Modellierung von Ladungstransport in organischen Feldeffekttransistoren mittels eines kinetischen Monte-Carlo-Ansatzes

Dissertation

Von der Universität Bayreuth
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Tobias Meier

geboren in Regensburg, Deutschland

Bayreuth, 2023

Erster Gutachter: Prof. Dr. Anna Köhler

Zweiter Gutachter: Prof. Dr. Harald Oberhofer

Tag der Einreichung: 09.12.2022

Tag des Kolloquiums: 26.06.2023

Inhaltsverzeichnis

1. Kurzzusammenfassung

1.1. Deutsche Fassung

In dieser Arbeit untersuche ich den Einfluss der Morphologie und der Coulombwechselwirkung auf den Ladungstransport in organischen Feldeffekttransistoren mittels kinetischer Monte-Carlo-Simulation. Dabei gehe ich darauf ein wie sich Korngrenzen, die Anordnung von Oligomeren zueinander und die Coulombwechselwirkung auf den Ladungstransport und die Transfercharakteristiken auswirken. Für die Untersuchung habe ich ein Programm entwickelt und eine kommerzielle Software um Funktionen zur Berücksichtigung der Morphologie des Halbleiters erweitert. In polykristallinen organischen Feldeffekttransistoren habe ich drei Bereiche des Ladungstransports identifiziert, die durch die energetische Lage und die Breite der Korngrenzen bestimmt werden. Für energetisch flache Korngrenzen, dem wahrscheinlich häufigsten Fall, spielt die Lage des Ferminiveaus eine wichtige Rolle. In oligomeren Systemen habe ich herausgefunden, dass eine strenge Anordnung der Oligomere nicht nötig ist für eine hohe Ladungsträgermoblität, sondern eine Vorzugsrichtung der Oligomere mit einem beliebigen Versatz benachbarter Oligomere genügt. Optimal ist dabei ein Versatz benachbarter Oligomere, der genau der Hälfte der Kettenlänge entspricht. Weiterhin habe ich gezeigt, dass die Coulombwechselwirkung zu gekrümmten Transfercharakteristiken führen kann, die sogar ein Maximum aufweisen können. Dieser Effekt kann teilweise durch energetische Unordnung und Temperatur kompensiert werden.

1.2. English version

In this thesis, I investigate the influence of morphology and Coulomb interaction on charge transport in organic field-effect transistors using kinetic Monte Carlo simulation. I am investigating how grain boundaries, the arrangement of oligomers in relation to each other and the Coulomb interaction affect the charge transport and the transfer characteristics. For the investigation I have developed a program and extended a commercial software with functions to consider the morphology of the semiconductor. In polycrystalline organic field-effect transistors, I have identified three regions of charge transport that are determined by the energetic position and width of the grain boundaries. For energetically flat grain boundaries, probably the most common case, the position of the Fermi level plays an important role. In oligomeric systems, I have found that a strict arrangement of the oligomers is not necessary for a high charge-carrier mobility. Instead, a preferred direction of the oligomers with an arbitrary offset of neighbouring oligomers is sufficient. Optimal is an offset of neighbouring oligomers that corresponds to exactly half of the chain length. Furthermore, I have shown that the Coulomb interaction can lead to curved transfer characteristics, which can even show a maximum. This effect can be partially compensated by energetic disorder and temperature.

2. Einleitung

Die halbleitenden Eigenschaften von organischen Halbleitern wurden seit den 1960er Jahren mit wachsendem Interesse untersucht.[1,2] Die Gründe für das Forschungsinteresse an organischen Halbleitern sind vielerlei: Im Gegensatz zu anorganischen Halbleitern, die extreme Herstellungsbedingungen wie z.B. hohe Temperatur oder Hochvakuum, sowie Reinraumbedingungen verlangen, können organische Halbleiter bereits bei Raumtemperatur und normalem Umgebungsdruck auch ohne Reinräume hergestellt werden.[3] Aufgrund ihrer niedrigen Prozessierungstemperatur eignen sie sich für die Herstellung von dünnen Halbleiterschichten aus der Gasphase oder aus einer Lösung heraus.[4] Damit können auch Substrate, die keine hohe Prozessierungstemperatur vertragen, wie z.B. Plastik, verwendet werden, wodurch flexible Bauteile ermöglicht werden können.[5] Die Prozessierbarkeit aus der Lösung heraus macht sie zudem interessant für großflächige Druckherstellungsverfahren.[6] Des Weiteren sind eine einfache chemische Modifikationsmöglichkeit, oxidfreie Grenzschichten, gute Ionentransportfähigkeit und eine starke Kopplung von Anregungen an Moleküle verschiedene Gründe, warum sich organische Halbleiter besser für bioelektronische Anwendungen eignen als anorganischen Materialien.[7] Eine Anwendung von organischen Halbleitern, die dafür nötig ist, sind organische Feldeffekttransistoren (OFETs), die erstmalig 1986 entwickelt wurden.[8] Dabei handelt es sich um elektronische Bauteile, mit denen die Ladungsträgerdichte im Halbleiter genau gesteuert werden kann, wodurch sie sich auf hervorragende Art und Weise für Untersuchungen der Ladungstransporteigenschaften von organischen Halbleitern eignen.[3] Daneben bieten sie vielversprechende technische Anwendungsmöglichkeiten in verschiedenen Arten von Sensoren, vor allem chemische und biologische Sensoren.[9–11] Bemerkenswert sind dabei insbesondere organische elektrochemische Transistoren, bei denen ein Elektrolyt zwischen dem Ladungstransportkanal und der Gateelektrode statt einem konventionellen Dielektrikum verwendet wird.[12] Mit dieser Art von Transistor lassen sich unter anderem Aufbauten realisieren, mit denen sogar einzelne Proteinmoleküle detektiert werden können.[13] Daneben bieten OFETs aufgrund ihrer mechanisch flexiblen Eigenschaften die Möglichkeit, biegbare elektrophoretische oder organische Leuchtdioden-Displays herzustellen, in denen sie in der Aktivmatrix-Schicht eingesetzt werden.[14] Einige Smartphonehersteller bieten hierzu bereits faltbare Smartphones an.[15,16] Ein weiterer wichtiger Aspekt von organischen Halbleitern, die sie für die Forschung interessant machen, sind die Ladungstransporteigenschaften, die mit diesen Materialien erzielt werden kön-

nen. Eine wichtige Kenngröße für die Beschreibung der Ladungstransporteigenschaften ist dabei die Ladungsträgermobilität, die das Verhältnis zwischen der Driftgeschwindigkeit der Ladungsträger und dem angelegten elektrischen Feld angibt.[17] Organische Halbleiter können grob in zwei Kategorien aufgeteilt werden, „kleine“ Moleküle und Polymere.[3] „Kleine“ Moleküle – typischerweise immer noch Makromoleküle, aber keine Polymere – weisen häufig eine höhere Kristallinität auf, jedoch werden mittlerweile für kleine Moleküle und Polymere ähnliche Werte der Ladungsträgermobilität erreicht.[3] Die häufigsten in OFETs verwendeten kleinen Moleküle, die auf Löcherleitung basieren, sind Polyacene und Heteroacene und ihre Derivate.[3] Typische Vertreter sind beispielsweise Triisopropylsilylethynyl-Pentacen (TIPS Pentacen) oder Triethylsilylethynyl-Anthradithiophen (TES ADT), mit denen Ladungsträgermobilitäten größer als 1 $cm^2 V^{-1} s^{-1}$ erreicht werden.[18,19] Mit anderen Vertretern aus der Kategorie der Heteroacene werden Ladungsträgermobilitäten größer als 10 $cm^2 V^{-1} s^{-1}$ erreicht, beispielsweise für Derivate von [1]Benzothieno[3,2-b][1]benzothiophen (BTBT) und Dinaphtho[2,3-b:2' ,3'-f]thieno[3,2-b]thiophen (DNTT).[20–22] Für OFETs mit kristallinem Pentacen oder Rubren wurden sogar Ladungsträgermobilitäten über 15 $cm^2 V^{-1} s^{-1}$ erreicht.[23,24]

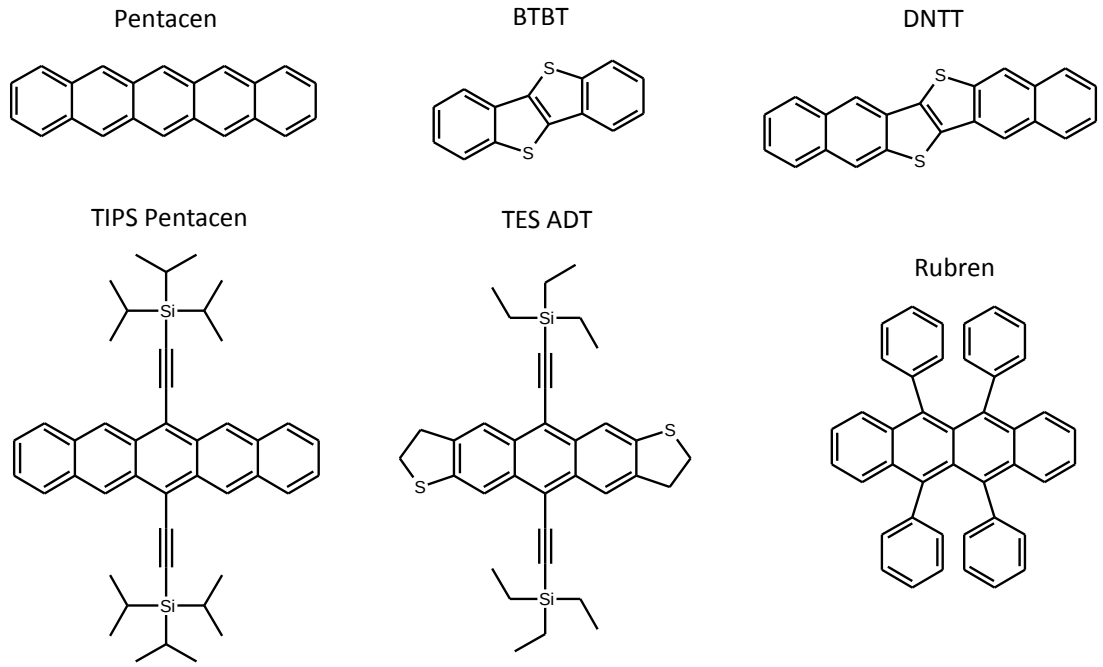

Abbildung 2.1 Eine Auswahl an kleinen Molekülen, die in OFET-Anwendungen hohe Mobilitäten erreichen.

Mit Polymeren können ebenfalls Ladungsträgermobilitäten über 20 $cm^2 V^{-1} s^{-1}$ erreicht werden.[25] Verglichen mit dem etablierten anorganischen Halbleiter Silizium liegt die Ladungsträgermobilität von organischen Halbleitern noch deutlich unterhalb der von kristallinem Silizium

(100 – 1000 $cm^2 V^{-1} s^{-1}$), übersteigt jedoch bereits die von amorphen Silizium und liegt in der Größenordnung von polykristallinem Silizium.[26,27] Dies macht sie interessant für technische Anwendungen.

Die Prozessierbarkeit von organischen Halbleitern aus der Lösung ermöglicht einfache Herstellungsmethoden wie Rotationsbeschichtung, Aufdampfverfahren, preisgünstiges Drucken oder Rolle-zu-Rolle-Verfahren.[28] Dabei hat die Herstellungsmethode einen großen Einfluss auf morphologische Parameter des Films wie die Korngröße, ihre Form und Orientierung.[28] Bei diesen Herstellungsmethoden entstehen Korngrenzen, die sich im Allgemeinen negativ auf den Ladungstransport auswirken.[29–34] Der genaue Mechanismus dahinter ist jedoch noch Gegenstand aktueller Forschung.[29,35–39] Ein weiterer Vorteil der vielfältigen chemischen Modifikationsmöglichkeiten von organischen Halbleitern ist, dass beispielsweise durch das Anpassen der Seitenkette der Moleküle die Orientierung der Moleküle zueinander im Film angepasst werden kann.[28] Dies ist wichtig, da in kristallinen molekularen Halbleitern neben der molekularen Struktur vor allem die Anordnung der Moleküle zueinander einen erheblichen Einfluss auf den Ladungstransport hat.[5,40]

Neben der prozessierungsbedingten morphologischen Vielfalt von organischen Halbleitern ist auch die Frage nach der dem Ladungstransport zugrunde liegenden Theorie noch Thema aktueller Forschung.[41,42] Die Ladungsträgermobilität, eine wichtige Kenngröße zur Bewertung der Ladungstransporteigenschaften von Materialien, wird experimentell zumeist aus Strom-Spannungskennlinien bestimmt.[9] Für eine genaue Bestimmung der Mobilität sind lineare Strom-Spannungskennlinien erforderlich. Häufig treten dabei jedoch Nichtlinearitäten auf, die zu einer Verfälschung der Angabe der Ladungsträgermobilität führen.[43] Ursachen dafür können Kontaktwiderstandseffekte, Mobilitäten, die von der Ladungsträgerdichte abhängen, oder nicht gleichmäßige Source-Drain-Spannung in OFETs mit kurzem Channel sein.[43] Wird ein einkristalliner OFET mit einem stark polarisierbarem Dielektrikum verwendet, so wurde zudem auch die Coulombwechselwirkung zwischen den Ladungsträgern im Transportkanal als eine Begründung für die beobachtete Sättigung der Stromdichte bei hoher Ladungsträgerdichte angeführt.[44] Diese Beobachtung wirft die Frage auf, inwieweit grundlegende physikalische Prinzipien wie die Coulombwechselwirkung Einfluss auf elektrische Charakteristiken in OFETs haben. Dass die Coulombwechselwirkung mit morphologischen Aspekten eines organischen Halbleiters zusammenwirken können, zeigt sich in Überlegungen, dass sich Ladungsträger beispielsweise in Korngrenzen ansammeln können und ihr Coulombpotential sich wiederrum auf den Ladungstransport auswirkt.[33,45,46]

Deswegen stellt sich somit insgesamt die Frage, welchen Einfluss die Morphologie des organischen Halbleiters und die Coulombwechselwirkung auf die Ladungstransporteigenschaften in organischen Feldeffekttransistoren haben.

In dieser Arbeit untersuche ich wie sich in polykristallinen OFETs und OFETs, die aus Oligomeren bestehen, einerseits morphologische Merkmale wie Korngrenzen und die Anordnung von Oligomeren auf den Ladungstransport, und andererseits die Coulombwechselwirkung auf die Transfercharakteristiken auswirken. Als Methode habe ich dafür Monte-Carlo-Simulationen gewählt, da sich damit das etablierte gaußsche Unordnungsmodell für Ladungstransport in organischen Halbleitern abbilden lässt.[47] Außerdem bietet ein Computerexperiment die Möglichkeit eines kontrollierten Systems, in dem sich gezielt der Einfluss einzelner Parameter auf das System untersuchen lässt.

Zu Beginn gebe ich nach einer kurzen Einführung in den Ladungstransport in OFETs einen Überblick über den aktuellen Stand in der Forschung zu den zuvor aufgekommenen Fragen, insbesondere im Hinblick auf bereits bestehende Monte-Carlo-Simulationen. Für meine Untersuchungen habe ich neben einem kommerziellen Programm eine eigenentwickelte Software verwendet, deren Methode ich in Kapitel 5 erläutern werde. Darüber hinaus habe ich Programmkomponenten entwickelt, die in das kommerzielle Programm eingebunden werden und die für die Untersuchungen zur Morphologie im organischen Halbleiter relevant sind. Diese werden ebenfalls in Kapitel 5 vorgestellt. In den darauffolgenden Kapiteln widme ich mich den eingangs gestellten Fragen und werde jeweils neben einer kurzen Motivation des spezifischen Teilaspekts und einer Erläuterung der verwendeten Methoden die Ergebnisse darstellen und in den Kontext der Literatur bringen. Dabei gehe ich in Kapitel 6 zunächst darauf ein, wie sich Korngrenzen in polykristallinen OFETs auf den Ladungstransport auswirken. Wesentliche Untersuchungsaspekte sind dabei die energetische Lage und die Breite der Korngrenzen, sowie die Größe der Kristallite. Als nächstes beleuchte ich in Kapitel 7 den Einfluss der Morphologie von Oligomeren in OFETs auf den Ladungstransport und gehe dabei auf die Länge der Oligomere und deren Ausrichtung zueinander, sowie einer möglichen vorhandenen energetischen Unordnung zwischen den Oligomeren ein. Anschließend untersuche ich in Kapitel 8 den expliziten Effekt der Coulombwechselwirkung auf die Transfercharakteristik von OFETs. Dabei verwende ich verschiedene Berechnungsweisen für die Coulombwechselwirkung, um deren Einfluss auf die Transfercharakteristiken zu untersuchen. Außerdem untersuche ich den Einfluss der Morphologie und der energetischen Unordnung des Halbleiters auf die Transfercharakteristik, sowie die Temperaturabhängigkeit der Ladungsträgermobilität in Abhängigkeit von der energetischen Unordnung. Abschließend gebe ich in Kapitel 9 eine kurze

Zusammenfassung der in dieser Dissertation gewonnenen Erkenntnisse. Mit dieser Arbeit werde ich zu einem vertieften Verständnis des grundlegenden Mechanismus von Ladungstransport und dem Einfluss wichtiger prozessierungsbedingter Merkmale in OFETs beitragen.

3. Einführung in den Ladungstransport in organischen Feldeffekttransistoren

3.1. Elektronische Struktur organischer Halbleiter

Klassische anorganische Halbleiter wie Silizium, Germanium und Galliumarsenid, zeichnen sich dadurch aus, dass sie ein vollständig gefülltes Valenzband und ein vollständig leeres Leitungsband mit einer geringen Bandlücke in der Größenordnung von wenigen eV aufweisen.[17,48] Die Atome in klassischen anorganischen Halbleitern werden dabei durch kovalente Bindungen zusammengehalten.[49] Im Gegensatz dazu liegen in organischen Halbleitern vergleichsweise schwache van-der-Waals-Bindungen zwischen den Molekülen vor.[49] Die in dieser Arbeit relevanten Klassen von organischen Halbleitern sind Molekülkristalle, amorphe molekulare Filme und Oligomere. Molekülkristalle zeichnen sich dadurch aus, dass sie wie klassische Festkörper ein Kristallgitter mit einer Einheitszelle aufweisen.[17] Typische Vertreter von Molekülkristallen sind Polyacene, Pyren und Perylen.[17] Ladungstransport in Molekülkristallen kann mittels Bandtransport erfolgen, da sie eine Bandstruktur mit einer Breite bis zu 500 meV aufweisen.[49,50] Im Gegensatz zu den Molekülkristallen liegt in amorphen organischen Halbleitern eine statistische Verteilung der Abstände zwischen den Molekülen vor. Dies führt zu einer räumlichen Variation der Polarisations- und van-der-Waals-Wechselwirkungen innerhalb des Halbleiters, wodurch sich die Energieniveaus der einzelnen Moleküle statistisch verteilen.[49] Aufgrund des zentralen Grenzwerttheorems führt dies zu einer gaußförmigen Verteilung der Energieniveaus der Moleküle, deren Standardabweichung σ charakteristisch für die energetische Unordnung des Halbleiters ist.[49,51] Die statistische Verteilung der Energien ist dabei in der Größenordnung von 0.1 eV und deutlich größer als die zugehörigen Transferintegrale, wodurch die Wellenfunktionen der Ladungsträger als stark lokalisiert betrachtet werden können.[42] Der Ladungstransport geschieht dann durch inkohärentes Hüpfen der Ladungsträger zwischen diesen lokalisierten Zuständen.[17] Oligomere schließlich bestehen aus einer Zusammensetzung von einzelnen chemischen Wiederholeinheiten, wobei ab einer Länge von etwa 100 Wiederholeinheiten von einem Polymer gesprochen wird.[17] Ladungstransport in Oligomeren geschieht kohärent innerhalb von konjugierten Segmenten und dazwischen mittels Hüpfprozess.[49]

3.2. Ladungstransport in organischen Halbleitern

3.2.1. Ladungsträgerdichte in unterschiedlichen Halbleiterbauteilen

Die Ladungsträgermobilität in organischen Halbleitern kann stark von der vorliegenden Ladungsträgerdichte abhängen.[52] Je nach Bauteil, in dem die Ladungsträgermobilität gemessen wird, unterscheidet sich die dabei auftretende Ladungsträgerdichte deutlich. Typische Anwendungen von organischen Halbleiterbauteilen sind beispielsweise OLEDs und OFETs.[17] Der prinzipielle Aufbau einer OLED und eines OFETs ist in Abbildung 3.1 dargestellt. Der Hauptunterschied zwischen diesen beiden Konfigurationen besteht darin, dass bei einem OFET zusätzlich ein elektrisches Feld senkrecht zur Source-Drain-Richtung angelegt wird, welches die Ladungsträger an der Grenzschicht zwischen Halbleiter und Dielektrikum konzentriert und somit die Ladungsträgerdichte in diesem Bereich erhöht.

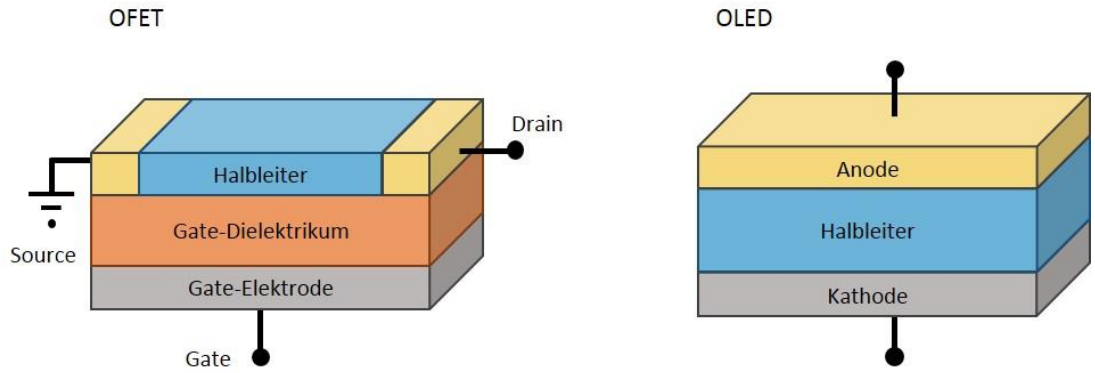

Abbildung 3.1 Schematische Zeichnung eines OFETs und einer OLED. Die relevanten elektrischen Anschlüsse sind dargestellt. Im Fall der OFET sind dies Source-, Drain- und Gateelektrode, im Fall der OLED Anode und Kathode.

Die Ladungsträgermobilität eines organischen Halbleiters kann sich um Größenordnungen unterscheiden, je nachdem ob sie in einem OFET oder einer OLED gemessen wird.[52] Beispiele von zwei organischen Halbleitern, deren Ladungsträgermobilität einmal in einem OFET und einmal in einer OLED gemessen wurde, sind in Abbildung 3.2 dargestellt.

