukturen Funktionen*. 4. Aufl. Springer, 1995. URL: http://link.springer.com/content/pdf/10.1007/b138747.pdf (besucht am 15. 10. 2015).

[6] Norbert Beckmann u. a. *The R*-tree: an efficient and robust access method for points and rectangles*. Bd. 19. 2. ACM, 1990. URL: http://dl.acm.org/citation.cfm?id=98741 (besucht am 05. 02. 2016).

[7] Jeff Bezanson u. a. „Julia: A Fast Dynamic Language for Technical Computing“. In: *CoRR* abs/1209.5145 (2012). URL: http://arxiv.org/abs/1209.5145.

[8] Iwo Białynicki-Birula und Iwona Białynicka-Birula. *Modeling reality: how computers mirror life*. Oxford; New York: Oxford University Press, 2004. ISBN: 978-0-19-853100-5.

[9] Ralf Bill. *Grundlagen der Geo-Informationssysteme*. 5. Aufl. Berlin: Wichmann, 2010.

[10] Bodhi Linux. *Welcome*. Aug. 2014. URL: http://www.bodhilinux.com/ (besucht am 19. 05. 2016).

[11] Folkmar Bornemann. *Numerische lineare Algebra*. 1. Aufl. Wiesbaden: Springer Fachmedien, 2016. ISBN: 978-3-658-12884-5.

[12] Brainwy Software Ltda. *PyDev*. 2016. URL: http://www.pydev.org/index.html (besucht am 23. 06. 2016).

[13] Georg Brandl. *Overview — Sphinx 1.4.1 documentation*. 2016. URL: http://www.sphinx-doc.org/en/stable/ (besucht am 23. 05. 2016).

[14] Thomas Brinkhoff. *Geodatenbanksysteme in Theorie und Praxis: Einführung in objektrelationale Geodatenbanken unter besonderer Berücksichtigung von Oracle Spatial*. 3. Aufl. Berlin: Wichmann/VDE-Verl., 2013. ISBN: 978-3-87907-513-3.

[15] Howard Butler. *Rtree: Spatial indexing for Python — Rtree 0.7.0 documentation*. 2011. URL: http://toblerity.org/rtree/ (besucht am 01. 06. 2016).

[16] Scott Chacon und Ben Straub. *Pro GIT - Everything you need to know about GIT*. 2. Aufl. APress, 2015. URL: http://git-scm.com/book/en/v2 (besucht am 01. 03. 2015).

[17] Martin Dietzfelbinger, Kurt Mehlhorn und Peter Sanders. *Algorithmen und Datenstrukturen*. eXamen.press. Berlin, Heidelberg: Springer, 2014. ISBN: 978-3-642-05471-6 978-3-642-05472-3. URL: http://link.springer.com/10.1007/978-3-642-05472-3 (besucht am 23. 02. 2016).

[18] Eclipse Foundation Inc. *Eclipse Downloads*. 2016. URL: https://www.eclipse.org/downloads/ (besucht am 20. 05. 2016).

[19] Patrick Ediger. „Modellierung und Techniken zur Optimierung von Multiagentensystemen in Zellularen Automaten“. Diss. Darmstadt: Technische Universität, Juni 2011. URL: http://tuprints.ulb.tu-darmstadt.de/2643/.

[20] Ahmed Eldawy. „SpatialHadoop: towards flexible and scalable spatial processing using mapreduce“. en. In: ACM Press, 2014, S. 46–50. ISBN: 978-1-4503-2924-8. DOI: 10.1145/2602622.2602625. URL: http://dl.acm.org/citation.cfm?doid=2602622.2602625 (besucht am 18. 04. 2016).

[21] Ramez A. Elmasri und Shamkant B. Navathe. *Fundamentals of database systems*. 5. Aufl. Boston: Pearson/Addison-Wesley, 2007. ISBN: 0-321-41506-X 978-0-321-41506-6.

[22] Ramez A. Elmasri, Shamkant B. Navathe und Angelika Shafir. *Grundlagen von Datenbanksystemen*. 3. Aufl. IT - Informatik. München: Pearson Studium, 2011. ISBN: 978-3-86894-012-1.

[23] EnterpriseDB Corporation. *Download PostgreSQL | EnterpriseDB*. Dez. 2010. URL: http://www.enterprisedb.com/products-services-training/pgdownload (besucht am 20. 05. 2016).