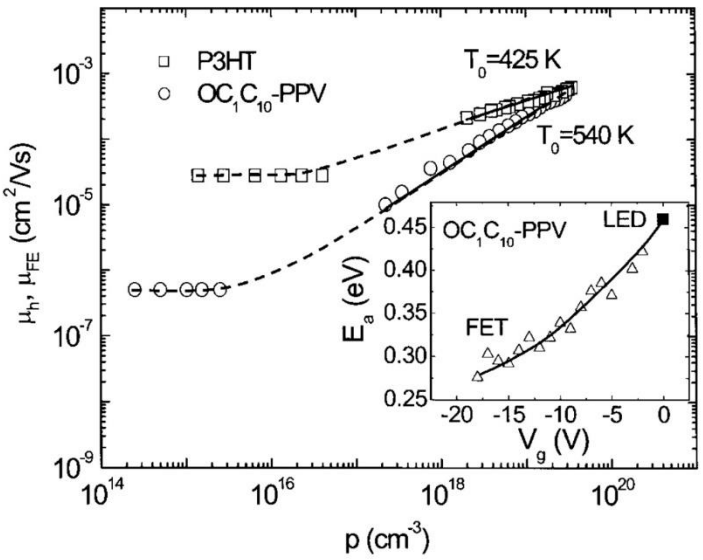

Abbildung 3.2 Ladungsträgermobilität als Funktion der Löcherdichte p gemessen in einer OLED- und einer OFET-Konfiguration für die beiden organischen Halbleiter P3HT und OC_1C_{10}-PPV. Die eingesetzte Abbildung zeigt die Aktivierungsenergie der Mobilität eines OFETs mit dem Halbleiter OC_1C_{10}-PPV als Funktion der Gatespannung (Dreiecke), sowie die Aktivierungsenergie aus der OLED-Konfiguration bei niedriger Ladungsträgerdichte (Rechteck). Genehmigter Abdruck.[52]

3.2.2. Ladungstransport in Bereichen unterschiedlicher Ladungsträgerdichte

Für die Anpassung der Mobilität in organischen Halbleiter wird häufig eine Arrheniusgleichung verwendet:[42]

$$\mu = \mu_\infty \exp\left(-\frac{E_A}{k_B T}\right) \tag{3.1}$$

Dabei ist μ_∞ der Wert der Mobilität, der sich bei unendlich hoher Temperatur ergeben würde und E_A die Aktivierungsenergie. Der funktionale Zusammenhang zwischen Ladungsträgermobilität und Temperatur hängt jedoch wesentlich von der Ladungsträgerdichte ab, wie im Abschnitt davor gezeigt. Wichtig für die Unterscheidung zwischen dem Fall hoher und niedriger Ladungsträgerdichte im organischen Halbleiter ist die thermische Gleichgewichtsenergie $\langle \varepsilon_\infty \rangle$. Sie beschreibt die Energie, die ein Ladungsträger im Mittel bei der Bewegung durch die Zustandsdichte eines organischen Halbleiters bei einer bestimmten Temperatur T einnimmt und ist für eine gaußförmige Zustandsdichte mit einer Standardabweichung der Verteilung σ gegeben durch:[47]

$$\langle \varepsilon_\infty \rangle = -\frac{\sigma^2}{k_B T} \tag{3.2}$$

Je nachdem, wo das Ferminiveau ε_F, das heißt die Energie, bis zu der die Zustände aufgefüllt sind, im Vergleich zur thermischen Gleichgewichtsenergie liegt, liegt eine hohe bzw. niedrige Ladungsträgerdichte im organischen Halbleiter vor. In Abbildung 3.3 sind beide Bereiche dargestellt.

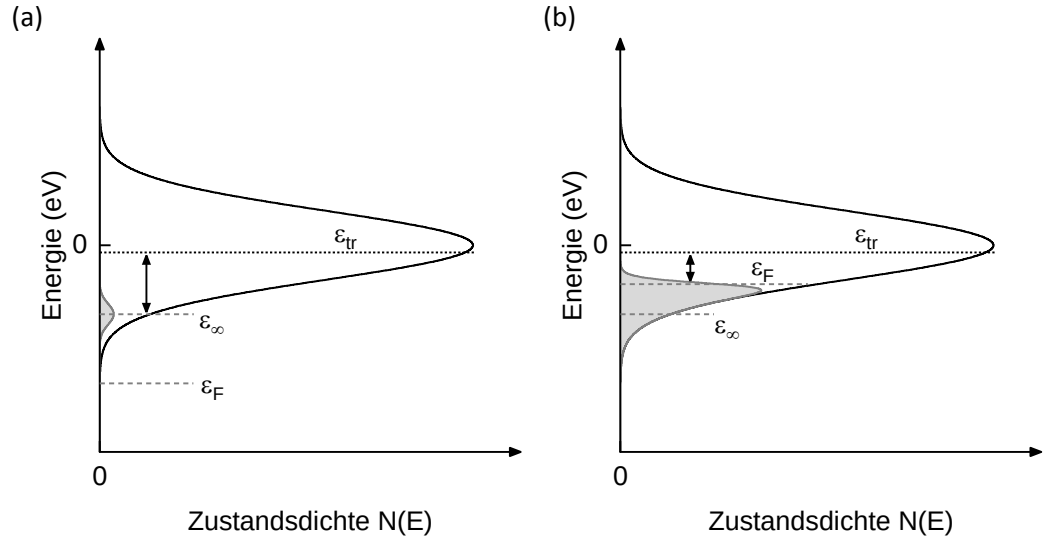

Abbildung 3.3 Schematische Darstellung der Zustandsdichte für den Fall (a) niedriger und (b) hoher Ladungsträgerdichte. Das Ferminiveau ε_F, die thermische Gleichgewichtsenergie ε_∞ und die Transportenergie ε_{tr} sind als gestrichelte Linien eingezeichnet. Die ausgefüllten Kurven stellen die besetzte Zustandsdichte dar.

Der Bereich hoher Ladungsträgerdichte liegt vor, wenn das Ferminiveau oberhalb der thermischen Gleichgewichtsenergie liegt.[42] Eine wichtige Größe für den Ladungstransport ist die Transportenergie ε_{tr}. Sie beschreibt die Energie, die ein Ladungsträger thermisch aktiviert erreichen muss, um sich durch den organischen Halbleiter bewegen zu können und nicht auf seinen Ausgangspunkt zurückfallen zu müssen.[17] Bei hoher Ladungsträgerdichte nimmt die Mobilität als Funktion der Temperatur die Form einer Arrheniusgleichung an und es gilt:[42,53,54]

$$\mu(T) \cong \mu' \exp\left(-\frac{\Delta}{k_B T}\right) \tag{3.3}$$

Dabei ist $\Delta = \varepsilon_{tr} - \varepsilon_F$ die Differenz zwischen der Transportenergie und dem Ferminiveau. Da das Ferminiveau von der Ladungsträgerdichte abhängt, ist auch die Mobilität abhängig von der Ladungsträgerdichte.[42]

Eine niedrige Ladungsträgerdichte liegt vor, wenn das Ferminiveau unterhalb der thermischen Gleichgewichtsenergie liegt. Da sich die Ladungsträger im thermischen Gleichgewicht in der Nähe der thermischen Gleichgewichtsenergie befinden, welche temperaturabhängig ist, müssen sie von dieser Energie aus thermisch aktiviert auf die Transportenergie angehoben werden. Daraus ergibt sich für die Ladungsträgermobilität als Funktion der Temperatur:[42,47]

$$\mu(T) \propto \exp\left(-\left(\frac{C\sigma}{k_B T}\right)^2\right) \tag{3.4}$$

Dabei ist $C \approx \frac{2}{3}$. In diesem Bereich ist die Mobilität unabhängig von der Ladungsträgerkonzentration.[42] Die Temperaturabhängigkeit der Mobilität unterscheidet sich somit in den beiden Bereichen hoher und niedriger Ladungsträgerdichte.

3.3. Funktionsweise eines Feldeffekttransistors

Ein OFET ist ein elektronisches Bauteil mit drei Anschlüssen, eine Source-, Drain- und Gateelektrode. Diese Anschlüsse können auf verschiedene Weise im Bauteil angeordnet sein, wie in Abbildung 3.4 gezeigt ist.[3] Die bottom gate/bottom contact-Struktur wird dabei häufig verwendet und ist auch kommerziell erhältlich.[17]

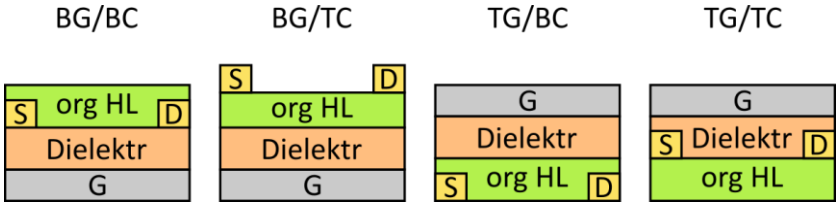

Abbildung 3.4 Schemadarstellung verschiedener OFET-Strukturen. S, D und G bezeichnen die Source-, Drain- und Gateelektrode, org HL den organischen Halbleiter und Dielektr das Gatedielektrikum. Von links nach rechts: bottom gate/bottom contact, bottom gate/top contact, top gate/bottom contact, top gate/top contact.

Die Funktionsweise eines OFETs basiert dabei auf folgendem Mechanismus:[17] Das System aus Gateelektrode, Gatedielektrikum und organischer Halbleiter kann als Kondensator betrachtet werden, wobei der organische Halbleiter die Gegenelektrode zur Gateelektrode darstellt. Beim Anlegen einer Spannung U_G an die Gateelektrode werden Ladungsträger aus der Sourceelektrode in den Halbleiter gezogen und sammeln sich in einer dünnen Schicht in der Nähe zum Dielektrikum an. Die Sourceelektrode befindet sich dabei auf dem Potential null und

wird als ohmisch angenommen, sodass beliebig viele Ladungsträger injeziert werden können. Schematisch ist dies in Abbildung 3.5 dargestellt.

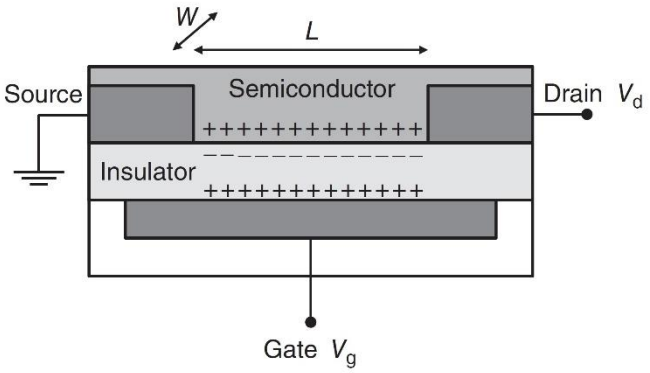

Abbildung 3.5 Prinzip der Funktionsweise eines OFETs mit angelegten Spannungen an die Gate- und Drainelektrode. Genehmigter Abdruck.[17]

Die durch die angelegte Gatespannung injezierten Ladungsträger stehen nun dem Ladungstransport zwischen Source- und Drainelektrode zur Verfügung. Dabei muss berücksichtigt werden, dass typischerweise ein gewisser Schwellwert U_{th} der Gatespannung überschritten werden muss, um eventuelle Fallenzustände im Halbleiter mit Ladungsträgern aufzufüllen, bevor ein Ladungstransport möglich ist.[17] Wird zusätzlich eine Spannung U_{SD} an die Drainelektrode angelegt, so bewegen sich die Ladungsträger anschließend in Richtung des dadurch entstehenden Source-Drain-Feldes. Die Strecke im Halbleiter von Source- zu Drainelektrode wird dabei als Kanallänge L bezeichnet, die Ausdehnung der Source- und Drainelektrode in der Richtung senkrecht dazu als Kanalbreite W. Je nach Stärke der Spannung U_{SD} werden verschiedene Arbeitsbereiche des Transistors klassifiziert: Der lineare Bereich gilt für $U_{SD} \ll U_G - U_{th}$ und der Sättigungsbereich für $U_{SD} > U_G - U_{th}$.[17] Der Bereich dazwischen ist ein Übergangsbereich. Die Unterscheidung in die verschiedenen Arbeitsbereiche kommt dadurch zustande, dass die Ladungsträgerdichte, die für niedrige Source-Drain Spannungen an jeder Stelle in x-Richtung gleich ist, durch höhere Source-Drain-Felder modifiziert wird. Dabei fließen die Ladungsträger an der Drain-Elektrode aus dem Halbleiter hinaus, wodurch die Ladungsträgerdichte zu dieser Elektrode hin abfällt. Bei der Spannung $U_{SD} = U_G - U_{th}$ fällt sie an der Stelle der Drain-Elektrode vollständig auf null ab. Dies markiert den Übergang zwischen den Arbeitsbereichen.

Die Ladungsträgermobilität bzw. die Beweglichkeit der Ladungsträger ist der Proportionalitätsfaktor zwischen der mittleren Driftgeschwindigkeit v der Ladungsträger im Halbleiter unter dem Einfluss eines elektrischen Feldes F und ebendiesem Feld, gemäß $v =$

μF.[17] Sie ist mit der Stromdichte j über $j = \rho v$ verknüpft, wobei ρ die Ladungsträgerdichte ist. Daraus ergibt sich die Ladungsträgermobilität zu:

$$\mu = \frac{j}{\rho F} \tag{3.5}$$

Die Ladungsträgerdichte der beweglichen Ladungsträger bestimmt sich aus der Gatespannung. Durch die Betrachtung des Systems Gateelektrode, Dielektrikum und organischer Halbleiter als Kondensator ergibt sich für die Flächenladungsdichte n:[55]

$$n = CU_G = \frac{\varepsilon_0 \varepsilon_G}{d_G} U_G \tag{3.6}$$

wobei C die Kapazität des Dielektrikums pro Einheitsfläche, U_G die Gatespannung, ε_0 die elektrische Feldkonstante, ε_G die Dielektrizitätskonstante des Gate-Dielektrikums und d_G die Dicke des Gate-Dielektrikums ist. Ein typisches Material für das Gatedielektrikum ist Siliziumdioxid, welches eine Dielektrizitätskonstante von 3.7 – 3.9 hat.[56] Das Gatedielektrikum hat dabei häufig eine Dicke in der Größenordnung von einigen 100 nm und die Kanallänge ist in der Größenordnung von einigen 100 µm.[25,57,58] Bei einer Gatespannung von 10 V ergibt sich bei einer Dicke des Gatedielektrikums von 100 nm und einer Dielektrizitätskonstante von 4 eine Flächenladungsdichte von 0.022 e_0 nm^{-2}, wobei e_0 die Elementarladung ist.

3.4. Experimentelle Bestimmung der Ladungsträgermobilität

Experimentell wird die Ladungsträgermobilität aus der Strom-Spannungskennlinie bestimmt. Dazu wird die Source-Drain-Stromstärke I_{SD} als Funktion der Gatespannung U_G gemessen. Die Ladungsträgermobilität bestimmt sich aus der Steigung von I_{SD} bzw. $\sqrt{|I_{SD}|}$ wie folgt:[17]

$$\mu_{lin} = \frac{L}{WC_i U_{SD}} \left(\frac{\partial I_{SD}}{\partial U_G} \right) \tag{3.7}$$

$$\mu_{sat} = \frac{2L}{WC_i} \left(\frac{\partial \sqrt{|I_{SD}|}}{\partial U_G} \right)^2 \tag{3.8}$$

wobei μ_{lin} die Mobilität im linearen Bereich und μ_{sat} im Sättigungsbereich darstellt. L ist dabei die Länge des Transistorkanals und W dessen Breite. Die Gleichungen ergeben sich aus der Annahme, dass das elektrische Feld in Source-Drain-Richtung viel kleiner ist als das, welches

durch die Gate-Elektrode verursacht wird. Das ist die sogenannte „gradual channel approximation“.[17]

Um mit diesem Modell aussagekräftige Werte für die Ladungsträgermobilität zu bekommen, ist es wichtig, dass die Strom-Spannungskurven $I_{SD}(U_G)$ im linearen Bereich und $\sqrt{|I_{SD}|}(U_G)$ im Sättigungsbereich in einem großen Bereich der Gatespannung linear sind.[43] Andernfalls können unrealistische Werte entstehen, die durch mögliche Fehlerquellen wie z.B. der Kontaktwiderstand, eine Abhängigkeit der Mobilität von der Ladungsträgerdichte oder Hystereseffekte hervorgerufen werden können.[43]

4. Stand der Forschung zu Ladungstransportsimulationen mittels kinetischer Monte-Carlo-Simulation in OFET-Strukturen

Eine Monte-Carlo-Simulation ist eine Methode, die auf der Verwendung von Zufallszahlen zur Beschreibung des zeitlichen Verlaufs eines Systems basiert.[59] Für die Übergangswahrscheinlichkeiten zwischen zwei Zuständen des Systems kommt der sogenannte Metropolis-Algorithmus zum Einsatz, der von der Differenz der Energien zwischen Anfangs- und Endzustand des Systems abhängt.[60] Handelt es sich um Simulationen, bei denen neben der Energie des Anfangs- und Endzustands des Systems auch eine eventuelle energetische Barriere zwischen den Zuständen eine Rolle spielen kann, so wird dies als kinetische Monte-Carlo-Simulation bezeichnet.[59]

Zur Untersuchung von Ladungstransport in organischen Halbleiterbauteilen wurden in der Vergangenheit erfolgreich Monte-Carlo-Simulationen verwendet.[47,61–67] Aufgrund der starken Lokalisierung von Orbitalen in molekularen Materialien sind solche Simulationen ein geeignetes Mittel, um Ladungstransport in diesen Materialien zu beschreiben, da der Ladungstransport dabei als eine Abfolge von Hüpfprozessen der Ladungsträger zwischen diesen lokalisierten Zuständen beschrieben werden kann.[68,69]

Neben Monte-Carlo-Simulationen gibt es noch weitere Möglichkeiten zur Simulation von Ladungstransport in organischen Halbleitern, beispielsweise mittels eines Master-Equation-Ansatzes.[70] Dabei wird die Coulombwechselwirkung nur über ein mittleres Feld berücksichtigt, jedoch kann die lokale Energiedifferenz zwischen zwei Gitterpunkten deutlich größer als die intrinsische energetische Unordnung sein.[61] Dazu gibt es bereits Erweiterungen, jedoch kann eine Korrelation zwischen den Ladungsträgern, die sich, wenn auch gering, auf den Stromfluss auswirken kann,[71] nicht berücksichtigt werden.[61]

Im Folgenden stelle ich bisherige wichtige Ergebnisse zu Ladungstransportsimulationen dar, beschränke mich dabei jedoch auf solche, die mittels Monte-Carlo-Simulation erzielt wurden,

da dies die von mir verwendete Methode ist und somit eine bessere Vergleichbarkeit erzielt werden kann.

4.1. Berücksichtigung der Coulombwechselwirkung in der Simulation

Die Coulombwechselwirkung zwischen den Ladungsträgern in elektronischen Bauteilen kann einen Einfluss auf die Ladungsträgermobilität, eine relevante Kenngröße für die Effizienz elektronischer Bauteile, haben.[72,73] Deswegen ist eine genaue Beschreibung der Coulombwechselwirkung innerhalb von Ladungstransportsimulationen wichtig.

Für die Berechnung der Coulombwechselwirkung in organischen Halbleiterbauteilen gibt es unterschiedliche Ansätze. Für Dioden wurde beispielsweise die eindimensionale Poisson-Gleichung verwendet, um das Coulombpotential zu berechnen.[74,75] Dabei wird die Ladungsträgerdichte gemittelt in einzelnen Schichten verwendet. Bei dieser Herangehensweise wird jedoch der kurzreichweitige Anteil der Coulombwechselwirkung zwischen den Ladungsträgern vernachlässigt.[76] Eine Erweiterung dazu ist die exakte Berechnung der Coulombwechselwirkung innerhalb einer Kugel mit einem Cut-off-Radius, welche in der Gruppe um die Forscher Coehoorn und Bobbert entwickelt wurde.[77] In OFET-Simulationen, die die Ladungsträgerinjektion und -extraktion an der Source- und Drain-Elektrode mit einschließen, ist mindestens die Verwendung der zweidimensionalen Poisson-Gleichung notwendig.[76] Die Gruppe um Li und Brédas hat ein solches OFET-Modell entwickelt. Sie können damit den Strom-Spannungsverlauf im linearen und Sättigungsbereich eines OFETs wiedergeben.[63] Für die Richtung senkrecht zur Source-Drain-Richtung in der Ebene des Gatesubstrats nehmen sie dabei ein konstantes Potential an. Bei der Entwicklung ihres Modells haben sie eine mögliche Fehlerquelle identifiziert, die durch eine falsche Berücksichtigung des Coulombpotentials des hüpfenden Ladungsträgers entstehen kann.[78] Sie schlagen dafür vor, entweder das Coulombpotential in der Anfangs- und Endkonfiguration der Ladungsträger zu berechnen oder den hüpfenden Ladungsträger bei der Berechnung des Coulombpotentials auszuschließen. Bei der Verwendung eines Cut-off-Radius für die Coulombwechselwirkung wird die Summe der Coulombpotentiale der einzelnen Ladungsträger an einem bestimmten Punkt abgeschnitten. Um das Problem der schlechten Konvergenz dieser Summe zu umgehen, haben Pippig und Mercuri einen gemeinsamen Cut-off-Radius zwischen dem Anfangs- und Endgitterpunkt des hüpfenden Ladungsträgers eingeführt.[68] Da in die Berechnung

der Hüpfraten nur die Differenz des Coulombpotentials zwischen zwei Gitterpunkten eingeht, können sie dadurch eine deutliche Verbesserung der Konvergenz der durch den Cut-off-Radius abgeschnittenen Summe erzielen. Die in dieser Dissertation verwendeten Ansätze sind in Kapitel 5 vorgestellt.

4.2. Einfluss der Konjugation und Oligomere auf den Ladungstransport

Organische Halbleiterbauteile, die auf polymeren Filmen basieren, zeigen ein enormes technologisches Potential aufgrund ihrer guten Ladungstransporteigenschaften.[40] Der Ladungstransport innerhalb der Polymerketten spielt dabei eine wichtige Rolle,[79,80] aber auch die Ausrichtung der Polymere entlang der Ladungstransportrichtung,[57,81,82] sowie die Anordnung von Molekülen zueinander.[83] Beispielsweise ermöglichen nematisch ausgerichtete Ketten in einem Flüssigkristall eine höhere Mobilität in Richtung der Ausrichtung, sowie gemittelt über alle Richtungen.[84] Die Konjugationslänge kann zudem auch einen Einfluss auf die Ladungsträgermobilität haben.[27]

Die Morphologie des organischen Halbleiters wurde in Monte-Carlo-Simulationen bereits durch verschiedene Ansätze berücksichtigt. Ein einfacher Ansatz dabei ist zum Beispiel die Verwendung von anisotropen Kopplungskonstanten für die Sprünge der Ladungsträger.[85] Li und Brédas haben damit einen OFET mit Source- und Drainelektrode modelliert und dabei festgestellt, dass bei anisotropen Kopplungskonstanten mit einem Verhältnis zwischen den Sprüngen in Richtung des Source-Drain-Feldes und denen senkrecht dazu größer als 500 und der Verwendung einer bottom-gate-top-contact-Struktur des OFETs nichtlineare Stromspannungskurven auftreten können.[85] Die Anisotropie der Kopplungskonstanten kann dabei als morphologisches Merkmal von beispielsweise organischen Kristallen oder stark ausgerichteten Polymeren aufgefasst werden. Eine fortgeschrittene Variante für die Berücksichtigung von polymeren Morphologien in einer Simulation ist die Erzeugung von wurmähnlichen Polymerketten auf einem kubischen Punktgitter. Durch Monte-Carlo-Simulation mit einer solchen Morphologie des Halbleiters konnte gezeigt werden, dass der Ladungstransport auf kurzer Zeitskala durch eine hohe Mobilität innerhalb der Ketten und auf langer Zeitskala durch eine niedrige Mobilität zwischen den Ketten charakterisiert wird.[80] Mendels und Tessler verwenden einen ähnlichen angepassten Algorithmus von gleitenden Schlangen zur Erzeugung von geschlängelten Polymerketten und stel-

len dabei fest, dass eine Zunahme des Verhältnisses der Transferintegrale der Sprünge innerhalb und zwischen den Polymerketten eine Steigerung der Mobilität bewirkt.[86–89] Der Verstärkungseffekt ist dabei deutlicher ausgeprägt bei kleinerer energetischer Unordnung. Außerdem beobachten sie eine Zunahme der Mobilität mit zunehmender Länge der Ketten in Feldrichtung, genauer mit zunehmender Ausrichtung der Ketten in Feldrichtung, bei insgesamt konstanter Kettenlänge. Bei konstanter Länge der Ketten in Feldrichtung sinkt jedoch die Mobilität leicht mit zunehmender Gesamtkettenlänge.

4.3. Einfluss von Fallen und Korngrenzen auf den Ladungstransport

In polykristallinen organischen Halbleitern können bedingt durch die Prozessierung Korngrenzen auftreten. Diese wirken sich im Allgemeinen negativ auf den Ladungstransport aus.[29–34] Ein genaues Verständnis des Einflusses von Korngrenzen auf den Ladungstransport ist jedoch zum jetzigen Zeitpunkt noch fehlend.

Untersuchungen zum Einfluss von Korngrenzen auf den Ladungstransport mittels Monte-Carlo-Simulation wurden von Vladimirov et al. durchgeführt.[38] Sie simulieren die Bewegung eines einzelnen Ladungsträgers in einer Monoschicht, für die sie die Energielandschaft mittels elektronischer Strukturberechnung ermitteln und finden dabei heraus, dass in ihrem System Korngrenzen, die als Barrieren wirken, den Ladungstransport maßgeblich beeinflussen. Eine weitere Simulationsmethode für polykristalline Morphologien wurde von der Gruppe um Mohan angewandt.[37,90,91] Sie nehmen in ihrem Modell an, dass sich innerhalb der Ladungstransportschicht quaderförmige, kristalline Bereiche befinden, die eine niedrigere energetische Unordnung als die umliegende amorphe Matrix besitzt. Mit der Zunahme des Anteils der kristallinen Bereiche beobachten sie eine Zunahme der Ladungsträgermobilität.[91] Mittels atomistischer Molekulardynamik und Monte-Carlo-Simulation berechnen Steiner et al., dass sich in TIPS-Pentacen energetische Barrieren zwischen Kristalliten befinden, die die Ladungsträgermobilität reduzieren.[92] Der Effekt ist schwach abhängig vom Winkel zwischen der Ausrichtung von Kristalliten und nimmt zu, wenn die Kontaktfläche zwischen zwei Kristalliten kleiner wird. Im Allgemeinen können jedoch auch Fallenzustände und Kombinationen aus Fallen und Barrieren in den Korngrenzen auftreten.

Neben der expliziten Untersuchung von Fallen- und Barrierenzuständen in polykristallinen Systemen mittels Monte-Carlo-Simulation gibt es noch weitere Untersuchungen mit Fallen- und Barrierenzuständen, die gleichmäßig im Halbleiter verteilt sind. So wurde festgestellt, dass niedrige Barrieren und flache Fallen die Mobilität stark verringern, wohingegen tiefe Fallen und hohe Barrieren kaum einen Einfluss auf die Mobilität haben.[93,94] Neben der energetischen Höhe der Fallen- oder Barrierenzustände hat auch deren Konzentration und Anordnung einen Einfluss auf die Ladungsträgermobilität. Mit steigender Konzentration nimmt die Mobilität zunächst ab und wächst wieder an, wenn die Konzentration wieder gegen 100 % ansteigt.[95] Bilden die Fallen- oder Barrierenzustände bei gleicher Konzentration zusammenhängende Bereiche, so ist die Mobilität dabei größer als bei homogen verteilten Fallen- oder Barrierenzuständen.[95]

5. Entwicklung der verwendeten theoretischen Methoden

5.1. Einteilchensimulationen mit effektiver Energielandschaft

Zur Modellierung des Ladungstransports in einem organischen Feldeffekttransistor habe ich zu Beginn meiner Promotion ein Programm zur Monte-Carlo-Simulation selbst geschrieben. Das Ziel dabei war einen verlässlichen Programmcode zu entwickeln, der die Morphologie des organischen Halbleiters im Rahmen eines Punktgittermodells berücksichtigen kann bei gleichzeitig vertretbarem Rechenaufwand. Die Schwierigkeit hierbei besteht darin, dass für die Abbildung von Strukturen wie zum Beispiel Oligomeren oder Polykristalliten in der Simulation eine ausreichend große Simulationsbox erforderlich ist. Mit größerer Simulationsbox steigt jedoch der Rechenaufwand. Dies liegt daran, dass mit größer werdender Simulationsbox eine größere Anzahl an Ladungsträgern bei gleicher Ladungsträgerdichte einhergeht. Der Rechenaufwand steigt dabei in etwa quadratisch mit der Anzahl der Ladungsträger N an, da die Anzahl der Wechselwirkungen zwischen N Teilchen mit $\sum_{k=0}^{N-1} k = \frac{1}{2}(N^2 - N)$ steigt.

Die einfachste denkbare Variante einer Ladungstransportsimulation ist eine Einteilchensimulation, bei der sich ein einzelner Ladungsträger zufällig durch die Energielandschaft eines organischen Halbleiters unter dem Einfluss von externen Feldern bewegt. Dies ist ein geeignetes Mittel für die Simulation von beispielsweise OLEDs oder time-of-flight-Simulationen,[96,97] bei denen nur geringe Ladungsträgerdichten im Halbleiter auftreten und somit die Coulombwechselwirkung der Ladungsträger untereinander vernachlässigt werden kann.[17] In organischen Feldeffekttransistoren können jedoch Ladungsträgerdichten über 10^{19} cm^{-3} auftreten,[52] bei denen Coulombeffekte eine Auswirkung auf die Ladungsträgermobilität haben können.[73]

Deshalb habe ich als ersten Ansatz ein Modell entwickelt, bei dem die Gesamtzahl der Ladungsträger im Feldeffekttransistor berücksichtigt wird in der Art, dass alle Ladungsträger bis auf einen freien Ladungsträger feste Gitterplätze einnehmen und sich während der Simulation nicht bewegen. Bei dieser Einteilchensimulation mit einer effektiven Energielandschaft bewegt sich nur der eine freie Ladungsträger unter dem Einfluss der externen Felder. Die festen Ladungs-

träger nehmen dabei die Gitterplätze mit den niedrigsten Energien ein, das heißt, die Zustandsdichte des Halbleiters wird von unten aufgefüllt. Dabei unterscheide ich zwei verschiedene Varianten, einmal ohne und einmal mit Berücksichtigung der Coulombwechselwirkung der festen Ladungsträger wie unten ausgeführt. Die besetzten Gitterplätze stehen bei beiden Varianten für den freien Ladungsträger nicht mehr zur Verfügung.

Das zugrunde liegende Modell für die Monte-Carlo-Simulation ist in Abbildung 5.1 dargestellt. Der simulierte Bereich des Transistors ist angegeben, wobei ein Bereich innerhalb des Transistors simuliert wird ohne explizite Berücksichtigung der Injektion und Extraktion von Ladungsträgern an der Source- und Drain-Elektrode. Dies ist eine gültige Näherung für die Simulation eines Transistors im linearen Bereich, solange die Spannung an der Drain-Elektrode klein ist gegenüber der Spannung an der Gateelektrode.[17]

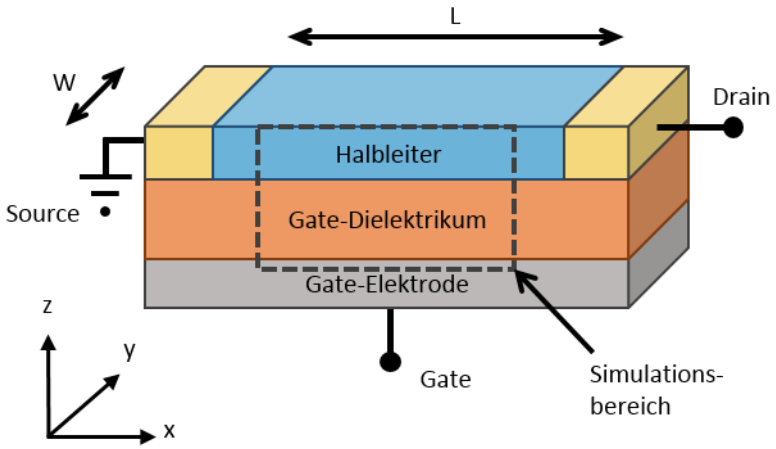

Abbildung 5.1 Schematische Darstellung eines organischen Feldeffekttransistors mit dem Simulationsbereich, den ich in meinem eigenen Modell verwendet habe. Es werden periodische Randbedingungen für die Simulationsbox verwendet.

Für die Berechnung des elektrischen Feldes, welches durch die Gateelektrode verursacht wird, verwende ich für die Einteilchensimulationen mit effektiver Energielandschaft folgende Vorgehensweise. Ich betrachte das Gatedielektrikum mit der Gateelektrode als einen halboffenen Kondensator, wobei die Spannung an der Gateelektrode zuvor eine bestimmte Anzahl an Ladungsträgern im Halbleiter induziert. Die Gateelektrode nehme ich als eine unendlich ausgedehnte Platte an. Das elektrische Feld einer unendlich ausgedehnten Platte beträgt:[98]

$$E(z) = \frac{\sigma}{2\varepsilon_0\varepsilon_G}\frac{z}{|z|} \tag{5.1}$$

Dabei ist σ die Flächenladungsdichte auf der Elektrode und ε_G die relative Dielektrizitätskonstante im Gatedielektrikum. Das elektrische Feld macht einen Sprung an der Grenzschicht zwischen Gatedielektrikum und organischem Halbleiter. Für das elektrische Feld im Halbleiter ergibt sich dann:[55]

$$E_{SC} = \frac{\varepsilon_G U_G}{2\varepsilon_{SC} d_G} \tag{5.2}$$

ε_{SC} ist die Dielektrizitätskonstante im Halbleiter, d_G die Dicke des Gatedielektrikums und U_G die Gatespannung. Dabei habe ich verwendet, dass sich die Flächenladungsdichte aus $n = CU_G$ ergibt, wobei C die Kapazität pro Flächeneinheit des Kondensators um das Gatedielektrikum ist, gemäß:[55]

$$n = CU_G = \frac{\varepsilon_0\varepsilon_G}{d_G}U_G \tag{5.3}$$

Die Anzahl der Ladungsträger in der Simulation lässt sich dann über die Größe der Simulationsbox in x- und y-Richtung l_x und l_y mit der Elementarladung e bestimmen:

$$N = \frac{nl_x l_y}{e} \tag{5.4}$$

Hierbei können freilich nur ganzzahlige Werte in einer Simulation auftreten. Anschließend berechne ich den Potentialunterschied zwischen zwei Lagen in z-Richtung gemäß $\Delta\varphi_z = E_{SC} d_z$, wobei d_z der Gitterabstand in z-Richtung ist. Diesen addiere ich dann in jeder Lage in z-Richtung als statischen Wert auf die Energielandschaft, sodass das elektrische Feld durch die Gatespannung im Lauf der Simulation nicht erneut berechnet werden muss. Das Potential durch die Gateelektrode setze ich dabei in der untersten Halbleiterschicht auf null.

Gegen Ende der Arbeit habe ich die Vorgehensweise für das elektrische Feld durch die Gateelektrode neu evaluiert. Dabei habe ich festgestellt, dass die Berechnung des elektrischen Feldes der Gateelektrode aus der Annahme einer unendlich ausgedehnten geladenen Platte nicht korrekt ist. Stattdessen sollte durchgehend die Annahme eines geladenen Plattenkondensators mit dem Gatedielektrikum als Medium im Kondensator verwendet werden (vergleiche hierzu Kapitel 3.3). Das elektrische Feld im Halbleiter ist dann um einen Faktor 2 größer (siehe Kapitel 5.2). Diese Korrektur trat erst im späteren Verlauf der Arbeit auf und wird in meinen eigenen Vielteilchensimulationen berücksichtigt. Wie ich in Kapitel 5.2 erläutern werde, ist der

Unterschied in der Mobilität zwischen den beiden Vorgehensweisen zur Berechnung des elektrischen Feldes durch die Gateelektrode nur geringfügig, weswegen ich keine nachträgliche Anpassung der Vorgehensweise in den Einteilchensimulationen mit effektiver Energielandschaft durchgeführt habe.

Der Ablauf meines Simulationsprogramms ist schematisch in Abbildung 5.2 dargestellt. Zu Beginn der Simulation werden die Parameter festgelegt. Dazu zählen neben den technischen Simulationsparametern, wie die Größe der Simulationsbox und die maximale Anzahl der Simulationsschritte, elektrische und elektronische Eigenschaften, wie die angelegten Spannungen, Dielektrizitätskonstanten, Kopplungskonstanten, energetische Unordnung, sowie geometrische Eigenschaften und die Temperatur. Als nächster Schritt erfolgt die Initialisierung der Simulation. Dabei wird die Energielandschaft zufällig erstellt, die Anzahl der Ladungsträger wird gemäß Gleichung (5.4) bestimmt und die festen Ladungsträger werden anhand der Vorgaben auf den Gitterplätzen platziert. Je nach Simulationstyp können die Hüpfraten bereits im Voraus berechnet werden, um die Rechenlast während der Simulation gering zu halten. Anschließend beginnt die eigentliche Monte-Carlo-Simulation. Sie besteht aus vielen einzelnen Monte-Carlo-Schritten, die alle ähnlich ablaufen. Zunächst werden die Hüpfraten für die zu einem Zeitschritt möglichen Sprünge des Ladungsträgers ermittelt. Durch Ziehen einer Zufallszahl X_1 im Bereich [0; 1) wird ein Hüpfereignis j ausgewählt über die Beziehung:[76]

$$\frac{\sum_{i=1}^{j-1} k_i}{k_{ges}} \leq X_1 < \frac{\sum_{i=1}^{j} k_i}{k_{ges}} \tag{5.5}$$

Dabei ist k_i die Rate für das Ereignis i und $k_{ges} = \sum_i k_i$. Mit einer zweiten Zufallszahl X_2 aus dem Bereich [0; 1) wird die Zeit τ für dieses Ereignis ermittelt:[76]

$$\tau = -\frac{\ln(X_2)}{k_{ges}} \tag{5.6}$$

Um diese Zeit τ erhöht sich dann die Zeit in der Simulation. Anschließend folgt der nächste Monte-Carlo-Schritt, bis das Abbruchkriterium erfüllt ist. Ich habe als Abbruchkriterium das Erreichen einer bestimmten Anzahl an Simulationsschritten gewählt, da dies bei allen simulierten Parametern ähnliche statistische Schwankungen liefert. Alternativ wäre als Abbruchkriterium das Erreichen einer gewissen Simulationszeit denkbar. Damit ist gemeint, dass die Zeit innerhalb der Simulation, die sich bei jedem Simulationsschritt gemäß Gleichung (5.6) erhöht, einen gewissen Wert überschreitet. Dieser Wert wird jedoch bei tieferen Temperaturen schneller überschritten, da dort Sprünge aufwärts in der Energie mit einer langsameren Rate gemäß Glei-

chung (5.7) stattfinden. Die maximale Anzahl an Monte-Carlo-Schritten beträgt 10^8. Dieses Kriterium habe ich sowohl bei den Einteilchensimulationen mit effektiver Energielandschaft, als auch bei den Vielteilchensimulationen in Kapitel 5.2 angewandt. Als Ergebnis der Simulation erhalte ich die Positionen des Ladungsträgers zu unterschiedlichen Zeitpunkten während der Simulation. Aus der zurückgelegten Strecke des Ladungsträgers in Source-Drain-Richtung Δx in der Zeit Δt kann ich über $\mu = \frac{\Delta x}{\Delta t F_{SD}}$ die Mobilität bestimmen, wobei F_{SD} das elektrische Feld in Source-Drain-Richtung ist. Bei der Ermittlung der Mobilität lasse ich die ersten 10 % der Datenpunkte der Simulation weg, da am Anfang der Simulation Relaxationseffekte auftreten können, die eine Verfälschung der Mobilität bewirken würden.[47] Ich habe überprüft, dass ein größerer Anteil, der zu Beginn der Simulation weggelassen wird, keine Änderung der Mobilität bewirkt und somit Relaxationseffekte als abgeschlossen betrachtet werden können. Um eine Statistik für die erhaltenen Werte der Mobilität zu erhalten, führe ich die Simulation typischerweise 1000 Mal durch und bilde anschließend den Mittelwert über alle einzelnen Mobilitäten.

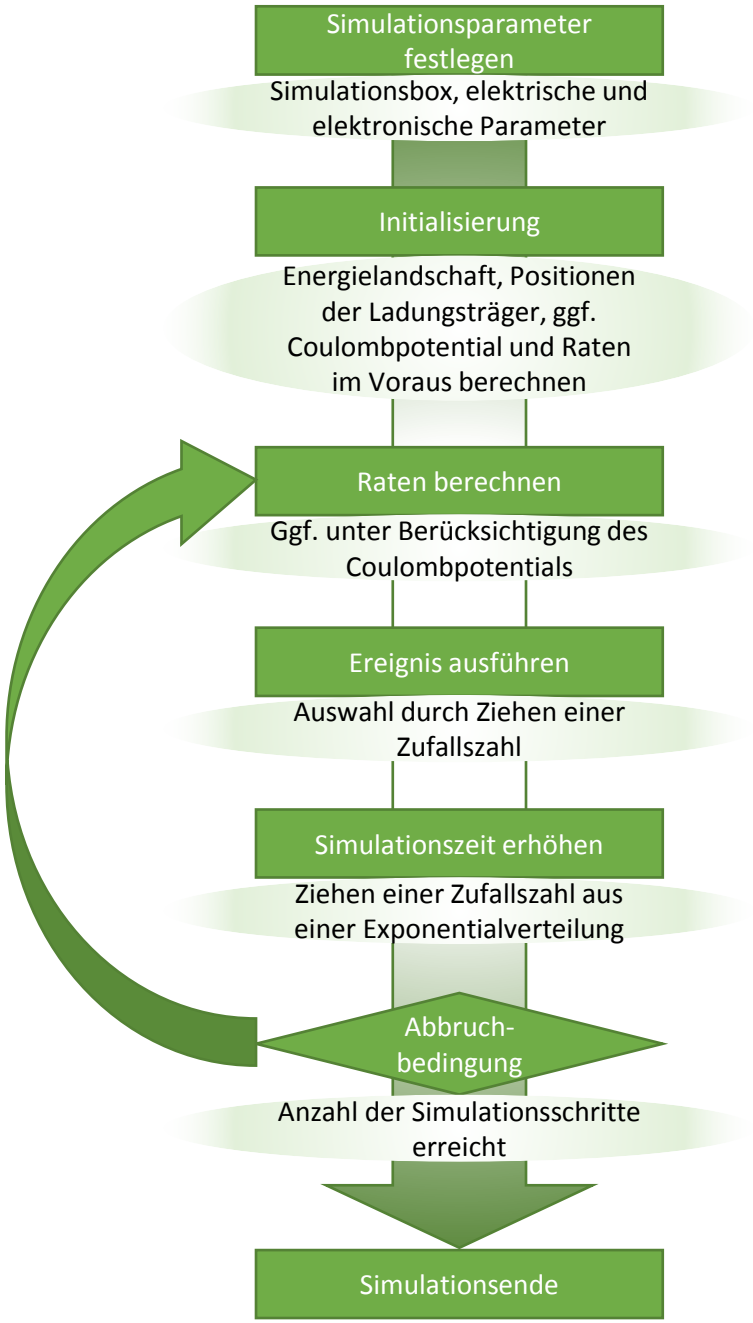

Abbildung 5.2 Ablaufdiagramm einer Monte-Carlo-Simulation zur Modellierung von Ladungstransport.

Als Hüpfraten k_{ij} verwende ich in allen meinen Simulation durchwegs Miller-Abrahams-Hüpfraten:[99]

$$k_{ij} = \nu_0 \exp(-2\gamma|r_{ij}|) \begin{cases} \exp\left(-\frac{\Delta E_{ij}}{k_B T}\right) & \Delta E_{ij} > 0 \\ 1 & \Delta E_{ij} \leq 0 \end{cases} \quad (5.7)$$

ν_0 ist ein allgemeiner Vorfaktor und wird als attempt-to-hop Frequenz bezeichnet. In meinen Einteilchensimulationen habe ich diesen durchgängig zu 10^{13} s^{-1} gewählt. γ ist die inverse Lokalisierungslänge und kann als Kopplungskonstante bezeichnet werden. r_{ij} ist der Abstand zwischen den beiden Gitterpunkten, ΔE_{ij} der Energieunterschied, der Beiträge der intrinsischen energetischen Unordnung aus dem Gaußschen Unordnungsmodell,[47] von externen elektrischen Felder, sowie der Coulombwechselwirkung der Ladungsträger untereinander enthält. k_B ist die Boltzmannkonstante und T die Temperatur. In meiner Simulation berücksichtige ich Sprünge bis zum drittnächsten Nachbar.

5.1.1. Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung

In einem ersten Schritt habe ich eine Einteilchensimulation mit effektiver Energielandschaft „ohne Coulombwechselwirkung“ durchgeführt. Dabei wird bei der Befüllung der Zustandsdichte durch die Ladungsträger die Coulombwechselwirkung zwischen den Ladungsträgern vernachlässigt. Die Gitterplätze werden nacheinander mit Ladungsträgern energetisch von unten aufgefüllt, bis die durch die Gatespannung vorgegebene Anzahl an Ladungsträgern platziert sind. Um die Qualität dieses Modells zu testen habe ich die Temperaturabhängigkeit der Ladungsträgermobilität für einen typischen Parametersatz simuliert. Die Breite der gaußförmig verteilten Energien der Gitterplätze beträgt σ = 50 meV, wobei ich unkorrelierte Energien nach dem Gaußschen Unordnungsmodell verwende.[47] Für die Feldstärke in Source-Drain-Richtung habe ich F_{SD} = 10^5 V cm^{-1} gewählt, die Dicke des Dielektrikums beträgt d_G = 100 nm, die Dielektrizitätskonstante im Halbleiter ε_{SC} = 4, ein typischer Wert für organische Halbleiter,[100] die Dielektrizitätskonstante im Dielektrikum ebenfalls ε_G = 4, was in etwa der von Siliziumdioxid, ein typisches Gatedielektrikum, entspricht, und die inverse Lokalisierungskonstante γ = 5 nm^{-1}. Als Simulationsbox verwende ich 100 x 50 x 3 Gitterpunkte bei einem Gitterabstand von 1 nm.

Die Energie eines Gitterpunkts setzt sich zusammen aus der statistischen Unordnung und dem Potential durch die angelegte Spannung. Bei der Wahl der Gatespannung habe ich berücksichtigt, dass mein Simulationsmodell Ladungstransport nur im linearen Bereich simulieren kann. Die Bedingung, dass die Source-Drain-Spannung klein ist gegenüber der Gatespannung, ist für Gatespannungen größer als 1 V erfüllt. Hierbei ist die Gatespannung von 1 V als Grenzfall zu

sehen, ab dem diese Näherung gültig ist. Die Ergebnisse der Simulation mit dem Modell, bei dem alle Ladungsträger bis auf einen fixiert sind und keinerlei Coulombwechselwirkung zwischen den festen Ladungsträgern berücksichtigt wird, sind in Abbildung 5.3 dargestellt.

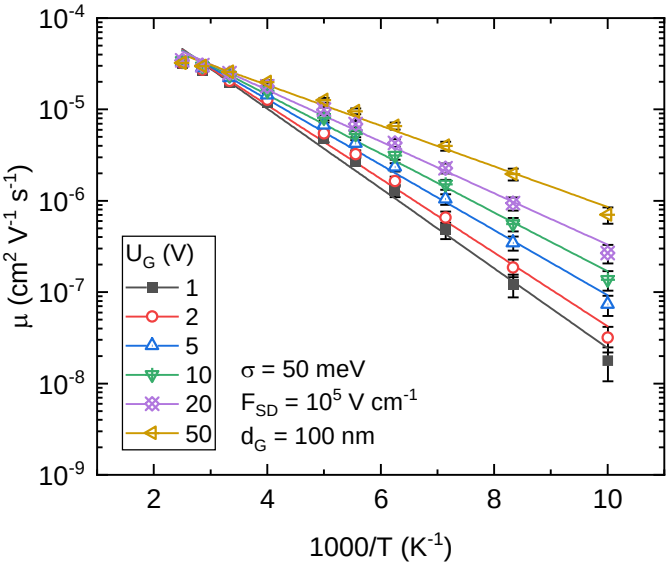

Abbildung 5.3 Mobilität als Funktion der Temperatur für verschiedene Gatespannungen wie angegeben ermittelt aus einer Simulation, bei der sich ein Ladungsträger frei bewegen kann und die übrigen in den energetisch tiefsten Plätzen fixiert sind. Zwischen dem sich bewegenden und den ortsfesten Ladungsträgern besteht keinerlei Wechselwirkung. Die Linien stellen Kurvenanpassungen gemäß Gleichung (3.1) dar und dienen lediglich der Veranschaulichung.