[24] Raphael A. Finkel und Jon Louis Bentley. „Quad trees a data structure for retrieval on composite keys“. In: *Acta informatica* 4.1 (1974), S. 1–9. URL: http://link.springer.com/article/10.1007/BF00288933 (besucht am 29. 02. 2016).

[25] Anthony Fox u. a. „Spatio-temporal indexing in non-relational distributed databases“. In: *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, S. 291–299. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6691586 (besucht am 15. 04. 2016).

[26] Alessandro Furieri. *SpatiaLite SQL functions reference list*. 2014. URL: https://www.gaia-gis.it/gaia-sins/spatialite-sql-4.4.0.html#p16 (besucht am 15. 06. 2016).

[27] Volker Gaede und Oliver Günther. *Multidimensional access methods*. Techn. Ber. 1998, S. 77. URL: http://mistral.in.tum.de/rwork/gg98.pdf (besucht am 05. 02. 2016).

[28] Volker Gaede und Oliver Günther. „Multidimensional access methods“. In: *ACM Computing Surveys (CSUR)* 30.2 (1998), S. 170–231. URL: http://dl.acm.org/citation.cfm?id=280279 (besucht am 05. 02. 2016).

[29] Fabio Grandi. „Temporal Databases“. In: *Encyclopedia of information science and technology*. Hrsg. von Mehdi Khosrow-Pour. 3. Aufl. Hershey, PA: Information Science Reference, 2015, S. 1914 –1922. ISBN: 9781466658882.

[30] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*. Bd. 14. 2. ACM, 1984. URL: http://dl.acm.org/citation.cfm?id=602266 (besucht am 05. 02. 2016).

[31] Oliver Günther. *Environmental information systems*. Berlin; New York: Springer, 1998. ISBN: 3540609261.

[32] George Brent Hall. „Open source approaches in spatial data handling“. In: *Advances in geographic information science*. Hrsg. von Michael G. Leahy. Berlin: Springer, 2008. ISBN: 978-3-540-74830-4.

[33] Dimitri van Heesch. *Doxygen: Main Page*. 2016. URL: http://www.stack.nl/~dimitri/doxygen/ (besucht am 23. 05. 2016).

[34] Dimitri van Heesch. *Doxygen Manual: Doxywizard usage*. 2016. URL: https://www.stack.nl/~dimitri/doxygen/manual/doxywizard_usage.html (besucht am 24. 06. 2016).

[35] Joseph M. Hellerstein. „Generalized Search Tree“. In: *Encyclopedia of database systems*. Hrsg. von Ling Liu und M. Tamer Özsu. Bd. 2. Springer reference. New York: Springer, 2009. ISBN: 9780387355443 9780387399409 9780387496160.

[36] Joseph M. Hellerstein, Jeffrey F. Naughton und Avi Pfeffer. „Generalized Search Trees for Database Systems“. In: *Readings in database systems*. Hrsg. von Michael Stonebraker und Joseph M. Hellerstein. 3. Aufl. The Morgan Kaufmann series in data management systems. San Francisco: Morgan Kaufmann, 1998. ISBN: 1558605231.

[37] Ekbert Hering, Rolf Martin und Martin Stohrer. *Taschenbuch der Mathematik und Physik*. ger. 5. Aufl. Berlin: Springer, 2009. ISBN: 978-3-540-78684-9 978-3-540-78683-2.

[38] Helmut Herold. *Grundlagen der Informatik*. München: Pearson, 2012. ISBN: 978-3-86894-111-1.

[39] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing In Science & Engineering* 9.3 (2007), S. 90–95.

[40] Theo Härder und Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Berlin: Springer, 2001. ISBN: 3-540-42133-5.

[41] Andrew Ilachinski. *Cellular automata: a discrete universe*. Singapore: World Scientific, 2002. ISBN: 981-238-183-X.

[42] Robert Johansson. *Numerical Python*. en. Berkeley, CA: Apress, 2015. ISBN: 978-1-4842-0554-9 978-1-4842-0553-2. URL: `http://link.springer.com/10.1007/978-1-4842-0553-2` (besucht am 10. 05. 2016).

[43] Søren Hauberg John W. Eaton David Bateman und Rik Wehbring. *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*. 2015. URL: `http://www.gnu.org/software/octave/doc/interpreter`.

[44] Hiroaki Kawai. *python-geohash*. 2014. URL: `https://github.com/hkwi/python-geohash` (besucht am 01. 06. 2016).

[45] Alfons Kemper und André Eickler. *Datenbanksysteme: eine Einführung*. 8. Aufl. München: Oldenbourg, 2011. ISBN: 978-3-486-72139-3.

[46] Lemont B. Kier, Paul G. Seybold und Chao-Kun Cheng. *Cellular automata modeling of chemical systems: a textbook and laboratory manual*. Dordrecht: Springer, 2005. ISBN: 978-1-4020-3657-6 978-1-4020-3690-3.

[47] Bernd Klein. *Einführung in Python 3: für Ein- und Umsteiger*. ger. 2. Aufl. München: Hanser, 2014. ISBN: 978-3-446-44151-4 978-3-446-44133-0.

[48] Daniel Klich. „cell.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[49] Daniel Klich. „cellular_automata_moquito“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[50] Daniel Klich. „cellular_automaton.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[51] Daniel Klich. „Entwicklung eines Software-Prototypen zur Modellierung des Ausbreitungsprozesses von Mückenarten“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[52] Daniel Klich. „layer.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[53] Daniel Klich. „parse_rule.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[54] Daniel Klich. „rule.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[55] Daniel Klich. „ruleset.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[56] Daniel Klich. „std_rule.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[57] Daniel Klich. „user_implemented_rule.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[58] Daniel Klich. „view.py“. Bachelorarbeit. Oldenburg: Carl von Ossietzky Universität Oldenburg, 2013.

[59] Daniel Klich u. a. „Entwicklung eines Software-Prototypen zur Modellierung des Ausbreitungsprozesses von Mückenarten“. In: *Modellierung und Simulation von Ökosystemen*. Hrsg. von Nguyen Xuan Thinh. Aachen: Shaker, 2013, S. 177 –190. ISBN: 978-3-8440-2275-9.

[60] Alfredo K. Kojima, Michael Vogt und Gustavo Niemeyer. *Synaptic Package Manager - Home*. 2016. URL: `http://www.nongnu.org/synaptic/` (besucht am 20. 05. 2016).

[61] Manolis Koubarakis, Hrsg. *Spatio-temporal databases: the CHOROCHRONOS approach*. Lecture notes in computer science 2520. Berlin; New York: Springer, 2003. ISBN: 978-3-540-40552-8.

[62] Wolfgang Kresse und David M. Danko, Hrsg. *Springer Handbook of Geographic Information*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-540-72678-4 978-3-540-72680-7. URL: `http://link.springer.com/10.1007/978-3-540-72680-7` (besucht am 23. 02. 2016).

[63] Bernd Kreußler und Gerhard Pfister. *Mathematik für Informatiker: Algebra, Analysis, diskrete Strukturen*. ger. eXamen.press. OCLC: 316271641. Berlin: Springer, 2009. ISBN: 978-3-540-89106-2 978-3-540-89107-9.

[64] Naba Kumar. *Anjuta DevStudio*. 2014. URL: `http://anjuta.org/` (besucht am 29. 06. 2016).

[65] Norbert de Lange. *Geoinformatik*. de. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-34806-8 978-3-642-34807-5. URL: `http://link.springer.com/10.1007/978-3-642-34807-5` (besucht am 23. 02. 2016).

[66] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. en. Texts in Computational Science and Engineering 6. DOI: 10.1007/978-3-642-30293-0_5. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-30292-3 978-3-642-30293-0. URL: `link.springer.com/content/pdf/10.1007%2F978-3-642-30293-0.pdf` (besucht am 14. 10. 2015).

[67] Hans Petter Langtangen. *Python scripting for computational science*. 3. Aufl. Texts in computational science and engineering 3. Berlin: Springer, 2009. ISBN: 978-3-540-73915-9 978-3-540-73916-6.

[68] Catherine Linard u. a. „A multi-agent simulation to assess the risk of malaria re-emergence in southern France“. In: *Ecological Modelling* 220.2 (2009), S. 160 –174. ISSN: 0304-3800. DOI: dx.doi.org/10.1016/j.ecolmodel.2008.09.001. URL: http://www.sciencedirect.com/science/article/pii/S0304380008004304.

[69] Peter Mandl. *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation*. 3. Aufl. Wiesbaden: Springer, 2013.

[70] Norman S. Matloff. *The art of R programming: tour of statistical software design*. OCLC: ocn711045702. San Francisco: No Starch Press, 2011. ISBN: 978-1-59327-384-2.