Für die Darstellung der Daten habe ich eine Anpassung an eine Arrheniusgleichung gemäß Gleichung (3.1) durchgeführt. Dabei bilde ich zunächst den natürlichen Logarithmus der Mobilitätswerte und führe in der Auftragung gegen $1000/T$ eine lineare Anpassung durch. Anschließend bilde ich wiederum die Exponentialfunktion der Funktionswerte und stelle die Kurven graphisch dar. Diese Vorgehensweise hat den Vorteil, dass die Kurvenanpassung augenscheinlich besser die Datenpunkte trifft als eine direkte Kurvenanpassung der Gleichung (3.1), insbesondere wenn der lineare Zusammenhang zwischen $\ln(\mu)$ und $1000/T$ nicht über den gesamten Temperaturbereich gegeben ist. Es ist erkennbar, dass die Mobilität in der gewählten halblogarithmischen Darstellung als Funktion der reziproken Temperatur in etwa einen linearen Verlauf bei allen Gatespannungen zeigt und somit der erwarteten Arrheniusgleichung entspricht. Außerdem zeigt sich, dass die Mobilität mit steigender Gatespannung zunimmt.

In meiner Simulation liegt die thermische Gleichgewichtsenergie bei $T \approx 250$ K bei $\langle \varepsilon_\infty \rangle \approx -2.3\sigma$. Für Gatespannungen größer als 5 V liegt die Konzentration der Ladungsträger unter der Annahme, dass sich alle Ladungsträger in der ersten Schicht befinden, oberhalb von 1.1 % je Gitterpunkt, was in etwa einem Ferminiveau von -2.3σ entspricht. Das bedeutet, dass für Temperaturen unterhalb von 250 K und einer Gatespannung von mindestens 5 V die Bedingung erfüllt ist, dass das Ferminiveau oberhalb der thermischen Gleichgewichtsenergie liegt, und damit der lineare Verlauf der Mobilität in diesem Temperaturbereich erklärt werden kann. Außerdem wird in diesem Bereich die Zunahme der Mobilität mit steigender Gatespannung bestätigt, da mit steigender Gatespannung die Ladungsträgerkonzentration und damit das Ferminiveau zunimmt und dadurch Δ in Gleichung (3.3) abnimmt. Für höhere Temperaturen oder niedrigere Gatespannungen, wenn das Ferminiveau unterhalb der thermischen Gleichgewichtsenergie liegt, gilt der Zusammenhang (siehe Gleichung (3.4)):[42,47]

$$\mu(T) \propto \exp\left(-\left(\frac{C\sigma}{k_B T}\right)^2\right) \tag{5.8}$$

In diesem Bereich ist die Mobilität unabhängig von der Ladungsträgerkonzentration, was mit meinem beobachteten leicht gekrümmten Verlauf der Mobilität bei hoher Temperatur, sowie einer schwächeren Abhängigkeit von der Gatespannung übereinstimmt.

5.1.2. Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung

In einem nächsten Schritt habe ich eine Einteilchensimulation mit effektiver Energielandschaft unter Berücksichtigung der Coulombwechselwirkung zwischen den festen Ladungsträgern durchgeführt. Die Coulombwechselwirkung wird dabei beim Besetzen der energetisch tiefsten Zustände in der Art berücksichtigt, dass die Ladungsträger nacheinander in die energetisch tiefsten Plätze gesetzt werden, wobei zwischen jeder Platzierung jeweils die Energielandschaft unter Berücksichtigung des Coulombpotentials der bereits eingefügten Ladungsträger neu berechnet wird. Der freie Ladungsträger bewegt sich dann in einer Energielandschaft mit dem zusätzlichen statischen Coulombpotential der festen Ladungsträger.

Für die Berechnung des Coulombpotentials, das ein Ladungsträger innerhalb der Simulationsbox unter Berücksichtigung der periodischen Fortsetzung der Simulationsbox verursacht, betrachte ich einen Ladungsträger, der sich in der Mitte der Simulationsbox befindet. Aufgrund der periodischen Randbedingungen in x- und y-Richtung erweitere ich die Simulationsbox auf

die Größe $(2l_x + 1) \times (2l_y + 1) \times (2l_z - 1)$, wobei l_x, l_y und l_z die Kantenlängen der Simulationsbox sind. Anschließend werden acht Ladungsträger in den Ecken und Kantenmitten dieser Box platziert. Diese stellen die Ladungsträger dar, die sich jeweils in der Mitte der die Simulationsbox umgebenden Boxen befinden und die periodischen Kopien der Simulationsbox darstellen. Dies ist in Abbildung 5.4 dargestellt. An jedem Gitterpunkt wird nun das elektrostatische Potential durch diese neun Ladungsträger berechnet, wobei immer nur der zu jedem Gitterplatz nächstliegende Ladungsträger berücksichtigt wird. Dadurch wird ausgeschlossen, dass das Potential an einem Gitterplatz mehrfach berechnet wird. Schließlich wird eine kleinere Box mit der Größe der Simulationsbox so in die Box gelegt, dass der Ladungsträger die Koordinaten seiner eigentlichen Position in der Simulationsbox besitzt, was in Abbildung 5.4 durch das grüne Rechteck angedeutet ist. Das gesamte elektrostatische Potential aller Ladungsträger ergibt sich durch Addition der Potentiale der einzelnen Ladungsträger.

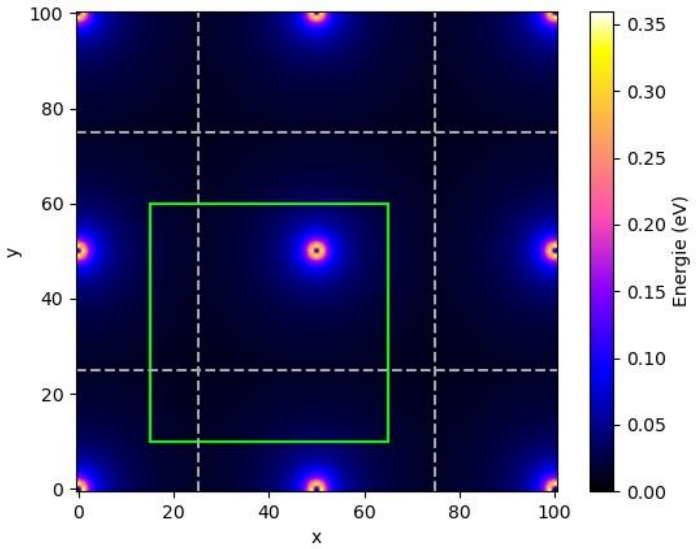

Abbildung 5.4 Berechnung des Coulombpotentials eines einzelnen Ladungsträgers unter Berücksichtigung der periodischen Randbedingungen wie im Text beschrieben. Die grau gestrichelten Linien trennen dabei die Bereiche der periodischen Fortsetzungen voneinander ab. Die Farbcodierung gibt die Stärke des Coulombpotentials an, welches durch den Ladungsträger in der Mitte und in den Kantenmitten und Ecken hervorgerufen wird. Durch geeignete Wahl eines Bereichs (grünes Rechteck) kann das Potential eines Ladungsträgers an einer beliebigen Stelle in der Simulationsbox angegeben werden. Die Größe der Simulationsbox beträgt hier 50 x 50.

Für die effektive Energielandschaft mit Berücksichtigung der Coulombwechselwirkung habe ich ebenfalls eine Simulation mit den gleichen Parametern wie in der Simulation ohne Coulombwechselwirkung in Abbildung 5.3 durchgeführt. Die Ergebnisse dieser Simulation sind in Abbildung 5.5 dargestellt.

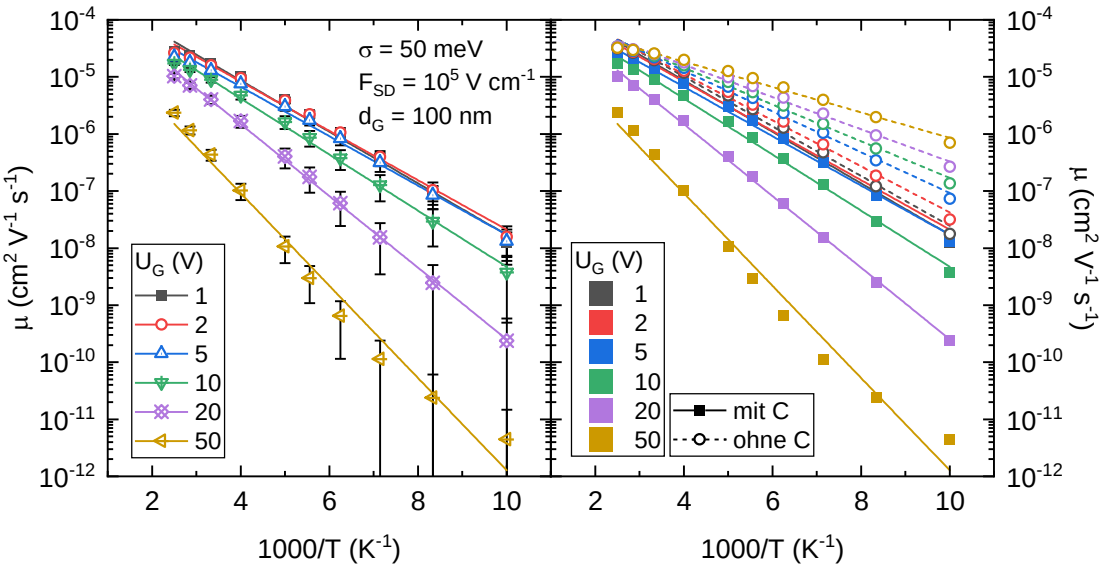

Abbildung 5.5 Links: Mobilität als Funktion der Temperatur für verschiedene Gatespannungen wie angegeben ermittelt aus einer Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung zwischen den festen Ladungsträgern. Rechts: Vergleich der Einteilchensimulationen mit effektiver Energielandschaft mit (durchgezogene Linien, gefüllte Symbole) und ohne (gestrichelte Linien, offene Symbole) Coulombwechselwirkung zwischen den festen Ladungsträgern. Die Daten entsprechen denen aus dem linken Diagramm und aus Abbildung 5.3. Die Linien stellen Kurvenanpassungen gemäß Gleichung (3.1) dar und dienen lediglich der Veranschaulichung.

Mit Berücksichtigung der Coulombwechselwirkung beim Befüllen der Energiezustände ergeben sich für niedrige Gatespannungen annähernd die gleichen Verläufe der Temperaturabhängigkeit der Ladungsträgermobilität wie in der Simulation ohne Coulombwechselwirkung. Bei höheren Gatespannungen ergeben sich jedoch eindeutige Unterschiede. Mit zunehmender Gatespannung ist bei der Simulation mit Berücksichtigung der Coulombwechselwirkung eine abnehmende Mobilität erkennbar. Dies ist im Widerspruch zu der üblichen Erwartung, dass die Ladungsträgermobilität mit steigender Gatespannung zunimmt aufgrund des höheren Fermiveaus.[52] Das leichte Abflachen der Kurve bei einer Gatespannung von 50 V kann möglicherweise auf eine nicht vollständige Relaxation des Ladungsträgers hinweisen und zeigt sich dadurch, dass dort die statistischen Schwankungen größer sind als bei den anderen Datenpunkten.

Ein möglicher Grund für die unerwartete Reduktion der Mobilität kann darin liegen, dass in diesem Modell das zusätzliche Coulombpotential der festen Ladungsträger eine zusätzliche energetische Unordnung erzeugt, welche die Bewegung des freien Ladungsträgers in der Energielandschaft erschwert. Dieser Effekt sollte mit der Größe der Gatespannung und damit der Anzahl an Ladungsträgern anwachsen. In Abbildung 5.6 ist beispielhaft für einen zweidimensionalen Halbleiter der Effekt der Verbreiterung der Energielandschaft durch die Coulombwechselwirkung der Ladungsträger dargestellt. Dabei habe ich die Ladungsträger nacheinander in die energetisch niedrigsten Gitterplätze platziert, einmal ohne und einmal mit Coulombwechselwirkung zwischen den Ladungsträgern. Bei der Platzierung der Ladungsträger mit Berücksichtigung der Coulombwechselwirkung habe ich nach jeder Platzierung eines Ladungsträgers das Coulombpotential neu berechnet und anschließend beim nächsten Ladungsträger mit einbezogen. Initial lag eine Breite der Energien der Gitterplätze von 50 meV vor. Die Dicke des Gatedielektrikums beträgt wie zuvor 100 nm bei einer relativen Dielektrizitätskonstante von 4 im Halbleiter und im Gatedielektrikum. Die Größe der Simulationsbox beträgt 100 x 100 x 1 mit einem Gitterabstand von 1 nm. In Abbildung 5.6 ist die effektive Standardabweichung der Energien der Gitterplätze σ_{eff} einmal mit und einmal ohne Berücksichtigung der Coulombwechselwirkung, sowie die Ladungsträgerdichte als Funktion der Gatespannung dargestellt. Für die Bestimmung der Standardabweichungen habe ich die Gitterplätze aus der Berechnung ausgeschlossen, auf denen sich Ladungsträger befinden. Somit erhalte ich die effektive energetische Unordnung, die der sich bewegende Ladungsträger in der jeweiligen Simulation vorfindet.

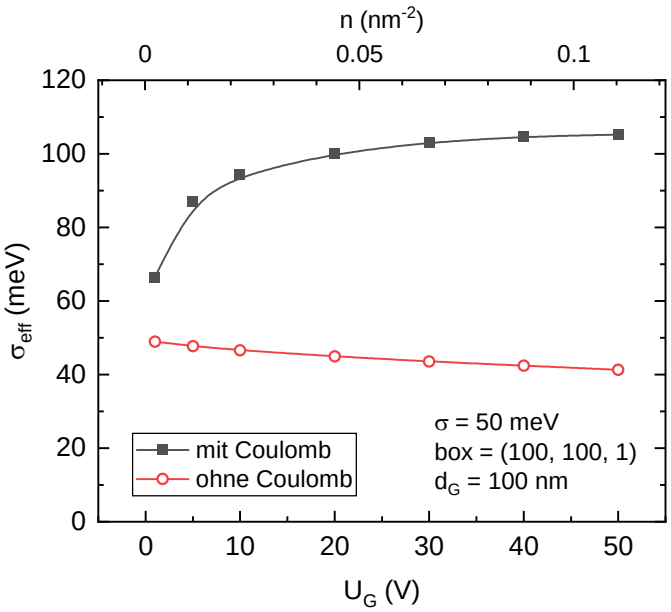

Abbildung 5.6 Effektive Standardabweichung der Energien der Gitterplätze als Funktion der Gatespannung für eine Energielandschaft einmal ohne und einmal mit Berücksichtigung der Coulombwechselwirkung zwischen den Ladungsträgern. Bei der Bestimmung der Standardabweichungen wurden die Gitterplätze ausgeschlossen, auf denen sich Ladungsträger befinden. Die intrinsische energetische Unordnung ohne Ladungsträger beträgt 50 meV. Die obere Achse zeigt dabei die der Gatespannung entsprechende Konzentration an Ladungsträgern je Gitterpunkt. Die Linien dienen lediglich der Veranschaulichung.

Es ist zunächst erkennbar, dass mit steigender Gatespannung die effektive Standardabweichung der Energien der Gitterplätze ohne Berücksichtigung der Coulombwechselwirkung leicht abnimmt. Dies liegt daran, dass in der Bestimmung der Standardabweichung die Gitterplätze mit den Ladungsträgern ausgeschlossen wurden. Mit steigender Gatespannungen werden mehr Gitterplätze energetisch von unten aufgefüllt, sodass sich die energetische Breite der verbleibenden Gitterplätze verkleinert. Bei der Berücksichtigung der Coulombwechselwirkung zeigt sich, dass die effektive Standardabweichung der Energien der Gitterplätze bei kleinen Gatespannungen zunächst stark ansteigt und ab einer Gatespannung von etwa 20 V in eine Sättigung übergeht. Der Sättigungswert liegt dabei bei etwa 100 meV, also dem doppelten der intrinsischen energetischen Unordnung des Halbleiters. Der sich bewegende Ladungsträger erfährt somit mit steigender Gatespannung eine zunehmend breitere Energieverteilung. Das ist konsistent mit der analytisch ermittelten Verbreiterung der Zustandsdichte mit steigender Dotierungskonzentration in ungeordneten organischen Halbleitern.[101] Damit ist nachvollziehbar, dass mit steigender Gatespannung die Ladungsträgermobilität der Einteilchensimulation bei Berück-

sichtigung der Coulombwechselwirkung zwischen den Ladungsträgern abnimmt. Zhou et al. beobachten ebenfalls eine Verringerung der Mobilität durch das Hinzunehmen der Coulombwechselwirkung.[73] Jedoch stellen sie nur bei niedriger energetischer Unordnung einen negativen Effekt der Coulombwechselwirkung auf die mittlere Barrierenhöhe für Sprünge aufwärts in der Energie fest, die sie als den limitierende Faktor für einen Ladungstransport durch Hüpfprozesse sehen. Bei großer energetischer Unordnung hingegen beobachten sie einen leicht gegenteiligen Effekt. Sharma et al. dagegen beobachten einen Anstieg der Mobilität mit steigender Gatespannung insbesondere bei großer energetischer Unordnung von $\sigma/k_BT \geq 3$.[102] Dies entspricht in meiner Simulation dem Bereich $T < 200$ K und stimmt dort mit meiner Beobachtung überein.

5.2. Vielteilchensimulationen

In der vorangegangenen Simulation hat sich gezeigt, dass es einen deutlichen Unterschied macht, ob bei den besetzten Gitterplätzen das Coulombpotential berücksichtigt wird oder nicht. Dabei ist es immer noch eine Näherung anzunehmen, dass sich die Ladungsträger in den energetisch tiefsten Zuständen nicht bewegen können. Für den Fall, dass die energetische Unordnung des Halbleiters groß ist, ist dies möglicherweise eine gute Näherung, wenn die Ladungsträger energetisch so tief sitzen im Vergleich zu den umliegenden Zuständen, dass sie weder thermisch aktiviert noch durch einen sich nahenden Ladungsträger aus den Fallenzuständen entkommen können. Bei geringer energetischer Unordnung kann jedoch davon ausgegangen werden, dass die Ladungsträger leichter aus energetisch tiefen Zuständen entkommen können, wenn sich ein weiterer Ladungsträger annähert. Somit ist es für eine genauere Beschreibung des Ladungstransports aufschlussreich die Bewegung aller Ladungsträger zu betrachten.

Wie bereits in Kapitel 5.1 erwähnt, habe ich beim Erstellen der Vielteilchensimulation die Berechnung des elektrischen Feldes durch die Gateelektrode neu evaluiert und festgestellt, dass die Annahme einer unendlich ausgedehnten geladenen Platte nicht angebracht ist. Statt der in Kapitel 5.1 verwendeten Methode muss ein geladener Plattenkondensator mit dem Gatedielektrikum als Medium im Kondensator angenommen werden.[17] Die Gegenelektrode zur Gateelektrode wird dabei an der Grenzschicht zwischen Gatedielektrikum und Halbleiter angenommen unter Vernachlässigung der Dicke des Halbleiters. Dies ist eine akzeptable Näherung,

da der Ladungstransport vorzugsweise in den untersten Halbleiterschichten stattfindet.[63] Das elektrische Feld durch die Gateelektrode im Halbleiter beträgt dann:

$$E_{SC} = \frac{\varepsilon_G}{\varepsilon_{SC} d_G} U_G \tag{5.9}$$

In Abbildung 5.7 habe ich die Ladungsträgermobilität als Funktion der Temperatur mit meiner Vielteilchensimulation Teil 2, deren Methode ich in Kapitel 5.2.2 erläutern werde, für die beiden Berechnungsarten des elektrischen Feldes durch die Gateelektrode gemäß Gleichung (5.2) und (5.9) dargestellt. Für den Vergleich der beiden Berechnungsweisen des elektrischen Feldes durch die Gateelektrode habe ich meine Vielteilchensimulation Teil 2 verwendet, da sie die bessere Berücksichtigung der Coulombwechselwirkung zwischen den Ladungsträgern liefert, wie sich später zeigen wird.

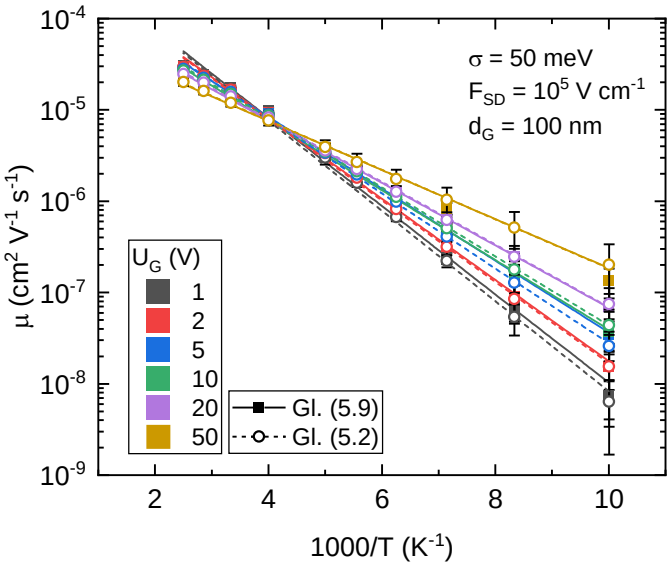

Abbildung 5.7 Ladungsträgermobilität als Funktion der Temperatur für verschiedene Gatespannungen wie angegeben berechnet mit meiner Vielteilchensimulation Teil 2. Das elektrische Feld durch die Gateelektrode wurde einmal nach Gleichung (5.2) (Gl. (5.2)) und einmal nach Gleichung (5.9) (Gl. (5.9)) berechnet.

Es zeigt sich, dass die unterschiedliche Berechnung des Gatepotentials nur geringen Einfluss auf die Ladungsträgermobilität hat. Nur bei Gatespannungen bis 5 V treten kaum sichtbare Unterschiede auf. Obwohl sich die Potentialdifferenz zwischen zwei Halbleiterschichten um einen Faktor von 2 unterscheidet, scheint die Stärke des Gatepotentials ab einer Gatespannung von

10 V ausreichend, um nahezu alle Ladungsträger in der untersten Halbleiterschicht zu konzentrieren. Bei einer Gatespannung von 10 V beträgt die Potentialdifferenz zwischen zwei Halbleiterschichten nach Gleichung (5.9) 100 meV und ist somit deutlich größer als die thermische Energie bei Raumtemperatur. Aufgrund der nur geringen Unterschiede zwischen den beiden Berechnungsweisen des Gatepotentials erscheint eine Wiederholung der Einteilchensimulation mit effektiver Energielandschaft in Kapitel 5.1 nicht notwendig.

5.2.1. Eigene Entwicklung einer Vielteilchensimulation – Teil 1

Um die Näherung der festsitzenden Ladungsträger in den energetisch tiefsten Gitterplätzen zu testen, habe ich eine Simulation entwickelt, bei der sich alle Ladungsträger bewegen können unter Berücksichtigung der Coulombwechselwirkung zwischen den Ladungsträgern. Bezüglich der Berechnung des Coulombpotentials sei hier angemerkt, dass dieser Teil der Modellentwicklung bereits in einem frühen Stadium dieser Arbeit entstanden ist. Deswegen ist die hier beschriebenen Coulombwechselwirkung zwischen den Ladungsträgern abweichend von der Berücksichtigung bei der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung. In einem späteren Abschnitt dieses Kapitels beschreibe ich noch eine weitere eigene Vielteilchensimulation, bei der die Berechnung der Coulombwechselwirkung analog zur Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung ist.

Für die Berechnung des Coulombpotentials im ersten Teil meiner Vielteilchensimulation gehe ich wie folgt vor: Zur Berücksichtigung der periodischen Randbedingungen kopiere ich alle Ladungsträger in der Simulationsbox einmal nach links und rechts neben die Simulationsbox in x-Richtung und einmal nach vorne und hinten in y-Richtung wie in Abbildung 5.8 dargestellt.