[71] Andreas Meier. *Relationale und postrelationale Datenbanken*. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-05255-2 978-3-642-05256-9. URL: http://link.springer.com/10.1007/978-3-642-05256-9 (besucht am 12. 04. 2016).

[72] MongoDB, Inc. *2d Index Internals - MongoDB Manual 3.2*. 2016. URL: https://docs.mongodb.org/manual/core/geospatial-indexes/#calculation-of-geohash-values-for-2d-indexes (besucht am 19. 04. 2016).

[73] Fabio Nelli. *Python Data Analytics*. Springer Berlin Heidelberg, 2015. ISBN: 978-1-4842-0959-2.

[74] Jürg Nievergelt, Hans Hinterberger und Kenneth C. Sevcik. „The grid file: An adaptable, symmetric multikey file structure“. In: *ACM Transactions on Database Systems (TODS)* 9.1 (1984), S. 38–71. URL: http://dl.acm.org/citation.cfm?id=318586 (besucht am 23. 02. 2016).

[75] Deborah Nolan und Duncan Temple Lang. *XML and Web Technologies for Data Sciences with R*. Use R! New York, NY: Springer, 2014. ISBN: 978-1-4614-7899-7 978-1-4614-7900-0. URL: http://link.springer.com/10.1007/978-1-4614-7900-0 (besucht am 19. 05. 2016).

[76] Regina O. Obe und Leo S. Hsu. *PostGIS in action*. 2. Aufl. Shelter Island, NY: Manning, 2015. ISBN: 9781617291395 1617291390.

[77] Obeo. *UML Designer Documentation*. 2016. URL: http://www.umldesigner.org/ (besucht am 20. 05. 2016).

[78] OGC. *Welcome to the OGC | OGC*. 2016. URL: http://www.opengeospatial.org/ (besucht am 23. 05. 2016).

[79] Oracle. *Oracle VM VirtualBox*. 2016. URL: https://www.virtualbox.org/ (besucht am 19. 04. 2016).

[80] Thomas Ottmann und Peter Widmayer. *Algorithmen und Datenstrukturen*. 5. Aufl. Heidelberg: Spektrum Akademischer Verlag, 2012.

[81] Norman H. Packard und Stephen Wolfram. „Two-dimensional cellular automata“. In: *Journal of Statistical physics* 38.5-6 (1985), S. 901–946. URL: http://link.springer.com/article/10.1007/BF01010423 (besucht am 09. 05. 2016).

[82] Thomas Pfeiffer und Andreas Wenk. *PostgreSQL: das Praxisbuch*. ger. 1. Aufl. Galileo computing. Bonn: Galileo-Press, 2010. ISBN: 978-3-8362-1346-2.

[83] pgAdmin Development-Team. *pgAdmin: PostgreSQL administration and management tools*. 2016. URL: https://www.pgadmin.org/ (besucht am 19. 09. 2016).

[84] PostgreSQL Wiki. *Python - PostgreSQL wiki*. 2016. URL: https://wiki.postgresql.org/wiki/Python (besucht am 18. 08. 2016).

[85] Lutz Prechelt. „An Empirical Comparison of Seven Programming Languages“. In: *Computer* 33.10 (2000), S. 23–29. ISSN: 0018-9162. DOI: 10.1109/2.876288. URL: http://dx.doi.org/10.1109/2.876288.

[86] PyPA. *get-pip.py*. 2016. URL: https://bootstrap.pypa.io/get-pip.py (besucht am 18. 08. 2016).

[87] PyPA. *Installation — pip 8.1.2 documentation*. 2014. URL: https://pip.pypa.io/en/stable/installing/#installing-with-get-pip-py (besucht am 18. 08. 2016).

[88] PyPA. *pip — pip 8.1.2 documentation*. 2014. URL: https://pip.pypa.io/en/stable/ (besucht am 18. 08. 2016).

[89] Python Software Foundation. *Download — Python 3.5.2rc1 documentation*. 2016. URL: https://docs.python.org/3.5/download.html (besucht am 20. 06. 2016).

[90] Python Software Foundation. *PostgreSQL - Python Wiki*. 2016. URL: https://wiki.python.org/moin/PostgreSQL (besucht am 18. 08. 2016).

[91] Python Software Foundation. *Pyqtree 0.24: Python Package Index*. 2015. URL: https://pypi.python.org/pypi/Pyqtree (besucht am 01. 06. 2016).