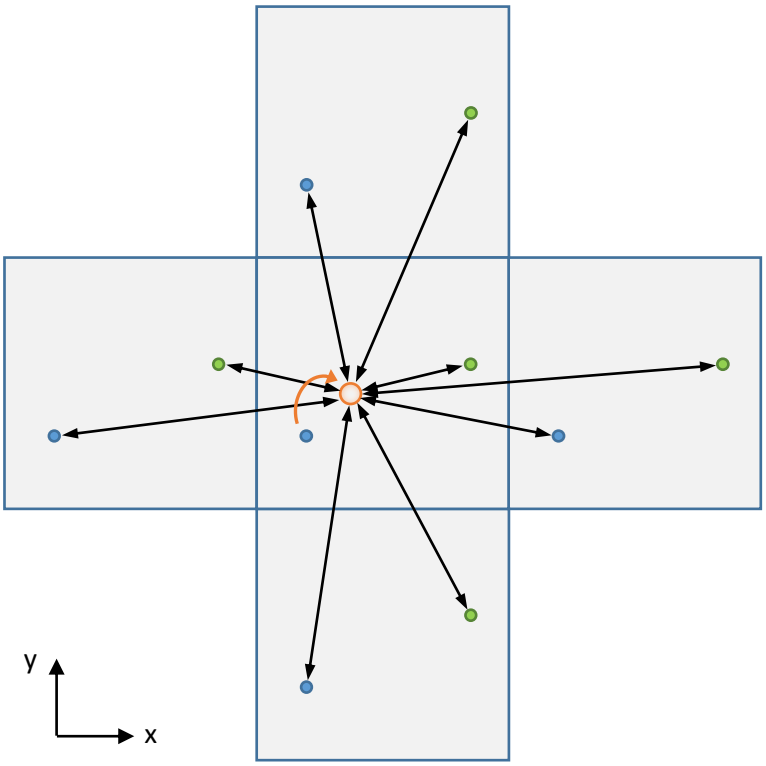

Abbildung 5.8 Schematische Darstellung der periodischen Fortsetzung der Simulationsbox in x- und y-Richtung. Das mittlere Quadrat stellt die Simulationsbox dar, die beiden Ladungsträger sind durch grüne und blaue Kreise angedeutet, der orange Kreis stellt die Position dar, auf die sich der durch den orangen Pfeil markierte Ladungsträger bewegen wird. Durch die schwarzen Pfeile sind alle für die neue Position des Ladungsträgers erforderlichen Coulombwechselwirkungen, wie sie in der Vielteilchensimulation Teil 1 berücksichtigt werden, angedeutet.

Um das Coulombpotential an einem Gitterpunkt zu berechnen, der für einen Sprung eines Ladungsträgers in Frage kommt, berechne ich die Abstände aller Ladungsträger außer dem voraussichtlich hüpfenden zu diesem Gitterpunkt, und daraus das Coulombpotential, wie es in Abbildung 5.8 dargestellt ist. Das Coulombpotential wird anschließend zu der vorhandenen energetischen Unordnung und dem Gradienten des Feldes addiert, um die Energiedifferenz zwischen zwei Gitterplätzen für die Hüpfrate zu berechnen.

Um auszuwählen, welcher Ladungsträger sich zu einem bestimmten Zeitschritt in der Simulation bewegt, bestimme ich die Hüpfraten aller Ladungsträger auf alle möglichen freie Gitterplätze. Die Bestimmung des Hüpfprozesses, der zu einem Zeitschritt stattfinden soll, erfolgt wie bei der Einteilchensimulation mit effektiver Energielandschaft über die Beziehung in Gleichung (5.5), wobei hier die Raten aller Ladungsträger mit eingehen und dadurch auch ausge-

wählt wird, welcher Ladungsträger sich zu einem Zeitschritt bewegt. In die Bestimmung der Simulationszeit für diesen Hüpfprozess nach Gleichung (5.6) gehen dabei auch die Hüpfraten aller Ladungsträger ein.

Die Bestimmung der Mobilität erfolgt analog zur Einteilchensimulation mit effektiver Energielandschaft für jeden Ladungsträger einzeln über den zurückgelegten Weg Δx innerhalb der Simulationszeit Δt gemäß $\mu = \frac{\Delta x}{\Delta t F_{SD}}$. Ebenfalls wie in der Einteilchensimulation mit effektiver Energielandschaft lasse ich einen gewissen Anteil der Datenpunkte zu Beginn der Simulation weg, um Relaxationseffekte auszuschließen. Hier habe ich 20 % als Anteil der Simulation zu Beginn weggelassen, was sich als guter Kompromiss zwischen einer Verfälschung durch Relaxationseffekte und einer geringeren statistischen Schwankung der Mobilität aufgrund längerer Simulationszeit herausgestellt hat. Anschließend bilde ich den Mittelwert über die Mobilitäten aller einzelnen Ladungsträger für die Bestimmung der gesamten Mobilität. Da in einer Simulation bis zu 552 Ladungsträger simuliert werden, war bereits eine Wiederholung eines Parametersatzes ausreichend für eine gute Statistik. Bei den Parametersätzen mit niedriger Gatespannung und somit geringer Anzahl an Ladungsträgern habe ich mehrere Wiederholungen der Simulation durchgeführt bis insgesamt mindestens 100 Ladungsträger simuliert wurden.

Ich habe die Vielteilchensimulation Teil 1 für den selben Parametersatz wie in den Einteilchensimulationen aus dem vorangegangenen Kapitel 5.1 durchgeführt. In Abbildung 5.9 sind die Ergebnisse dazu dargestellt.

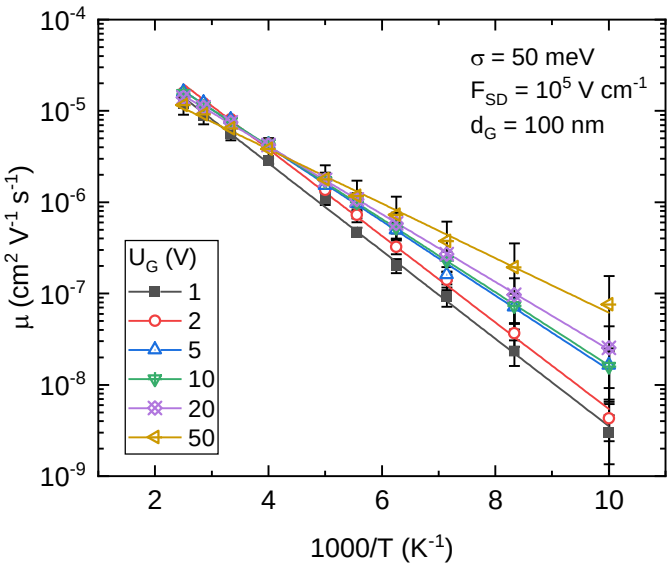

Abbildung 5.9 Ladungsträgermobilität als Funktion der Temperatur ermittelt aus meiner Vielteilchensimulation Teil 1 für verschiedene Gatespannungen wie angegeben. Die Coulombwechselwirkung zwischen den Ladungsträgern wird berücksichtigt. Die Linien stellen Kurvenanpassungen gemäß Gleichung (3.1) dar, dienen jedoch lediglich dazu das Auge zu führen.

Erkennbar ist, dass die Ladungsträgermobilität in der Arrheniusdarstellung ebenfalls wie in den Einteilchensimulationen ein lineares Verhalten zeigt, was somit dem erwarteten Verhalten entspricht.[42] Die Mobilität nimmt zudem als Funktion der Gatespannung zu im Gegensatz zur Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung.

Eine Zunahme der Mobilität mit der Gatespannung lässt auf das Auffüllen von energetisch tief liegenden Zuständen schließen. Ein möglicher Grund, warum in der Vielteilchensimulation Teil 1 die Mobilität mit der Gatespannung zunimmt und bei der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung nicht, könnte sein, dass dadurch, dass bei der Vielteilchensimulation die Ladungsträger beweglich sind, die Ladungsträger sich in anderen energetisch tiefen Zuständen ansammeln werden und somit ein energetisch niedrigerer Gesamtzustand angenommen werden kann als bei der Einteilchensimulation.

5.2.2. Eigene Entwicklung einer Vielteilchensimulation – Teil 2

Im späteren Verlauf dieser Arbeit wird sich zeigen, dass die Coulombwechselwirkung eine wichtige Rolle bei der Interpretation der Ergebnisse der Ladungstransportsimulationen in OFETs

spielt. Aufgrund dessen habe ich meine bestehende Vielteilchensimulation nochmal erweitert, sodass die Berücksichtigung der Coulombwechselwirkung dem entspricht, wie sie in der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung stattfindet.

Für die Berechnung des Coulombpotentials gehe ich wie folgt vor: Zunächst berechne ich das Coulombpotential, das ein einzelner Ladungsträger in einer Simulationsbox mit periodischen Randbedingungen wie in Abbildung 5.4 dargestellt verursacht. Die Berechnung ist identisch zum Coulombpotential eines festsitzenden Ladungsträgers in der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung. Anschließend summiere ich alle Coulombpotentiale der Ladungsträger in der Simulationsbox. Dadurch erhalte ich das Coulombpotential an jedem Gitterpunkt in der Simulationsbox. Für die Berechnung der Hüpfraten eines Ladungsträgers ermittle ich dann die Differenz des Coulombpotentials zwischen dem Ausgangs- und Endpunkt eines einzelnen Hüpfprozesses. Da im gesamten Coulombpotential auch das Coulombpotential des sich bewegenden Ladungsträgers enthalten ist und es zu Abweichungen in der Stromdichte kommt, wenn ein Ladungsträger sich innerhalb seines eigenen Coulombpotentials bewegt,[78] korrigiere ich die Differenz des Coulombpotentials zwischen Ausgangs- und Endpunkt um diesen Betrag. Dieser Korrekturterm C beträgt:

$$C = -\frac{e}{4\pi\varepsilon_0\varepsilon_r r_{if}} \tag{5.10}$$

wobei e die Elementarladung, ε_0 die Dielektrizitätskonstante von Vakuum, ε_r die relative Dielektrizitätskonstante des organischen Halbleiters und r_{if} der Abstand zwischen Ausgangs- und Endpunkt des Hüpfprozesses ist. Dies kann für alle vorher definierten möglichen Hüpfrichtungen zu Beginn der Simulation einmalig berechnet werden und verursacht im Verlauf der Simulation somit keine weitere Rechenlast.

Aus technischen Gründen zur Performancesteigerung, wie es beispielsweise von Li et al. durchgeführt wird,[63] wird das Coulombpotential nach jedem Monte-Carlo-Schritt lediglich aktualisiert anstatt es vollständig neu zu berechnen, indem das Potential des springenden Ladungsträgers an seiner ursprünglichen Position vom Gesamtpotential subtrahiert und an der neuen Stelle addiert wird.

Es wurden ebenso wie in der Vielteilchensimulation so viele Wiederholungen eines Parametersatzes durchgeführt, bis insgesamt eine Anzahl von mindestens 100 Ladungsträgern simuliert wurde. Die Bestimmung der Ladungsträgermobilität μ erfolgt wie zuvor in der Vielteilchensimulation Teil 1 als Mittelwert über die Mobilität aller einzelnen Ladungsträger, die sich aus dem zurückgelegten Weg eines Ladungsträgers in Feldrichtung innerhalb der Simulationszeit ergibt.

Dabei habe ich ebenfalls die ersten 20 % der Datenpunkte aufgrund von möglichen Relaxationseffekten zu Beginn der Simulation weggelassen.

In Abbildung 5.10 ist die Ladungsträgermobilität als Funktion der Temperatur ermittelt mit der Vielteilchensimulation Teil 2 mit dem selben Parametersatz wie in den Einteilchensimulationen mit effektiver Energielandschaft und der Vielteilchensimulation Teil 1 dargestellt.

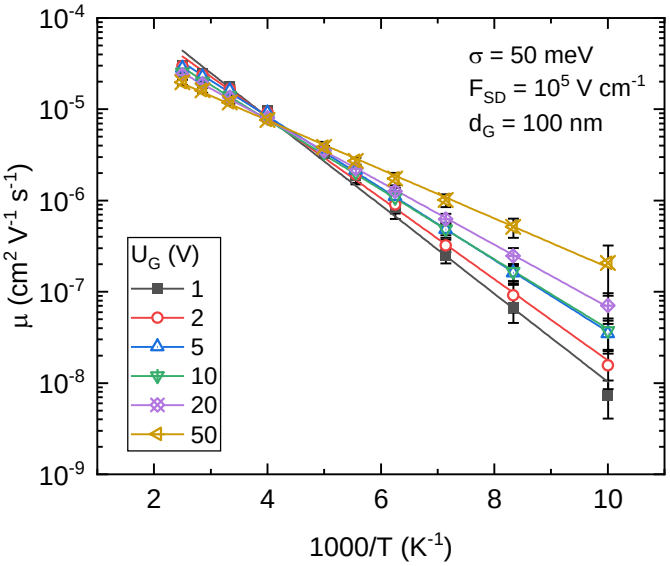

Abbildung 5.10 Ladungsträgermobilität als Funktion der Temperatur ermittelt aus meiner Vielteilchensimulation Teil 2 für verschiedene Gatespannungen wie angegeben. Die Linien stellen Kurvenanpassungen gemäß Gleichung (3.1) dar und dienen lediglich der Orientierung.

Erkennbar ist ebenfalls wie in den Simulationen zuvor ein erwarteter linearer Zusammenhang in der Arrheniusdarstellung.[42] Die Geraden, die ich per Kurvenanpassung gemäß Gleichung (3.1) in die Zeichnung gelegt habe, nehmen betragsmäßig in ihrer Steigung mit zunehmender Gatespannung ab und schneiden sich augenscheinlich in einem Punkt bei etwa T = 250 K. Bei Temperaturen unterhalb 250 K nimmt somit die Mobilität mit steigender Gatespannung zu. Wie bereits in der Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung erläutert, liegt die thermische Gleichgewichtsenergie für T < 250 K und für Gatespannungen von mindestens 5 V unterhalb dem Ferminiveau, sodass eine zunehmende Gatespannung zu einem Auffüllen der Zustandsdichte und damit einer höheren Mobilität führt. Da sich durch das Auffüllen der Zustandsdichte das Ferminiveau ε_F erhöht, verringert sich damit die Aktivierungsenergie $\Delta = \varepsilon_t - \varepsilon_F$ in Gleichung (3.3), was mit der Beobachtung hier übereinstimmt.

5.2.3. Vergleich der Vielteilchensimulationen Teil 1 und Teil 2

Um die beiden Methoden zur Berücksichtigung der Coulombwechselwirkung zwischen den Ladungsträgern vergleichen zu können, habe ich in Abbildung 5.11 die Ergebnisse der beiden Vielteilchensimulationen Teil 1 und Teil 2 gegenübergestellt.

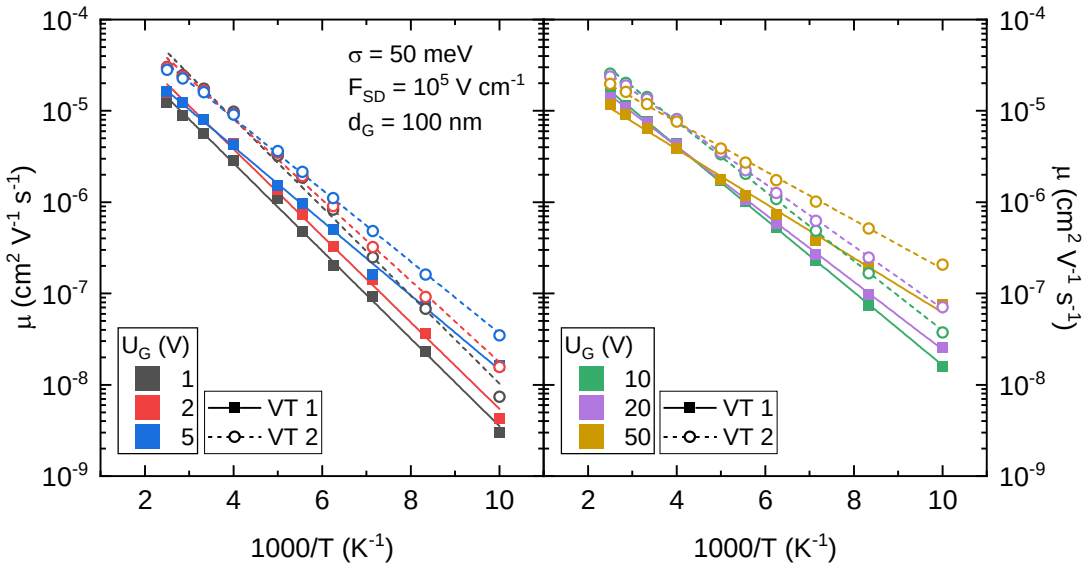

Abbildung 5.11 Vergleich der Ladungsträgermobilität als Funktion der Temperatur für beide Simulationsmethoden, Vielteilchensimulation Teil 1 (VT 1) und Vielteilchensimulation Teil 2 (VT 2). Links ist der Bereich der Gatespannungen 1 bis 5 V, und rechts der Bereich der Gatespannungen 10 bis 50 V dargestellt. Die Daten entstammen aus Abbildung 5.9 und Abbildung 5.10, wobei die Linien lediglich der Orientierung dienen.

Es ist erkennbar, dass die Werte der Vielteilchensimulation Teil 1 bei allen Gatespannungen grob um einen Faktor von 2 unterhalb denen der Vielteilchensimulation Teil 2 liegen. Die Steigungen der Geraden, die Kurvenanpassungen gemäß Gleichung (3.1) entsprechen, sind augenscheinlich sehr ähnlich.

Um die Qualität der Berücksichtigung beider Methoden für die Coulombwechselwirkung besser bewerten zu können, bietet es sich an eine gleichmäßige Verteilung an Ladungsträgern im Halbleiter zu platzieren und das sich ergebende Coulombpotential zu betrachten. Dazu betrachte ich eine einzelne Halbleiterschicht der Größe 100 x 100 x 1, in die ich auf einem quadratischen Gitter 100 Ladungsträger anordne. Der Halbleiter weist dabei keine energetische Unordnung

auf. In Abbildung 5.12 und Abbildung 5.13 habe ich das Coulombpotential der platzierten Ladungsträger, das sich aus den Berechnungsweisen in der Vielteilchensimulation Teil 1 und Teil 2 ergibt, dargestellt. Dargestellt ist ebenfalls ein Querschnitt durch das Coulombpotential in der Mitte und am Rand der Simulationsbox (blaue und orange Linie in jeder Abbildung). An den Stellen der Ladungsträger habe ich das Potential auf null gesetzt. Für die Vielteilchensimulation Teil 1 in Abbildung 5.12 sind die periodisch angrenzenden Simulationsboxen mit dargestellt.

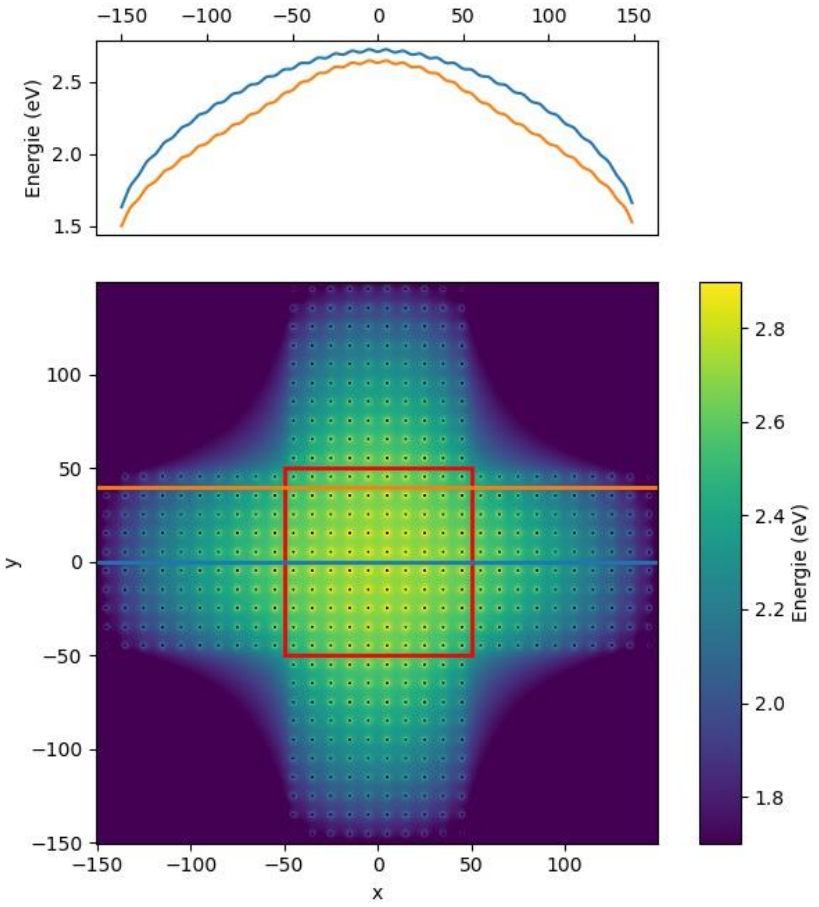

Abbildung 5.12 Coulombpotential der Ladungsträger in der Simulationsbox inklusive der Ladungsträger in den periodischen Fortsetzungen der Simulationsbox für einen Halbleiter ohne energetischer Unordnung wie sie in der Vielteilchensimulation Teil 1 verwendet wird. Die Farbe gibt die Stärke des Coulombpotentials an, die Punkte stellen die Positionen der Ladungsträger dar. Der obere Graph zeigt einen Querschnitt durch das Coulombpotential an der Stelle der blauen und orangen Linie im Bild. Das rote Rechteck markiert den Bereich der Simulationsbox. Die angrenzenden, periodisch fortgesetzten Bereiche sind ebenfalls dargestellt.

Erkennbar ist dabei, dass in der Mitte der Simulationsbox ein deutlicher Potentialberg entsteht. Der Querschnitt durch das Coulombpotential am Rand der Simulationsbox weist ebenfalls ein

Maximum auf. Da die Ladungsträger regelmäßig auf dem Punktgitter angeordnet wurden und das System periodische Randbedingungen in x- und y-Richtung aufweist, ist ein nahezu konstantes Coulombpotential zu erwarten. Dies ist mit der Berechnungsweise der Coulombwechselwirkung in der Vielteilchensimulation Teil 1 nicht gegeben, da dabei die periodischen Fortsetzungen eines sich bewegenden Ladungsträgers selbst zum Coulombpotential an seinem möglichen neuen Gitterplatz beitragen. Hinzukommt, dass die übrigen Ladungsträger in der mittleren Box jeweils fünfmal auf die neue Position des Ladungsträgers wirken (siehe auch Abbildung 5.8), einmal in der mittleren Box, und viermal in den periodischen Fortsetzungen. Außerdem fehlt noch der Beitrag der periodischen Fortsetzungen in den vier Ecken, durch die der minimale Abstand zweier Ladungsträger nochmal geringer sein kann, wenn sich zum Beispiel die Ladungsträger in diagonal gegenüberliegenden Ecken der mittleren Box befinden. Eine Möglichkeit zur Vermeidung der mehrfachen Berücksichtigung der Coulombwechselwirkung wäre die Verwendung eines Cut-off-Radius, innerhalb dessen nur das Coulombpotential berechnet wird.[77]

In Abbildung 5.13 ist das sich ergebende Coulombpotential mit der Berechnungsweise der Vielteilchensimulation Teil 2 für die gleichen Positionen der Ladungsträger wie in Abbildung 5.12 dargestellt. Da die Berechnungsweise des Coulombpotentials hier anders ist als bei der Vielteilchensimulation Teil 1, habe ich nur die Simulationsbox ohne die periodischen Fortsetzungen dargestellt.

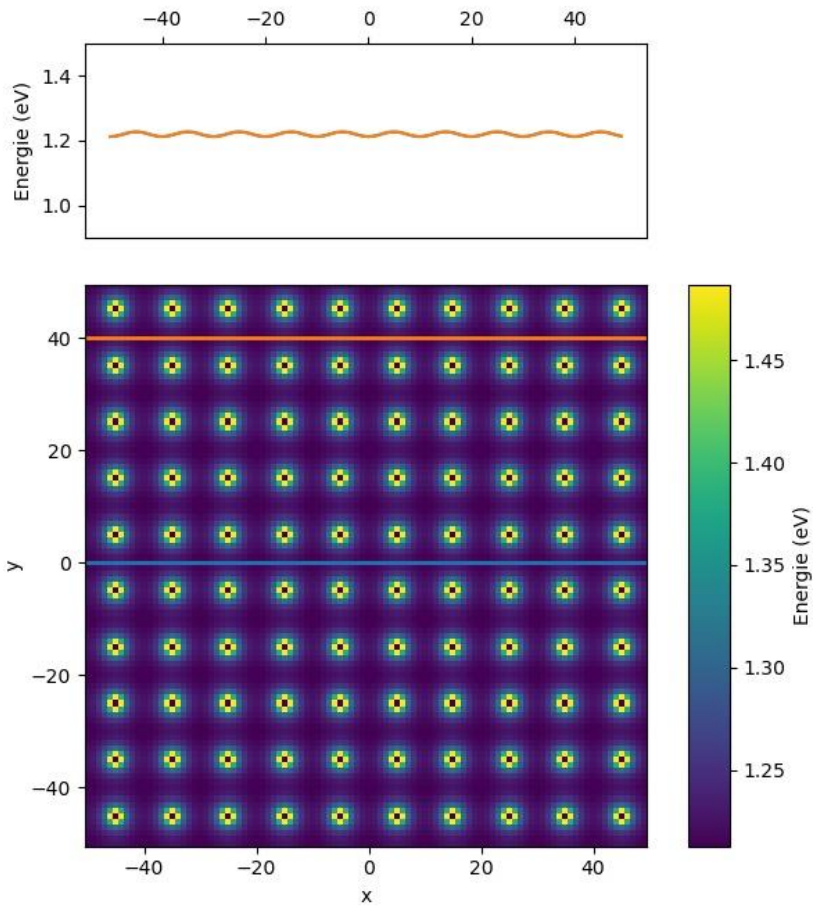

Abbildung 5.13 Coulombpotential der Ladungsträger in der Simulationsbox für einen Halbleiter ohne energetischer Unordnung wie sie in der Vielteilchensimulation Teil 2 verwendet wird. Die Farbe gibt die Stärke des Coulombpotentials an, die Punkte stellen die Positionen der Ladungsträger dar. Der obere Graph zeigt einen Querschnitt durch das Coulombpotential an der Stelle der blauen und orangen Linien im Bild, wobei beide Kurven deckungsgleich sind. Gezeigt ist nur der Bereich der Simulationsbox ohne periodische Fortsetzungen.

Erkennbar ist, dass das Coulombpotential anders als in der Vielteilchensimulation Teil 1 innerhalb der Simulationsbox nahezu konstant ist und somit der Erwartung für eine regelmäßige Verteilung der Ladungsträger mit periodischen Randbedingungen entspricht. Der Querschnitt durch das Coulombpotential in der Mitte und am Rand der Simulationsbox sind deckungsgleich. Die leichte Welligkeit der Querschnitte entsteht durch die regelmäßige Anordnung der Ladungsträger.

Freilich würden sich die Ladungsträger im Fall der Vielteilchensimulation Teil 1 während der Simulation solange umsortieren, bis das Coulombpotential einen nahezu konstanten Verlauf aufweist. Diese Umverteilung ist aber dann ein Artefakt aufgrund des Potentialgradienten, der

durch den Coulombberg entsteht, weswegen auf jeden Fall die Vielteilchensimulation Teil 2 vorzuziehen ist.

5.2.4. Vergleich Einteilchen- und Vielteilchensimulationen

Im Folgenden vergleiche ich beide Vielteilchensimulationen mit beiden Einteilchensimulationen, um ein Verständnis dafür zu bekommen, wie sich die unterschiedliche Berücksichtigung der Coulombwechselwirkung auf den Ladungstransport auswirkt. In Abbildung 5.14 ist die Vielteilchensimulation Teil 1 jeweils mit den Einteilchensimulationen mit effektiver Energielandschaft mit und ohne Coulombwechselwirkung dargestellt.

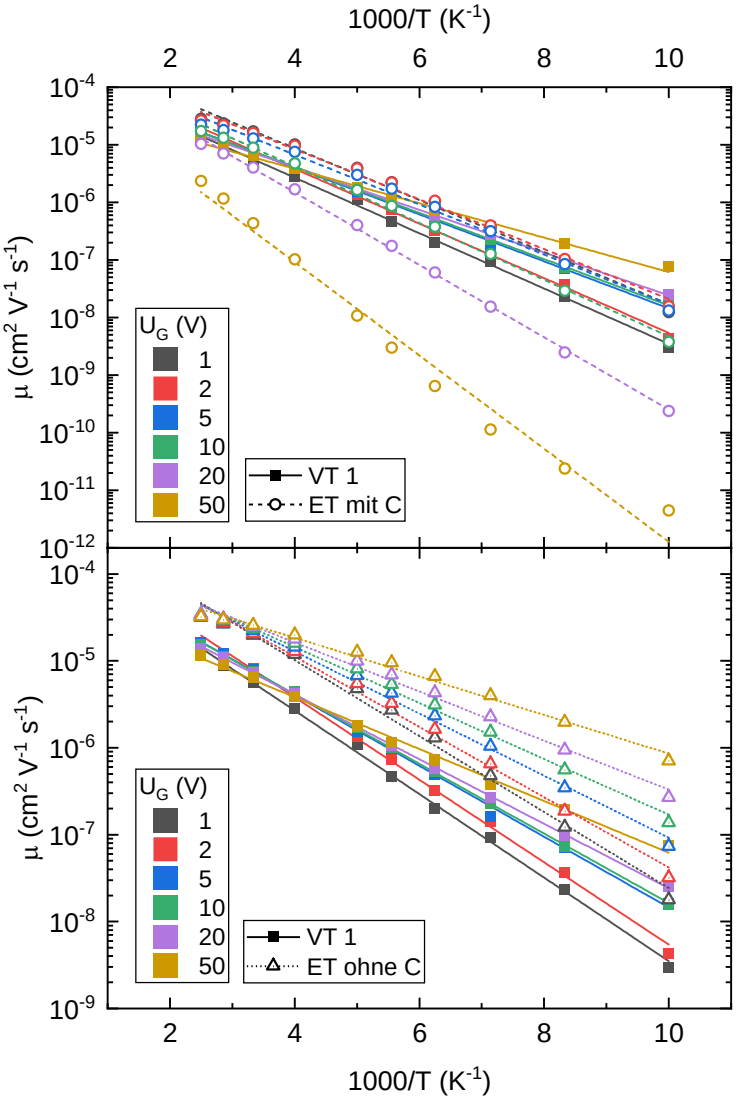

Abbildung 5.14 Vergleich der Vielteilchensimulation Teil 1 (VT 1) mit den Einteilchensimulationen mit effektiver Energielandschaft für verschiedene Gatespannungen wie angegeben. Oben: Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Vielteilchensimulation Teil 1 und der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung (ET mit C). Unten: Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Vielteilchensimulation Teil 1 und der Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung (ET ohne C). Die Linien stellen Anpassungen gemäß Gleichung (3.1) dar und dienen lediglich der Veranschaulichung.

Erkennbar ist im Vergleich zwischen der Vielteilchensimulation Teil 1 und der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung, dass die Mobilität der Vielteilchensimulation Teil 1 bei niedriger Gatespannung zunächst kleiner ist als bei der Einteilchensimulation. Bei höherer Gatespannung kehrt sich das jedoch um, sodass die Mobilität der

Vielteilchensimulation Teil 1 größer ist als die der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung. Wird die Vielteilchensimulation Teil 1 hingegen mit der Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung verglichen, so zeigt sich, dass die Mobilität der Vielteilchensimulation Teil 1 bei allen Gatespannungen kleiner ist als die der Einteilchensimulation. Dies ist plausibel, da eine zusätzliche Abstoßung zwischen den Ladungsträgern zu einer reduzierten Mobilität führt.[103]

In Abbildung 5.15 ist der Vergleich zwischen der Vielteilchensimulation Teil 2 mit den Einteilchensimulationen mit effektiver Energielandschaft mit und ohne Coulombwechselwirkung dargestellt.

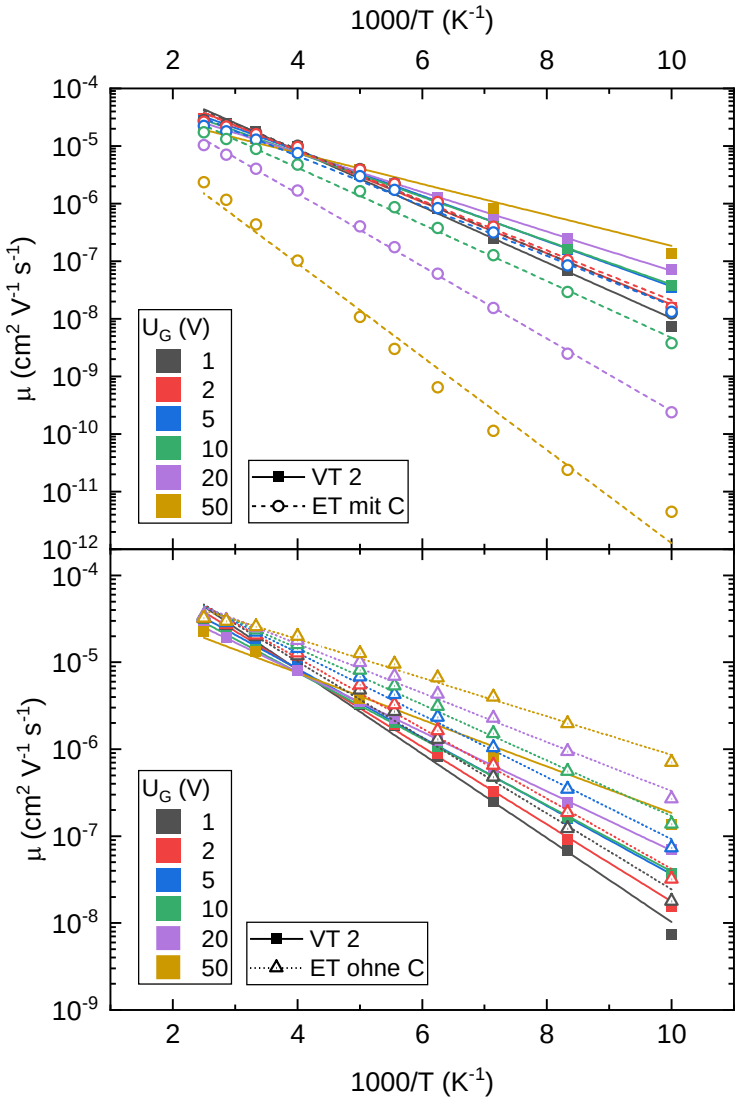

Abbildung 5.15 Vergleich der Vielteilchensimulation Teil 2 (VT 2) mit den Einteilchensimulationen mit effektiver Energielandschaft für verschiedene Gatespannungen wie angegeben. Oben: Ladungsträgermo-

bilität als Funktion der Temperatur ermittelt aus der Vielteilchensimulation Teil 2 und der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung (ET mit C). Unten: Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Vielteilchensimulation Teil 2 und der Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung (ET ohne C). Die Linien stellen Anpassungen gemäß Gleichung (3.1) dar und dienen lediglich der Veranschaulichung.

Hier ist erkennbar, dass die Mobilität der Vielteilchensimulation Teil 2 bei niedriger Gatespannung gut mit den Werten der Einteilchensimulation mit effektiver Energielandschaft mit Coulombwechselwirkung übereinstimmt. Dies ist zu erwarten, da bei niedrigen Gatespannungen weniger Ladungsträger vorhanden sind und die Stärke des Coulombpotentials somit geringer ist. Bei höherer Gatespannung treten deutliche Unterschiede auf, wobei die Mobilität der Vielteilchensimulation Teil 2 höhere Werte liefert als die Einteilchensimulation. Dies ist ebenfalls zu erwarten, da sich in der Vielteilchensimulation alle Ladungsträger als ein Ensemble bewegen können und somit auch Ladungsträger aus energetisch tiefen Gitterplätzen bewegt werden können, wenn dies für das Gesamtsystem energetisch günstiger ist. Der Vergleich zwischen Vielteilchensimulation Teil 2 und Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung zeigt ebenfalls bei niedriger Gatespannung eine gewisse Ähnlichkeit, bei einer Gatespannung von 50 V hingegen besteht jedoch beispielsweise ein Unterschied von einem Faktor von etwa 5 bei einer Temperatur von T = 100 K. Dies entspricht ebenfalls der Erwartung, da neben dem reinen Auffüllen der Zustandsdichte bei der Einteilchensimulation mit effektiver Energielandschaft ohne Coulombwechselwirkung bei der Vielteilchensimulation Teil 2 zusätzlich eine Abstoßung zwischen den Ladungsträgern dazukommt, welche die Mobilität weiter reduziert. Da die Vielteilchensimulation Teil 2 im Vergleich mit den Einteilchensimulationen mit effektiver Energielandschaft mit und ohne Coulombwechselwirkung das erwartete Verhalten zeigt, bestätigt dies nochmal, dass auf jeden Fall die Vielteilchensimulation Teil 2 gegenüber der Vielteilchensimulation Teil 1 vorzuziehen ist.

5.3. Simulationen mit der kommerziellen Software Bumblebee

Im Laufe meiner Promotionsarbeit erhielt ich Zugang zu der kommerziellen Software Bumblebee, die von der Firma Simbeyond Ltd. mit Sitz in den Niederlanden vertrieben wird. Simbeyond entstand als Spin-Off Company der Universität Eindhoven, sodass viele Mitarbeiter ebenfalls mit der Universität Eindhoven assoziiert sind. Bumblebee verwendet kinetische Monte-Carlo-Simulation, um elektrische und optoelektronische Eigenschaften von organischen Bauteilen wie

z.B. organische Leuchtdioden oder Feldeffekttransistoren zu untersuchen. Der Code beruht im Wesentlichen auf den publizierten Arbeiten der Forscher um Peter Bobbert, Reinder Coehoorn und Harm van Eersel.[67,72,104,105]

Der Vorteil der Verwendung von Bumblebee gegenüber meinen eigenen Simulationsprogrammen lag darin, dass die Software direkt verwendet werden konnte ohne sich Gedanken über die technische Ausgestaltung der Software machen zu müssen. Insbesondere enthält Bumblebee einen hochoptimierten Programmcode, sodass sich die Rechenzeit für Transistorsimulationen mit Morphologie in einem akzeptablen Rahmen befindet. Zudem bietet Bumblebee die Möglichkeit relativ einfach morphologische Strukturen im Halbleiter in Form von selbst geschriebenen Skripten integrieren zu können. Nachteilig hingegen erwies sich jedoch das noch nicht vollständig getestete Modul für die Transistorsimulation, wodurch es durch teilweises Auftreten von Fehlern im Programm immer wieder zu Verzögerungen kam.

Die Untersuchungen in den folgenden Kapiteln 6, 7 und 8 basieren auf der Software Bumblebee, weswegen ich die Software in den folgenden Abschnitten näher erläutern möchte. Ich untersuche außerdem den Einfluss von wichtigen technischen Simulationsparameter, die die Coulombwechselwirkung zwischen den Ladungsträgern betreffen. Um ein besseres Verständnis für die verwendete Software zu bekommen, vergleiche ich abschließend die Software mit meinem selbst entwickelten Simulationsprogramm.

5.3.1. Modellierung eines Feldeffekttransistors in Bumblebee

In der Software Bumblebee wird ein kubisches Punktgitter verwendet, um die einzelnen Moleküle des organischen Halbleiters bzw. anderer Materialen darzustellen, auf denen sich Ladungsträger und exzitonische Anregungen durch einen Hüpfprozess bewegen können.

Mit diesem Programm ist neben der Simulation von typischen elektrischen und elektronischen Prozessen in einer OLED auch die Simulation von Ladungstransport in OFETs möglich. Der schematische Aufbau eines OFETs in Bumblebee ist in Abbildung 5.16 gezeigt. Der Bereich im Transistor, der mit Bumblebee in dieser Arbeit simuliert wird, ist mit einem gestrichelten Rahmen eingezeichnet. Die Source- und Drain-Elektrode und damit Ladungsträgerinjektion und -extraktion werden nicht modelliert. Zum einen ist die Injektion für die Transistorgeometrie derzeit noch nicht im Code implementiert, zum anderen wird sie nicht benötigt für meine Untersuchung, da der Fokus meiner Arbeit auf der Ladungstransporteigenschaft im Halbleiter selbst

liegt. Bumblebee simuliert einen OFET mit zwei Gateelektroden ober- und unterhalb des Halbleiters, wie es in Abbildung 5.16 dargestellt ist. Mir wurde mitgeteilt, dass der Grund darin liegt, dass sich die Software aus der Modellierung von OLEDs heraus entwickelt hat und das Modell anschließend auf OFETs übertragen wurde. Durch die beiden Gateelektroden wird ein gleichmäßiges elektrisches Feld in z-Richtung sichergestellt, das für die Annahme von unendlich ausgedehnten geladenen Platten gilt. Damit ist es nicht mehr nötig das gesamte dreidimensionale Feld durch die Gateelektrode, welches sich auch ins Vakuum oberhalb des Halbleiters erstrecken kann, zu berechnen. Die Hauptbeschränkung dieses Modells liegt darin, dass mit dieser Konfiguration nur Simulationen im linearen Bereich des Transistors möglich sind. Nur in diesem Bereich ist die Annahme eines konstanten Feldes durch die Gateelektrode eine gültige Näherung. Dies ist erfüllt, solange die Source-Drain-Spannung klein ist gegenüber der Gatespannung.[17]

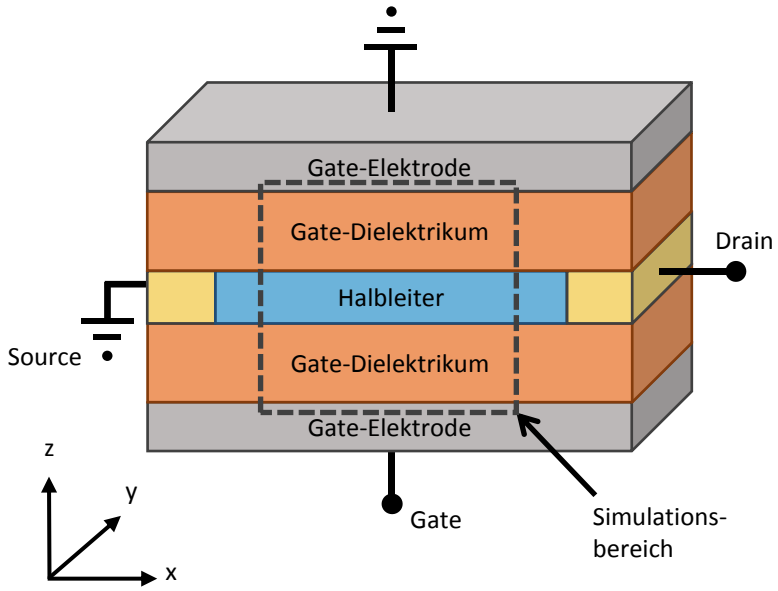

Abbildung 5.16 Schemazeichnung eines organischen Feldeffekttransistors mit dem Simulationsbereich, wie er in der Software Bumblebee modelliert wird.

Da das elektrische Feld durch die untere Gateelektrode von den Ladungsträgern im Halbleiter abgeschirmt wird und das elektrostatische Potential nur bis auf eine additive Konstante festgelegt ist, muss in den Simulationsparametern von Bumblebee eine Position angegeben werden, an der das elektrostatische Potential null sein soll. Dies ist typischerweise am oberen Rand des Halbleiters am Übergang zum oberen Dielektrikum der Fall. Um zu erreichen, dass das Potential an dieser Stelle null ist, nimmt Bumblebee eine Anpassung der Gatespannung vor, solange bis

diese Bedingung erreicht ist. Die Gatespannung ändert sich dabei typischerweise weniger als 1 %.

Im Vergleich dazu habe ich in meinen eigenen Simulationsprogrammen lediglich ein linear ansteigendes bzw. abfallendes Potential in z-Richtung dazu addiert, welches sich für eine unendlich ausgedehnte gleichmäßig geladene Platte ergibt. Durch die Bewegung der Ladungsträger ist dadurch nicht immer sichergestellt, dass das Potential an einer gewissen Stelle null wird.

Der Ablauf der Simulation in Bumblebee ist identisch zum Ablauf meiner eigenen Simulation, wie sie in Abbildung 5.2 dargestellt ist. Als Abbruchkriterium wird in Bumblebee jedoch zusätzlich die Stromdichte verwendet. Sobald die Schwankungen von Minimum und Maximum der Stromdichte einen vorher eingestellten Wert unterschreiten, wird die Simulation gestoppt, ebenso wie wenn die maximale Anzahl an Simulationsschritten erreicht wurde.

Bei der Simulation von Ladungstransport werden in Bumblebee für die Hüpfraten der Ladungsträger ebenfalls wie in meiner Simulation Miller-Abrahams-Raten verwendet. Dabei gilt für die Hüpfrate k_{ij} zwischen zwei Gitterplätzen i und j:[99]

$$k_{ij} = \nu_0 \exp\left(-2\gamma\left|r_{ij}\right|\right) \begin{cases} \exp\left(-\dfrac{\Delta E_{ij}}{k_B T}\right) & \Delta E_{ij} > 0 \\ 1 & \Delta E_{ij} \leq 0 \end{cases} \tag{5.11}$$

ν_0 ist die attempt-to-hop Frequenz, γ die inverse Lokalisierungslänge, r_{ij} der Abstand zwischen den beiden Gitterplätzen, ΔE_{ij} der Energieunterschied zwischen den Gitterplätzen i und j, k_B die Boltzmannkonstante und T die Temperatur.

5.3.2. Bestimmung der Coulombwechselwirkung in Bumblebee

In die Energiedifferenz in den Hüpfraten der Ladungsträger nach Gleichung (5.11) geht die Coulombwechselwirkung zwischen den Ladungsträgern, sowie Wechselwirkungen mit den Spiegelladungen an metallischen Elektroden mit ein. In Bumblebee werden die Spiegelladungen durch die beiden Gateelektroden mit berücksichtigt, jedoch ist dieser Beitrag bei der Verwendung von dicken Gatedielektrika gering. Ich habe typischerweise eine Dicke des Gatedielektrikums von 100 nm verwendet, wodurch der Effekt von Spiegelladungen an den Gateelektroden vernachlässigt werden kann.