[92] Python Software Foundation. *Python Release Python 3.5.1*. 2016. URL: https://www.python.org/downloads/release/python-351/ (besucht am 18. 05. 2016).

[93] QGIS Development Team. *DB Manager Plugin*. 2016. URL: http://docs.qgis.org/testing/en/docs/user_manual/plugins/plugins_db_manager.html (besucht am 15. 06. 2016).

[94] QGIS Development Team. *Developing Python Plugins*. 2016. URL: http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/plugins.html (besucht am 08. 06. 2016).

[95] QGIS Development Team. *QGIS entdecken*. 2016. URL: http://qgis.org/de/site/about/index.html (besucht am 20. 05. 2016).

[96] Alfio Quarteroni, Fausto Saleri und Paola Gervasio. *Scientific computing with MATLAB and Octave*. eng. 4. Aufl. Texts in computational science and engineering v. 2. Heidelberg: Springer, 2014. ISBN: 9783642453663 364245366X.

[97] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2013. URL: http://www.R-project.org/.

[98] Paul Ramsey u. a. *PostGIS 2.2.1dev Manual*. URL: postgis.net/stuff/postgis-2.2.pdf (besucht am 15. 04. 2016).

[99] Paul Ramsey u. a. *PostGIS — Spatial and Geographic Objects for PostgreSQL*. 2016. URL: http://www.postgis.net/ (besucht am 10. 02. 2016).

[100] Peter Revesz. *Introduction to databases: from biological to spatio-temporal*. Texts in computer science. London: Springer-Verlag, 2010. ISBN: 9781849960946 1849960941 9781849960953 184996095X.

[101] Philippe Rigaux, Michel O. Scholl und Agnes Voisard. *Spatial databases: with application to GIS*. The Morgan Kaufmann series in data management systems. San Francisco: Morgan Kaufmann Publishers, 2002. ISBN: 1558605886.

[102] L. B. L. Santos u. a. „Periodic forcing in a three-level cellular automata model for a vector-transmitted disease“. en. In: *Physical Review E* 80.1 (Juli 2009). ISSN: 1539-3755, 1550-2376. DOI: 10.1103/PhysRevE.80.016102. URL: http://link.aps.org/doi/10.1103/PhysRevE.80.016102 (besucht am 12. 10. 2015).

[103] Helmut Saurer und Franz-Josef Behr. *Geographische Informationssysteme: eine Einführung*. Darmstadt: Wiss. Buchges., 1997. ISBN: 3-534-12009-4.

[104] Edwin Schicker. *Datenbanken und SQL: eine praxisorientierte Einführung mit Anwendungen in Oracle, SQL Server und MySQL*. ger. 4. Aufl. Informatik & Praxis. Wiesbaden: Springer Vieweg, 2014. ISBN: 978-3-8348-2185-0 978-3-8348-1732-7.

[105] Michael Schumacher. „Multi-agent systems“. In: *Objective Coordination in Multi-Agent System Engineering: Design and Implementation*. 2001, S. 9–32. URL: http://link.springer.com/chapter/10.1007/3-540-44933-7_2 (besucht am 10. 05. 2016).

[106] Richard T. Snodgrass. *Developing time-oriented database applications in SQL*. San Francisco, Calif: Morgan Kaufmann Publishers, 2000. ISBN: 1558604367.

[107] Ian Sommerville. *Software Engineering*. 9. Aufl. München: Pearson Studium, 2011. ISBN: 978-0-13-703515-1.

[108] Michael Sonnenschein und Ute Vogel. „Asymmetric cellular automata for the modelling of ecological systems“. eng ger. In: *Sustainability in the Information Society*. Hrsg. von Lorenz M. Hilty, International Symposium Informatics for Environmental Protection und Gesellschaft für Informatik. Umwelt-Informatik aktuell. Marburg: Metropolis-Verlag, 2001. ISBN: 978-3-89518-370-6.

[109] Michael Sonnenschein und Ute Vogel. „Hierarchical Asymmetric Cellular Automata for Multiple-scale Modelling of Ecological and Socio-economic Systems“. ger. In: *Environmental Communication in the Information Society - Proceedings of the 16th Conference*. Hrsg. von Werner Pillmann u. a. Vienna: ISEP, International Society for Environmental Protection, 2002. ISBN: 978-3-9500036-7-3.

[110] René Steiner. *Grundkurs Relationale Datenbanken*. de. Wiesbaden: Springer Fachmedien Wiesbaden, 2014. ISBN: 978-3-658-04286-8 978-3-658-04287-5. URL: http://link.springer.com/10.1007/978-3-658-04287-5 (besucht am 03. 03. 2016).

[111] Benjamin A. Stickler. *Basic concepts in computational physics*. Cham: Springer, 2014.

[112] The PostgreSQL Global Development Group. *PostgreSQL: Documentation*. 2016. URL: www.postgresql.org/files/documentation/pdf/9.5/postgresql-9.5-A4.pdf (besucht am 15. 04. 2016).

[113] The Umbrello Team. *Umbrello Project - Welcome to Umbrello - The UML Modeller*. 2016. URL: https://umbrello.kde.org/ (besucht am 25. 08. 2016).

[114] ubuntu Deutschland e. V. *Eclipse > Wiki > ubuntuusers.de*. 2016. URL: https://wiki.ubuntuusers.de/Eclipse/ (besucht am 20. 05. 2016).

[115] ubuntu Deutschland e. V. *Geographische Informationssysteme > Wiki > ubuntuusers.de*. 2016. URL: https://wiki.ubuntuusers.de/Geographische_Informationssysteme/ (besucht am 20. 05. 2016).

[116] ubuntu Deutschland e. V. *OpenJDK > Installation > Java > Wiki > ubuntuusers.de*. 2016. URL: https://wiki.ubuntuusers.de/Java/Installation/OpenJDK/ (besucht am 27. 06. 2016).

[117] Michael Unterstein und Günter Matthiessen. *Relationale Datenbanken und SQL in Theorie und Praxis*. ger. 5. Aufl. eXamen.press. Berlin: Springer Vieweg, 2012. ISBN: 978-3-642-28986-6 978-3-642-28985-9.

[118] Sophie O. Vanwambeke u. a. „Impact of Land-use Change on Dengue and Malaria in Northern Thailand“. en. In: *EcoHealth* 4.1 (März 2007), S. 37–51. ISSN: 1612-9202, 1612-9210. DOI: 10.1007/s10393-007-0085-5. URL: http://link.springer.com/10.1007/s10393-007-0085-5 (besucht am 03. 05. 2016).

[119] Daniele Varrazzo. *PostgreSQL + Python | Psycopg*. 2014. URL: http://initd.org/psycopg/ (besucht am 18. 08. 2016).

[120] John Wainwright und Mark Mulligan. „Modelling and Model Building“. In: *Environmental modelling: finding simplicity in complexity*. Hrsg. von John Wainwright und Mark Mulligan. Chichester, West Sussex, England; Hoboken, NJ: Wiley, 2004, S. 7 –73. ISBN: 978-0-471-49617-5 978-0-471-49618-2.

[121] Michael Weigend. *Objektorientierte Programmierung mit Python 3: Einstieg, Praxis, professionelle Anwendung*. 4. Aufl. Heidelberg: mitp-Verl, 2010. ISBN: 978-3-8266-1750-8.

[122] David Wheeler. *temporal_tables: Temporal Tables Extension / PostgreSQL Extension Network*. URL: `http://pgxn.org/dist/temporal_tables/` (besucht am 14. 04. 2016).

[123] Christine Wolfinger. *Keine Angst vor Linux/Unix*. Xpert.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-32078-1 978-3-642-32079-8. URL: `http://link.springer.com/10.1007/978-3-642-32079-8` (besucht am 30. 06. 2016).

[124] Simonas Šaltenis u. a. *Indexing the positions of continuously moving objects*. Bd. 29. 2. ACM, 2000. URL: `http://dl.acm.org/citation.cfm?id=335427` (besucht am 12. 04. 2016).

Glossar

Abstrakte Methode

Eine abstrakte Methode enthält nur die Signatur, jedoch nicht die eigentliche Implementierung [80, S. 251].

Abstrakter Datentyp

Ein Abstrakter Datentyp fasst eine Menge von Objekten und auf diesen definierten Operationen zusammen [80, S. 26].

Aktive Region

In aktiven Regionen befinden sich weibliche Mücken(larven). Daher müssen die korrespondierenden Zellen der aktiven Regionen in einem Simulationsschritt neu berechnet werden.

Alias

Ein Alias einer Variable referenziert in Python3 auf den gleichen Inhalt wie die andere Variable.

Alphanumerische Daten

Unter alphanumerischen Daten werden alle Daten zusammengefasst, die sich mittels einer Kombination von Buchstaben, Ziffern und Sonderzeichen ergeben.