In Bumblebee wird die Coulombwechselwirkung zwischen Ladungsträgern und ihren Spiegelladungen in drei verschiedene Terme aufgeteilt, um einen guten Kompromiss zwischen der Genauigkeit der Coulombwechselwirkung und der Rechenzeit zu erzielen.[77] Schematisch ist diese

Berücksichtigung der Coulombwechselwirkung in Abbildung 5.17 dargestellt. Innerhalb einer Kugel mit Radius r_{cut}, dem sogenannten Cut-off-Radius, wird die Coulombwechselwirkung zwischen Ladungsträgern exakt berechnet. Darüber hinaus wird für den langreichweitigen Anteil der Coulombwechselwirkung das Punktgitter der Simulationsbox in Ebenen senkrecht zur z-Richtung aufgeteilt. In jeder dieser Ebenen wird aus der Anzahl der sich darin befindenden Ladungsträger die Flächenladungsdichte dieser Ebene bestimmt. Das Coulombpotential der Flächenladungsdichten wird anschließend über die Poisson-Gleichung berechnet. Ladungsträger, die sich innerhalb der Kugel mit dem Cut-off-Radius befinden, werden sowohl in der exakten Berechnung des Coulombpotentials und der Berechnung durch die Ebenen mit einer Flächenladungsdichte berücksichtigt. Diese Doppelberechnung wird anschließend innerhalb der Kugel wieder herausgerechnet, indem die Flächenladungsträgerdichte in den Schnittflächen zwischen den Ebenen und der Kugel abgezogen wird. Für die beiden Anteile der Coulombwechselwirkung, dem exakten Anteil innerhalb des Cut-off-Radius und den Flächenladungsdichten der Schichten, werden Spiegelladungen der metallischen Elektroden, im Fall eines Transistors die der Gateelektrode, berücksichtigt. Standardmäßig werden in der Simulation Spiegelladungen bis zu einer Ordnung von 100 berücksichtigt. Dabei ist mit der Ordnung der Spiegelladung im Wesentlichen die Anzahl der Mehrfachreflexionen gemeint. Mathematisch handelt es sich um den Endwert der Laufvariable in der Summation in Gleichung (7) von Referenz [77]. Der Cut-off-Radius beträgt standardmäßig r_{cut} = 10a, wobei a die Gitterkonstante ist. Die Ordnung der Spiegelladungen habe ich in meinen Simulationen nicht verändert, da sie sich in früheren Arbeiten als praktikabel mit einer ausreichenden Genauigkeit erwiesen haben.[77]

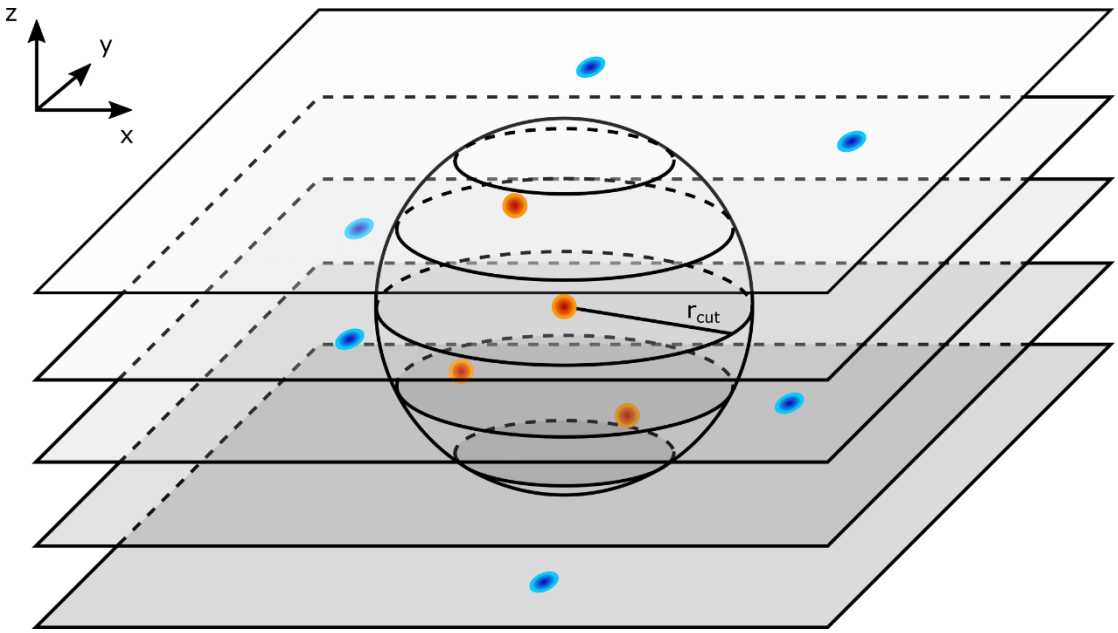

Abbildung 5.17 Schematische Darstellung der Berücksichtigung der Coulombwechselwirkung in Bumblebee. Die Kugel in der Mitte mit dem Cut-off-Radius r_{cut} markiert den Bereich, innerhalb dessen die Coulombwechselwirkung zwischen den Ladungsträgern exakt berechnet wird. Ladungsträger, die sich innerhalb der Kugel befinden, sind als orange Punkte dargestellt. Die Flächen stellen einzelne Ebenen des Punktgitters in z-Richtung im Transistor dar, innerhalb denen die Flächenladungsdichte aller in der jeweiligen Ebene befindlichen Ladungsträger bestimmt wird. Ladungsträger, die sich außerhalb der Kugel in der Mitte befinden, sind als blaue Punkte dargestellt.

5.3.3. Berechnung der Ladungsträgermobilität aus der Simulation

Die Stromdichte ist ein direktes Ergebnis der Simulation. Sie ergibt sich aus der Zahl der Ladungsträger, die pro Zeiteinheit durch eine Fläche senkrecht zur Feldrichtung durchtreten, und wird direkt in der Ausgabedatei angegeben. Dabei wird zu jedem gespeicherten Simulationszeitpunkt der Mittelwert der Stromdichte durch alle Schichten senkrecht zur Source-Drain-Feldrichtung ausgegeben, wobei zwischen zwei gespeicherten Simulationszeitpunkten 100,000 Simulationsschritte liegen. Um einen zeitlichen Mittelwert der Stromdichte über die Dauer der Simulation anzugeben, bilde ich den Mittelwert über die zweite Hälfte der Simulation, um Relaxationseffekte am Anfang der Simulation ausschließen zu können. Als Fehler für die Stromdichte verwende ich die Standardabweichung der Werte der Stromdichte zu jedem Zeitschritt in der Ausgabedatei in der zweiten Hälfte der Simulation.

Die Stromdichte rechne ich anschließend nach dem in Kapitel 3.3 beschriebenen Verfahren in eine Ladungsträgermobilität um. Der statistische Fehler für die Ladungsträgermobilität in der

Simulation lässt sich anschließend aus dem Fehler der Stromdichte mittels Fehlerfortpflanzung berechnen.

5.3.4. Einfluss relevanter Parameter der Coulombwechselwirkung auf die Ladungsträgermobilität in der Software Bumblebee

Um die Ergebnisse der Ladungsträgermobilität, die ich durch Simulation mit der Software Bumblebee erhalte, bewerten zu können, habe ich den Einfluss des Cut-off-Radius und die Berücksichtigung der Spiegelladungen in den Gateelektroden, die bei der Bestimmung der Coulombwechselwirkung eine Rolle spielen können, auf die Ladungsträgermobilität untersucht. In Abbildung 5.18 ist die Ladungsträgermobilität für den selben Parametersatz wie in den vorangegangenen Einteilchen- und Vielteilchensimulationen aus meiner eigenen Entwicklung ermittelt durch Simulation mit der Software Bumblebee dargestellt. Als Cut-off-Radius für die Coulombwechselwirkung habe ich hier R_c = 10 nm verwendet.

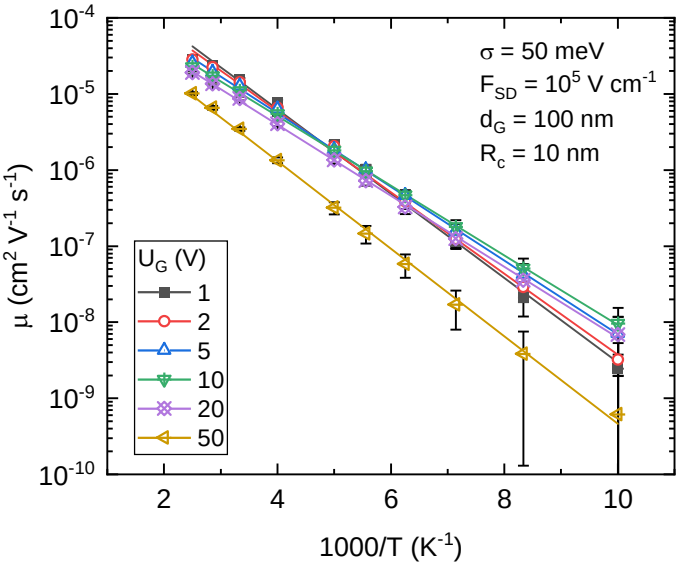

Abbildung 5.18 Mit der Software Bumblebee simulierte Ladungsträgermobilität als Funktion der Temperatur in einem OFET für verschiedene Gatespannungen wie angegeben. Die energetische Unordnung im Halbleiter beträgt 50 meV bei einem Cut-off-Radius für die Coulombwechselwirkung von R_c = 10 nm. Die Linien stellen Kurvenanpassungen gemäß Gleichung (3.1) dar.

Erkennbar ist ein linearer Zusammenhang in der Arrheniusdarstellung, was dem erwarteten Zusammenhang entspricht.[42] Die Kurven für die verschiedenen Gatespannungen liegen nahe beieinander mit einer leichten Zunahme der Mobilität bei tiefer Temperatur mit der Gatespannung. Eine Ausnahme bildet die Gatespannung 50 V, welche deutlich unterhalb der übrigen Kurven liegt. Da die Ladungsträgerdichte mit steigender Gatespannung zunimmt, lässt sich hier bereits vermuten, dass Coulombeffekte hierbei eine Rolle spielen können.

Da die Coulombwechselwirkung in der Software Bumblebee innerhalb der Kugel mit dem Cut-off-Radius exakt und darüber hinaus durch Ebenen mit einer Flächenladungsdichte berechnet wird, spielt die Wahl des Cut-off-Radius eine wichtige Rolle. Bumblebee gibt standardmäßig einen Cut-off-Radius von 10 Gitterplätzen vor, der sich für die Ladungstransportsimulation von sandwichförmigen organischen Bauteilen bewährt hat.[77] In OFETs kann jedoch eine deutlich größere Ladungsträgerkonzentration auftreten als dies beispielsweise in OLEDs der Fall ist. Deswegen habe ich den Einfluss des Cut-off-Radius auf die Ladungsträgermobilität in OFETs untersucht.

In Abbildung 5.19 ist der Einfluss des Cut-off-Radius auf die temperaturabhängige Ladungsträgermobilität in der Software Bumblebee für die Bereiche 1 – 10 V und 10 – 50 V Gatespannung gezeigt.

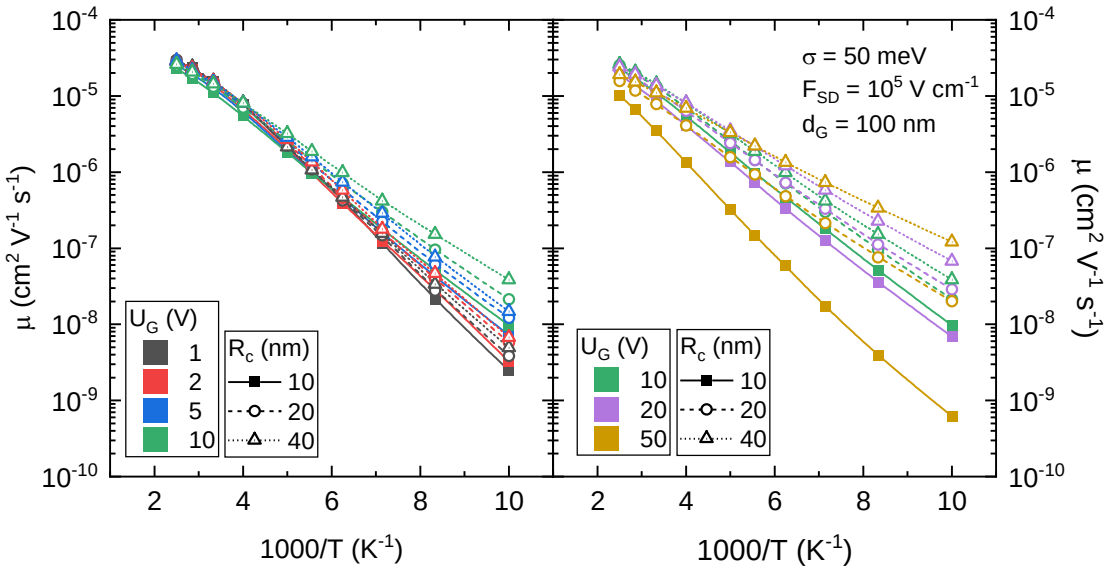

Abbildung 5.19 Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Simulation mit Bumblebee mit unterschiedlichen Cut-off-Radien der Coulombwechselwirkung für verschiedene Gatespannungen wie angegeben. Links sind die Werte für Gatespannungen von 1 – 10 V angegeben, rechts für Gatespannungen von 10 – 50 V. Die Linien dienen lediglich der Veranschaulichung.

Für den Bereich kleiner Gatespannungen von 1 – 10 V liegen die Kurven für die verschiedenen Gatespannungen nahe beieinander und es treten nur geringe Änderungen mit dem Cut-off-Radius insbesondere bei tiefer Temperatur auf. Für die Gatespannungen 20 und 50 V nimmt jedoch die Mobilität mit steigendem Cut-off-Radius bei allen Temperaturen merklich zu. Eine Berechnung der Coulombwechselwirkung mit größerem Cut-off-Radius bewirkt somit eine Erhöhung der Mobilität in dem dargestellten Parameterbereich. Daraus folgt, dass mit kleinerem Cut-off-Radius die Coulombwechselwirkung überschätzt wird. Dadurch fällt die intrinsische energetische Unordnung des Halbleiters von σ = 50 meV weniger stark ins Gewicht.

Für eine Gatespannung von 5 V ist nahezu keine Abhängigkeit der Ladungsträgermobilität vom Cut-off-Radius zu erkennen, weswegen eine Verwendung des Cut-off-Radius von 10 nm, wie er standardmäßig durch die Simulation vorgegeben wird, unkritisch ist. Diese Gatespannung wurde in den Kapiteln 6, 7 und 8 verwendet, weswegen der dort verwendete Cut-off-Radius von 10 nm ausreichend ist.

In Kapitel 7 wird zudem auch eine Gatespannung von 20 V verwendet. Da die Simulationen dort nur bei Raumtemperatur durchgeführt wurden, wo der Unterschied zwischen den verschiedenen Cut-off-Radien geringer ist, ist der verwendete Cut-off-Radius von 10 nm dort ebenfalls vertretbar. Der Unterschied der Mobilität zwischen R_c = 10 nm und R_c = 40 nm bei T = 300 K liegt bei einem Faktor von 1.6. Die beobachteten Trends werden dadurch nicht verändert.

Ein weiterer wichtiger Parameter bei der Berechnung der Coulombwechselwirkung zwischen Ladungsträgern ist die Berücksichtigung von Spiegelladungen, die an metallischen Oberflächen auftreten. In der Simulation mit der Software Bumblebee sind dies die Gateelektroden ober- und unterhalb des organischen Halbleiters. Da die Source- und Drain-Elektrode nicht mit simuliert werden, treten dort auch keine Spiegelladungen auf. Durch die Verwendung zweier Gateelektroden, an denen Spiegelladungen auftreten können, stellt sich die Frage, ob die Berücksichtigung der Spiegelladungen in der Coulombwechselwirkung einen Einfluss auf die Ladungstransporteigenschaften haben. Dies habe ich im Folgenden beispielhaft für zwei verschiedene Gatespannungen untersucht. Dabei habe ich die Temperaturabhängigkeit der Mobilität mit Berücksichtigung von Spiegelladungen, wie es in Bumblebee Standard ist, verglichen mit dem Fall, dass keine Spiegelladungen berücksichtigt werden. In der Software Bumblebee habe ich dazu die Anzahl der Spiegelladungen, die berücksichtigt werden, auf null gesetzt.

In Abbildung 5.20 ist für die Gatespannung 5 und 50 V die Temperaturabhängigkeit der Ladungsträgermobilität dargestellt für die gleichen Parameter wie in den Einteilchen- und Vielteilchensimulation zuvor in diesem Kapitel.

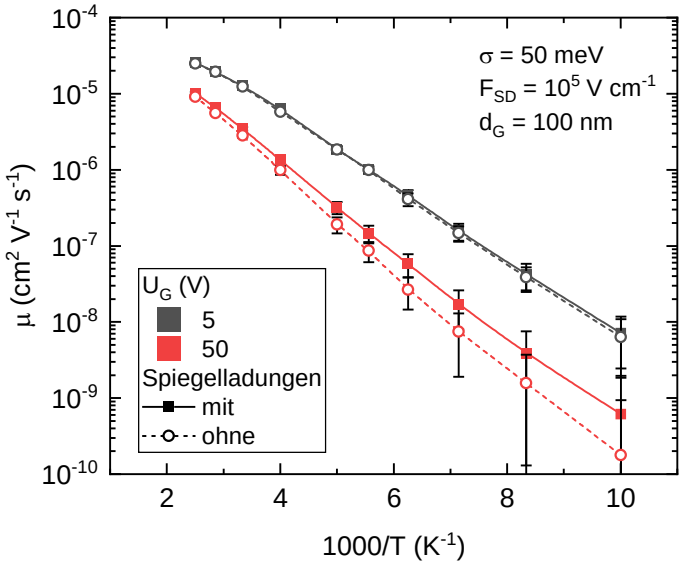

Abbildung 5.20 Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Simulation mit Bumblebee einmal mit und einmal ohne Berücksichtigung von Spiegelladungen in der Gateelektrode für verschiedene Gatespannungen wie angegeben. Die Linien dienen lediglich der Veranschaulichung.

Erkennbar ist, dass bei einer Gatespannung von 5 V nahezu kein Unterschied zwischen einer Berücksichtigung und keiner Berücksichtigung von Spiegelladungen ist. Für eine Gatespannung von 50 V treten bei tiefen Temperaturen Unterschiede auf. Obwohl die Werte der Mobilität mit und ohne Berücksichtigung von Spiegelladungen innerhalb der Fehlertoleranzen übereinstimmen, ist es doch empfehlenswert Spiegelladungen in der Berechnung der Coulombwechselwirkung zu berücksichtigen.

5.3.5. Vergleich der Ergebnisse aus eigenen Simulationsprogrammen mit Bumblebee

Um abschließend bewerten zu können, welche der vorgestellten Simulationsmethoden am geeignetsten für die Untersuchung von morphologischen Parametern auf den Ladungstransport in OFETs ist, vergleiche ich die Ergebnisse der Ladungstransportsimulation der Software Bumblebee mit denen meiner eigenen Entwicklung. Innerhalb meiner eigenen Entwicklung hat sich die Vielteilchensimulation Teil 2 als am verlässlichsten in Bezug auf die Berücksichtigung der Coulombwechselwirkung herausgestellt, weswegen ich diese mit der Software Bumblebee vergleichen werde.

In Abbildung 5.21 ist die Temperaturabhängigkeit der Ladungsträgermobilität ermittelt mit der Software Bumblebee und meiner eigenen Vielteilchensimulation Teil 2 dargestellt. Die Daten entsprechen denen in den Abbildungen Abbildung 5.18 und Abbildung 5.19, wobei ich die Daten der Software Bumblebee mit einem Cut-off-Radius von 40 nm verwendet habe.

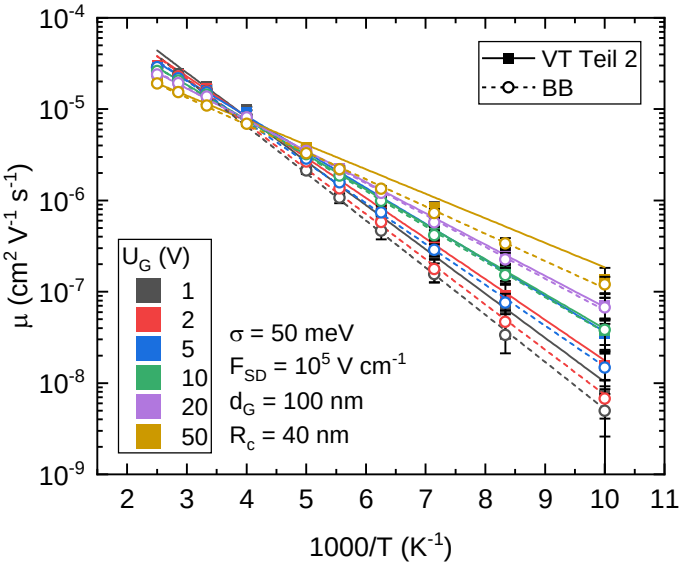

Abbildung 5.21 Vergleich der Ladungsträgermobilität als Funktion der Temperatur ermittelt aus der Simulation mit Bumblebee (BB) und meiner Vielteilchensimulation Teil 2 (VT Teil 2) für verschiedene Gatespannungen wie angegeben. In der Simulation mit Bumblebee wurde ein Cut-off-Radius von 40 nm verwendet. Die Linien dienen lediglich der Veranschaulichung.

Es ist erkennbar, dass die Kurven aus beiden Simulationsprogrammen gut übereinstimmen. Lediglich für eine Gatespannung von 50 V ist eine leichte Abweichung bei tiefen Temperaturen erkennbar, diese liegt jedoch innerhalb der Fehlertoleranzen, die sich wie in Kapitel 5.3.3 erläutert durch Fehlerfortpflanzung der statistischen Verteilung der Werte der Stromdichte ergibt.

Um den Vergleich beider Simulationsprogramme quantifizieren zu können, habe ich eine Kurvenanpassung gemäß Gleichung (3.1) an die Temperaturabhängigkeit der Mobilität durchgeführt. Die sich ergebenden Anpassungsparameter sind in Tabelle 5.1 aufgelistet.

Tabelle 5.1 Parameter der Anpassung an die Daten aus Abbildung 5.21 mittels Arrheniusfunktion nach Gleichung (3.1).

	Vielteilchensimulation Teil 2		Bumblebee mit R_c = 40 nm	
U_G (V)	μ_∞ (10^{-4} cm^2 V^{-1} s^{-1})	E_A (meV)	μ_∞ (10^{-4} cm^2 V^{-1} s^{-1})	E_A (meV)
1	7.1 ± 1.3	96 ± 3	7.5 ± 0.9	102.4 ± 1.8
2	4.9 ± 0.6	88.1 ± 1.7	6.6 ± 0.8	98.1 ± 1.8
5	3.14 ± 0.18	78.1 ± 0.8	4.4 ± 0.3	88.6 ± 1.1
10	2.57 ± 0.12	75.7 ± 0.7	2.63 ± 0.14	76.7 ± 0.7
20	1.79 ± 0.05	67.9 ± 0.4	1.88 ± 0.08	69.0 ± 0.6
50	0.90 ± 0.05	53.3 ± 0.8	1.05 ± 0.05	59.2 ± 0.6

Anhand der Werte der Anpassungsparameter ist ebenfalls eine gute Übereinstimmung erkennbar, wenn sich auch die Fehlerintervalle der Anpassungsparameter nicht bei allen Gatespannungen überlappen.

Abschließend lässt sich somit sagen, dass mit der Software Bumblebee und auch meiner eigenen Vielteilchensimulation Teil 2 übereinstimmende Ergebnisse für Ladungstransportsimulationen erzielt werden können. Bei der Verwendung der Software Bumblebee ist auf einen ausreichend großen Cut-off-Radius bei der Berechnung der Coulombwechselwirkung zu achten. Ich habe mich in den folgenden Kapiteln für die Verwendung der Software Bumblebee entschieden, da damit bereits seit dem Zeitpunkt, zu dem sie mir zur Verfügung stand, eine einsatzbereite Software vorlag, die bereits in der wissenschaftlichen Gemeinschaft erfolgreich zum Einsatz kam.[67,72,104–107] Hinzu kommen die Vorteile durch eine langjährige Entwicklung durch die Firma Simbeyond Ltd., wodurch der Programmcode hinsichtlich der Rechenzeit optimiert ist. Ein weiterer Grund für die Verwendung von Bumblebee liegt in der unlimitierten Rechenzeit einer Simulation im Vergleich zur Infrastruktur, die mir im Rahmen meiner Promotion an der Universität Bayreuth zur Verfügung stand. Trotz der vielen Vorteile der Software Bumblebee bietet meine eigene Entwicklung den Vorteil, dass sie kostenfrei zur Verfügung steht und der Programmcode leicht angepasst und erweitert werden kann, wodurch sie insbesondere für zukünftige Arbeiten weiterhin interessant bleiben wird.