B-Baum

Der B-Baum ist ein höhenbalancierter Suchbaum. Dessen Varianten B^+- bzw. B^*-Bäume in Datenbanken als Indexe für alphanumerische Daten eingesetzt werden.

Benutzer

Der Benutzer verwendet das in dieser Arbeit entwickelte Werkzeug.

Datenausgabe

Die Datenausgabe bezeichnet das Ausgeben von Daten, welche durch die Datenverarbeitung erzeugt worden sind.

Datenbank

Eine Datenbank ist ein System zur effizienten Speicherung und Verarbeitung von Daten. Der Benutzer muss die Datenbank unabhängig von ihrem inneren Aufbau nutzen können (auch wenn diese geändert wird). Der Nutzer darf nicht unerlaubt Daten verarbeiten oder beschädigen können. [110, S. 5]

Dateneingabe

Die Dateneingabe bezeichnet das Einlesen von Daten, welche für die weitere Verarbeitung benötigt werden.

Datenverarbeitung

Die Datenverarbeitung bezeichnet das Transformieren und Analysieren von Eingabedaten. Dabei werden die Daten verändert und/oder es werden neue Informationen erzeugt.

Dicht besetzter Zellularer Automat

Ein dicht besetzter Zellularer Automat zeichnet sich dadurch aus, dass ein überwiegender Teil der Zellen mit weiblichen Mücken(larven) besetzt ist.

Dünn besetzter Zellularer Automat

Ein dünn besetzter Zellularer Automat zeichnet sich dadurch aus, dass ein überwiegender Teil der Zellen frei von weiblichen Mücken(larven) ist.

Effizient

Effizient bedeutet in dieser Bachelorarbeit, dass möglichst wenig Rechenzeit benötigt wird.

Geografische Daten

Geografische Daten weisen einen räumlichen Bezug auf.

Geoinformationssystem

Ein Geoinformationssystem (GIS) dient dem Erfassen, Verarbeiten und Analysieren von Geografischen Daten bzw. Räumlichen Daten.

GiST

Der Generalized Search Tree (GiST) ist eine Verallgemeinerung von R- und B-Bäumen.

Grid File

Das Grid File ist ein auf Hashing basierter Index, welcher in räumlichen Datenbanken eingesetzt werden kann.

Immobile Agenten

Immobile Agenten können sich in einem Universum nicht selbstständig bewegen.

In-Memory-Datenbank

In-Memory-Datenbank speichern die Daten auf dem Hauptspeicher.

Index

Indexe sind eine interne Speicherstruktur und werden in Datenbanken für die effiziente Verarbeitung von Suchanfragen eingesetzt [117, S. 65 f.].

Initialisierung

Während der Initialisierung wird der Anfangszustand eines Systems gesetzt.

Integrationstest

Integrationstests dienen dem Testen des Zusammenspiels mehrerer Module.

MALCAM-Modell

Das MALCAM-Modell nach Linard et al. [68] wird in dieser Arbeit zur Beschreibung des Ausbreitungsverhaltens von Mücken genutzt.

Mobile Agenten

Mobile Agenten können sich in einem Universum selbstständig bewegen.

Mückencluster

Mückencluster sind in dieser Bachelorarbeit einzelne Regionen, in denen sich Mücken(larven) befinden. Zellulare Automaten, die einige Mückencluster enthalten, sollen nur in einigen wenigen Regionen Mücken(larven) enthalten und ansonsten keine.

Ohne großen Implementierungsaufwand

„Ohne großen Implementierungsaufwand“ bedeutet in dieser Arbeit, dass zur Erweiterung einer Funktionalität lediglich eine neue Klasse implementiert werden muss, welche bspw. von einem Interface oder einer (abstrakten) Klasse erbt. Keinesfalls darf hierbei der Simulationskern oder die übrige Datenhaltung verändert werden.

PostGIS

PostGIS ist eine räumliche Erweiterung der Datenbank PostgreSQL [76, S. 1].

PostgreSQL

PostgreSQL ist eine freie objektrelationale Datenbank unter der BSD-Lizenz [82, S. 14].

QGIS

QGIS ist ein freies Geoinformationssystem (GIS).

Quadtree

Der Quadtree ist ein nicht höhenbalancierter Suchbaum, der sich insbesondere für das abspeichern von Rasterdaten im Hauptspeicher eignet, da das Universum mittels zweier Hyperebenen in vier Unteruniversen partitioniert wird.

R-Baum

Der R-Baum ist ein höhenbalancierter Suchbaum, der als Index für räumliche Daten verwendet werdet werden kann. Das Einfügen und Löschen funktioniert ähnlich wie bei B^+- bzw. B^*-Bäumen.

Rasterdaten

Rasterdaten unterteilen das zweidimensionale Universum gitterartig in kleine Zellen.

Simulation

Eine Simulation analysiert komplexe Systeme.

Simulationsergebnis

Simulationsergebnis impliziert in dieser Arbeit, sofern nicht anders angegeben, auch den Startzustand.

Skripttest

Skripttests dienen in dieser Arbeit dem Testen von Teilfunktionalitäten des Werkzeugs.

Sparse Matrix

Eine Sparse Matrix speichert nur explizit alle Einträge ungleich Null. Dieses Vorgehen ist vorteilhaft, wenn fast alle Einträge großer Matrizen Null sind.

Spatiotemporale Datenbanken

Spatiotemporale Datenbanken speichern und verarbeiten Daten mit einem zeitlichen- und räumlichen Bezug.

Stapelspeicher

Ein Stapelspeicher ist eine Datenstruktur, welche nach dem LIFO-Prinzip (Last-In-First-Out) arbeitet.

System

Ein System ist ein aus verschiedenen Einzelteilen zusammengesetztes Ganzes, welches von der Umwelt isoliert ist, jedoch mit dieser interagiert. Dabei sind nur die Schnittstellen des Systems, nicht jedoch die eigentliche Zusammensetzung für die Umwelt sichtbar.

Temporale Datenbanken

Temporale Datenbanken speichern und verarbeiten Daten mit einem zeitlichen Bezug.

Testsuite

Eine Testsuite fasst mehrere Tests zusammen und ermöglicht ein gemeinsames Ausführen.

Unittest

Unittests dienen dem automatisierten Testen einzelner Module.

Universum

Ein Universum bezeichnet den zu modellierenden Raum, in dieser Bachelorarbeit ist der Raum immer zweidimensional.

Vektordaten

Vektordaten speichern räumliche Objekte als Punkte und die Verläufe von eindimensionalen- sowie die Umrisse von zweidimensionalen Objekten mit Vektoren.

Vollständige Ordnung

Eine Menge M heißt vollständig geordnet, wenn gilt $\forall x, y \in M : x \leq y \vee y \leq x$ [63, S. 390].

XML

Die eXtensible Markup Language (XML) ermöglicht es Daten standardisiert in einem menschen- und maschinenlesbaren Textfile abzuspeichern.

Zellularer Automat

Zellulare Automaten dienen der Simulation räumlich- und zeitlich diskretisierter Systeme, wobei der Zustand einer Zelle zum Zeitpunkt t von der Zelle zum Zeitpunkte $t-1$ sowie deren Nachbarzellen abhängt.

Index

Danksagung

Ich möchte Frau Dr. Ute Vogel für die Betreuung dieser Arbeit und der Stellung dieses interessanten Themas danken. Herrn Prof. Dr. Michael Sonnenschein danke ich für die Übernahme des Zweitgutachtens.

Kyra Bohnebeck danke ich für einige Tipps bei der Nutzung des MosquitoCA-Plugins.

Des Weiteren möchte ich mich bei René Wassermeier für das Durchsehen dieser Arbeit auf inhaltliche- und sprachliche Unstimmigkeiten bedanken.

Außerhalb der Universität danke ich meinen Eltern für ihre Unterstützung.

Daten-DVD

/
- benchmarks ..... Daten und Plots aus Kapitel 8.
  - db
  - fillingDegree_mit-Dynamik
  - fillingDegree_teil-Dynamik
  - size
- mosquito_bib ..... Literatur als bib- und als Zotero rdf-Datei.
  - files
- mosquito-simulation ..... Quellcode des Projekts.
  - doc
    - html
    - latex
  - run
  - src
  - test
  - tree
- pdf ..... pdf-Versionen der Bachelorarbeit, des Abschlussvortrags sowie des Forschungsseminarvortrags.
- sonstige_codes ..... Sammlung an verwendetem sonstigen Quellcode.
  - benchmarkplots
  - ca-test-plot
  - profile
- virtual_machine ..... Exportierte Version der virtuellen Maschine.

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Unterschrift