5.4. Erstellen der Morphologie

In der Simulation wird ein kubisches Punktgitter verwendet, welches die lokalisierten Plätze des organischen Halbleiters darstellt, auf denen sich die Ladungsträger durch einen inkohärenten Hüpfprozess bewegen können. Beispielsweise ist jeder Gitterpunkt ein Molekül oder eine Wiederholeinheit eines Polymers. Um dem Halbleiter eine Morphologie zu geben, habe ich Gitterpunkte zu zusammenhängenden Bereichen gruppiert. Die Bereiche zeichnen sich dadurch aus, dass für Sprünge der Ladungsträger innerhalb dieses Bereiches eine andere Sprungrate als für Sprünge zwischen Bereichen gilt. Die Zuordnung der Gitterpunkte zu einem zusammenhängenden Bereich erfolgt dabei über eine Nummer für jeden Gitterplatz. Haben Gitterpunkte eine gleiche Nummer, so handelt es sich dabei um denselben Bereich (beispielsweise denselben Kristallit oder dieselbe Polymerkette). Im Folgenden stelle ich die beiden Methoden zur Morphologieerstellung vor, die ich für die Untersuchung des Einflusses der Morphologie auf den Ladungstransport in OFETs verwendet habe.

5.4.1. Polykristalline Struktur

Ein polykristalliner Halbleiter besteht aus kristallinen Bereichen, die sich jedoch in der Vorzugsrichtung der einzelnen Kristallite unterscheiden können. Die Bereiche, in denen die Kristallite einander berühren, werden als Korngrenzen bezeichnet.

Für die Erstellung einer solchen Morphologie habe ich einen Algorithmus in Anlehnung an den von Vladimirov et al. verwendet.[38] Bei diesem wird Kristallitwachstum dadurch modelliert, indem zunächst eine bestimmte Anzahl an Gitterpunkten in der x-y-Ebene als Keime für die Kristallite bestimmt werden, die sich in gleichem Abstand zueinander befinden. Von diesen Keimen ausgehend wachsen die Kristallite auf einem zweidimensionalen Gitter. Jeder Keim erhält zunächst einen zufälligen Richtungsvektor in der x-y-Ebene, der die Orientierung der Kristallite mit einer monoklinen Einheitszelle darstellt. Dabei ist die Form der Einheitszelle für jeden Keim die gleiche. Das Kristallitwachstum findet schrittweise statt. Bei jedem Schritt wachsen die Kristallite gleichmäßig in x- und y-Richtung unter Berücksichtigung der Form und Ausrichtung der Einheitszelle. Die Kristallite wachsen, bis sie einen gewissen vordefinierten Abstand zwischen den Kristalliten erreichen. Dieser Abstand wird als die Breite der Korngrenzen bezeichnet. Das Wachstum hört auf, wenn alle möglichen Gitterplätze zugewiesen wurden. Anschließend wird die zweidimensionale Struktur identisch in z-Richtung verschoben, sodass eine säulenartige

Struktur entsteht, die von den Korngrenzen unterbrochen wird. Abbildung 5.22(a-d) zeigt verschieden Schritte während des Kristallitwachstums, wobei (a) die Keime zu Beginn und (d) die endgültige Morphologie darstellt.

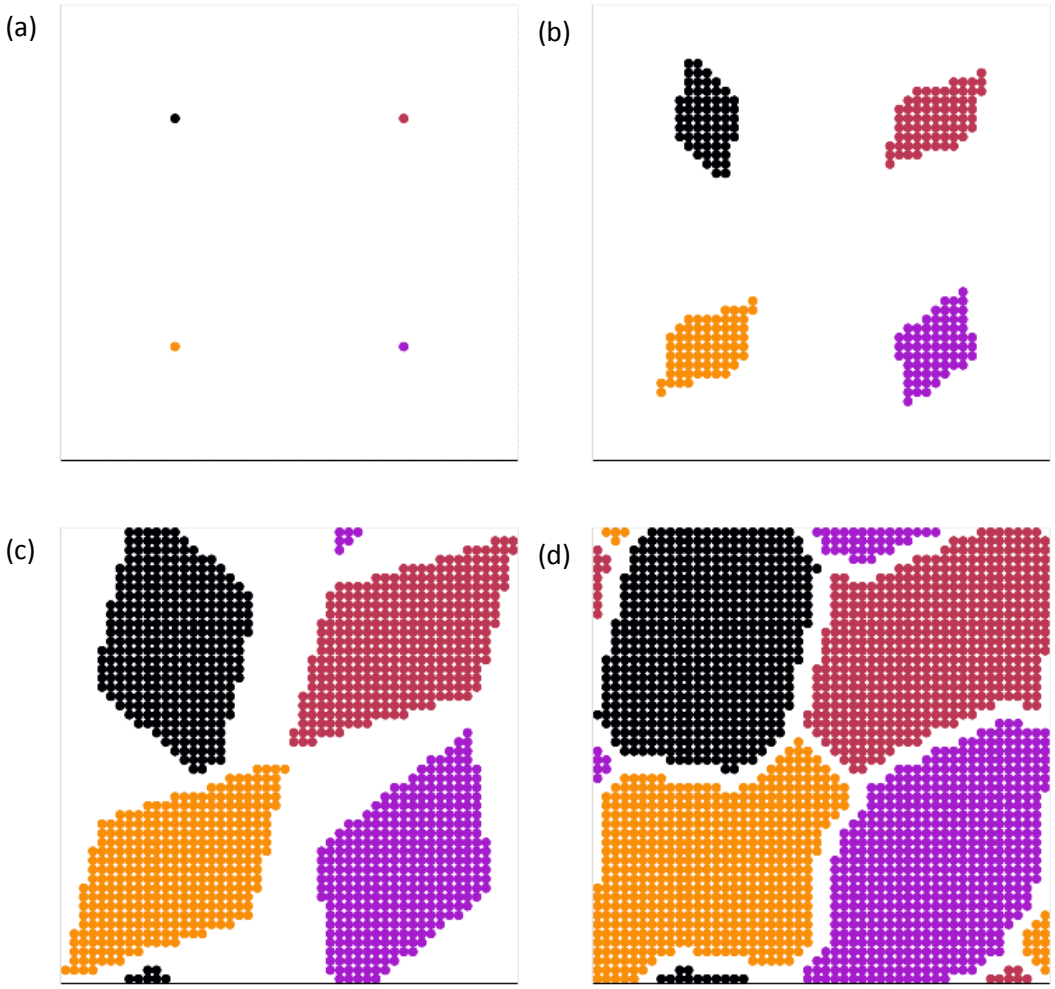

Abbildung 5.22 Schematische Darstellung einer zweidimensionalen polykristallinen Morphologie. (a-d) Einzelne Zwischenschritte während dem Wachstum der Kristallite auf einem zweidimensionalen Gitter. Aus Ref.[108]

5.4.2. Konjugierte Oligomere

Ein Oligomer ist ein Molekül, dass durch die Aneinanderreihung einer Wiederholeinheit entsteht. Typischerweise besteht dabei ein solches Molekül aus bis zu 20 Wiederholeinheiten.[17] Die Morphologie, die sich am einfachsten mittels eines Algorithmus erzeugen lässt, besteht aus Oligomeren, die alle entlang einer Vorzugsrichtung ausgerichtet sind.

Für die Untersuchung von Ladungstransport in Oligomeren habe ich einen Algorithmus zur Erzeugung einer Morphologie entwickelt. Der Algorithmus, der in Abbildung 5.23 dargestellt ist, läuft wie folgt ab: Der Ausgangspunkt ist ein kubisches Punktgitter. Nebeneinander liegende Punkte auf dem Gitter werden zu linearen Bereichen zusammengefasst, die die Oligomere darstellen. Dabei sind die Oligomere in y- und z-Richtung (letztere ist in der Abbildung nicht dargestellt) zunächst parallel angeordnet. In einem letzten Schritt können benachbarte Oligomere in y- oder z-Richtung um einen gewünschten Betrag zueinander verschoben werden.

In der Simulation verwende ich für die Länge der Oligomere nur ganze Teiler der Größe der Simulationsbox in x-Richtung, sodass innerhalb einer Zeile in x-Richtung alle Oligomere die gleiche Länge haben. Die scheinbar kürzeren Oligomere am rechten oder linken Rand in Abbildung 5.23c werden durch periodische Randbedingungen in x-Richtung wieder aufgehoben (in der Abbildung nicht maßstabsgetreu dargestellt).

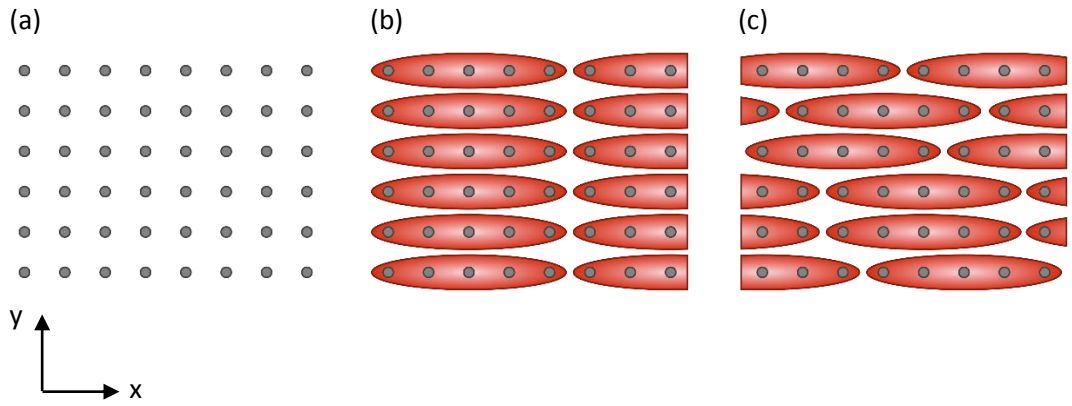

Abbildung 5.23 Schematische Darstellung der Erzeugung einer Morphologie mit Oligomeren in der x-y-Ebene. Die z-Richtung verhält sich dabei analog zur y-Richtung. Ein Punktgitter (a) wird zu parallel angeordneten linearen Ketten zusammengefasst (b), welche bei Bedarf einen Versatz zwischen benachbarten Ketten erhalten (c). Die roten Ellipsen stellen dabei die Oligomere dar.

Die so entstandenen Oligomere zeichnen sich dadurch aus, dass die Sprünge der Ladungsträger innerhalb eines Oligomers mit einer um einen bestimmten Faktor höheren Hüpfrate stattfinden. Diesen Faktor habe ich analog zu der Arbeit von Mendels und Tessler zu 100 gewählt.[86] Außerdem enthalten alle Gitterplätze eines Oligomers den gleichen Energiewert.

Da in realen Systemen häufig eine energetische Unordnung zu erwarten ist, habe ich außerdem einen Algorithmus entwickelt, mit dem ich die Energielandschaft bei einer Morphologie aus Oligomeren verändern kann. Mit diesem Algorithmus lässt sich eine annähernd gaußförmige Verteilung der Energien der Oligomere erreichen. Dies war nötig, da die Software Bumblebee zwar

eine Gaußverteilung der Energien innerhalb eines Materials zulässt, bei Verwendung einer eigenen Morphologie jedoch nicht die Morphologie mit berücksichtigt.

Die Idee hinter der Realisierung einer annähernd gaußförmigen Verteilung der Energien der Oligomere in der Simulation ist die Verwendung einer diskreten Gaußverteilung. Dazu habe ich eine Gaußverteilung in neun gleich große Intervalle aufgeteilt, wobei ich die Ränder so gewählt habe, dass das Integral der Gaußkurve innerhalb der Ränder 99 % des Integrals der gesamten Gaußkurve beträgt. Dies ist der Fall, wenn die Ränder das 2.576 fache der Standardabweichung vom Zentrum der Gaußverteilung entfernt gewählt werden. Für die Zustandsdichte des Materials bedeutet dies, dass 1 % der Zustände eine Energie hätten, die außerhalb der Intervalle liegt und somit nicht berücksichtigt werden. Dies erscheint mir eine vertretbare Näherung. Abbildung 5.24 zeigt die sich ergebende diskrete Gaußverteilung bei einem Histogramm mit neun Intervallen und einer Standardabweichung der Gaußverteilung von 50 meV.

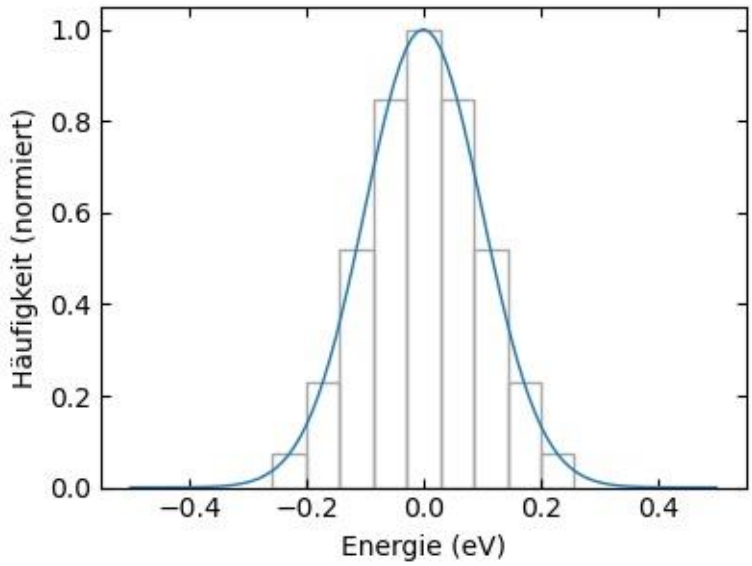

Abbildung 5.24 Gaußverteilung mit einer Standardabweichung von 50 meV und deren Histogramm. Die Ränder des Histogramms liegen bei dem ±2.576 fachen der Standardabweichung der Verteilung.

Mit Hilfe eines kleinen Programms berechne ich nun zuerst die Häufigkeit eines jeden Intervalls und die Energie in dessen Zentrum. Diese Häufigkeiten gebe ich dann als Parameter in das Skript für die Erstellung der Morphologie in Bumblebee. Dort muss dann für jedes Intervall ein Material mit der entsprechenden Energie hinterlegt sein. Bei der Erstellung der Morphologie wird dann jedem Oligomer zufällig eine Energie aus der diskreten Gaußverteilung zugewiesen, sodass die Energielandschaft dieser diskreten Gaußverteilung entspricht.

Ein Beispiel für solch eine Energielandschaft mit einer diskreten Gaußverteilung der Energien der Oligomere ist in Abbildung 5.25 dargestellt.

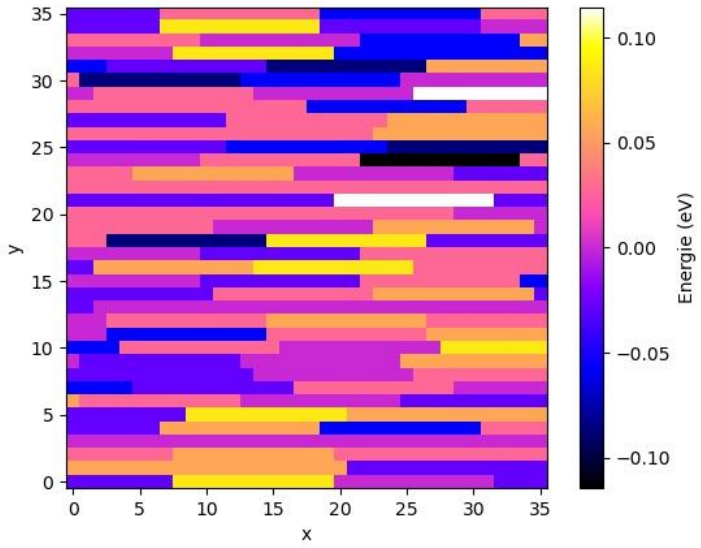

Abbildung 5.25 Beispiel einer Energielandschaft mit oligomerer Struktur. Dargestellt ist ein Ausschnitt aus einer Morphologie mit Oligomerlänge 12. Die Breite der diskreten Gaußverteilung der Energien der Oligomere beträgt 50 meV.

6. Der Einfluss von Korngrenzen auf den Ladungstransport in polykristallinen organischen Feldeffekttransistoren

Das vorliegende Kapitel basiert auf der Veröffentlichung Meier et al. in Advanced Optical Materials.[108]

6.1. Einleitung

Organische Feldeffekttransistoren (OFETs) haben innerhalb der letzten Jahre einen enormen technischen Fortschritt gemacht und erreichen mittlerweile Ladungsträgermobilitäten von über 10 cm^2 $V^{-1}s^{-1}$.[109,110] Das wurde ermöglicht durch ein erhöhtes theoretisches Verständnis über den Mechanismus des Ladungstransports.[4,5,49,50,111] Aber auch Fortschritte in der Synthese,[35,112,113] der Filmprozessierung[39,114] und der Bauteiloptimierung[115,116] haben zu diesem rapiden Wachstum in der Effizienz geführt. Obwohl sie bereits seit längerer Zeit untersucht werden, sind sie immer noch Gegenstand aktueller Forschung.[29,35–39] Eine wichtige Kenngröße für die Effizienz von elektronischen Bauteilen ist die Ladungsträgermobilität. Hohe Mobilitätswerte werden typischerweise durch eine hohe Kristallinität des ladungsführenden Materials erreicht. Bei der Herstellung von dünnen Schichten entsteht prozessbedingt jedoch eine polykristalline Struktur im Halbleiter. Beispiele solcher Herstellungsmethoden sind Rotationsbeschichtung oder Aufdampfverfahren. Die Korngrenzen, die bei diesen Herstellungsmethoden entstehen, wirken sich im Allgemeinen negativ auf den Ladungstransport aus.[29–34] Der genaue Mechanismus ist jedoch noch nicht vollständig erforscht. Mit Hinblick auf die Sorgfalt, die beim Herstellungsprozess möglicherweise erforderlich ist, ist es somit wichtig diese Frage näher zu klären.

Korngrenzen können entweder Fallen oder Barrieren für die Ladungsträger sein. Das bedeutet, dass die Energie in den Zuständen der Korngrenzen im Mittel niedriger oder höher als die in den kristallinen Bereichen liegt. Bei niedrigeren Energien handelt es sich um Fallenzustände, bei höheren um Barrieren. In organischen Halbleitern, bei denen Fallenzustände durch Dotierung

erzeugt werden, können flache Fallen und niedrige Barrieren die Mobilität stark verringern, wohingegen tiefe Fallen und hohe Barrieren nur einen geringen Effekt haben.[93,94] In polykristallinen Systemen ist die Situation jedoch anders, da hier innerhalb der Korngrenzen zusammenhängende Pfade existieren. Ob Korngrenzen Fallen- oder Barrierenzustände darstellen, kann einen unterschiedlichen Effekt auf die Ladungsträgermobilität haben. So wird beispielsweise in PDI1MPCN2, einem kleinmolekularen, polykristallinen organischen Halbleiter, beobachtet, dass Barrieren in den Korngrenzen den Ladungstransport dominieren.[38]

Sowohl Fallen als auch Barrieren in den Korngrenzen wurden experimentell beobachtet. Fallen wurden durch Kelvinsondenkraftmikroskopie,[117–120] zeitaufgelöste elektrostatische Rasterkraftmikroskopie[121] und durch einen Vergleich der Zustandsdichte von Fallen für Löcher[30] nachgewiesen. Barrierenzustände in Korngrenzen wurden mittels leitfähige Rasterkraftmikroskopie nachgewiesen.[122] Außerdem wird argumentiert, dass durch Ladungsträger, die sich in Fallenzuständen in Korngrenzen ansammeln, diese letztendlich als Barrieren für den Ladungstransport wirken.[33,45,46]

Aus theoretischer Sicht können Korngrenzen beides sein, sowohl Fallen als auch Barrieren. Elektronische Strukturberechnungen haben ergeben, dass Korngrenzen entweder aus Fallen[123] oder Barrieren[124] bestehen, Energielandschaftsberechnungen aus Molekulardynamik weisen auf Barrieren hin[92] und eine Argumentation aus der Bandstruktur in organischen Halbleitern ergibt ebenfalls, dass Korngrenzen Barrieren darstellen.[125] Zudem können sogar im selben System beide Arten in den Korngrenzen auftreten.[38,92,126]

In diesem Kapitel untersuche ich beides, sowohl Fallen als auch Barrieren in den Korngrenzen. Neben der Energie in den Korngrenzen untersuche ich den Einfluss der Breite der Korngrenzen, die implizite Korngröße, sowie den Anteil der Korngrenzen am gesamten Halbleiter auf den Ladungstransport in polykristallinen organischen Feldeffekttransistoren. Dies geschieht durch kinetische Monte-Carlo-Simulation mit einem einfachen Algorithmus zur Erzeugung einer polykristallinen Morphologie, die typisch für kleine organische Moleküle ist. Aus der Temperaturabhängigkeit der Ladungsträgermobilität und der Verteilung der Ladungsträger im Halbleiter kann daraufhin auf den Ladungstransportmechanismus geschlossen werden.

6.2. Verwendete Simulationsparameter

Mit der in Kapitel 5.3 beschriebenen Software Bumblebee habe ich Ladungstransport simuliert. Damit habe ich einen Feldeffekttransistor wie in Abbildung 6.1 gezeigt modelliert.

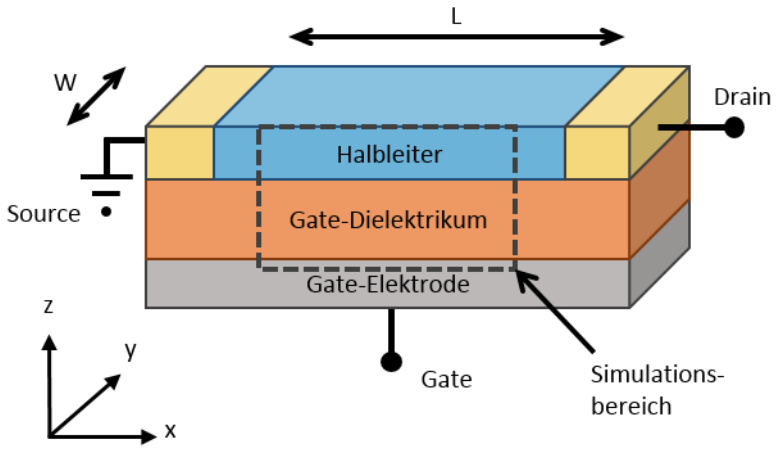

Abbildung 6.1 Schematische Darstellung eines Feldeffekttransistors.

Die Größe der Simulationsbox beträgt 120 x 120 x 10 Gitterpunkte in x-, y- und z-Ausdehnung bei einer Gitterkonstante von 1 nm. Die Anzahl der Gitterpunkte in x- und y-Richtung ist dabei möglichst groß bei gleichzeitig akzeptablem Rechenaufwand zu wählen, um störende Effekte durch eine zu geringe statistische Verteilung der Energiewerte der Gitterplätze zu vermeiden. Dies scheint durch die vorliegende Wahl ausreichend zu sein. In z-Richtung hingegen sind die zehn Lagen ausreichend, da der Ladungstransport in Feldeffekttransistoren vor allem in den untersten Lagen stattfindet.[102] Wie bereits beschrieben liegen in x- und y-Richtung periodische Randbedingungen vor. Der Spannungsunterschied in Source-Drain-Richtung U_{SD} beträgt 0.72 V über die gesamte Länge von 120 Gitterpunkten, was bei einem Gitterabstand von 1 nm einem elektrischen Feld von $6 \cdot 10^4$ V cm^{-1} entspricht, ein typischer Wert für den Betriebsbereich von Feldeffekttransistoren. Die Spannung an der Gateelektrode beträgt U_G = 5 V, was die Voraussetzung für die Simulation im linearen Bereich erfüllt. Bei dieser Gatespannung liegt eine Ladungsträgerkonzentration von 0.011 pro Gitterpunkt vor (nach Gleichung (3.6)), falls alle Ladungsträger in der untersten Lage konzentriert sind. Die dielektrische Konstante beträgt 4 in der Halbleiterschicht sowie im Dielektrikum, was ein typischer Wert für organische Halbleiter[100] und ähnlich zum Wert für Siliziumdioxid ist, das häufig als Material für das Dielektrikum verwendet wird. Dabei beträgt die Dicke des Dielektrikums 100 nm.

In der verwendeten Hüpfrate nach Miller-Abrahams (siehe Kapitel 5.3)[99]

$$k_{ij} = \nu_0 \exp(-2\gamma|r_{ij}|) \begin{cases} \exp\left(-\frac{\Delta E_{ij}}{k_B T}\right) & \Delta E_{ij} > 0 \\ 1 & \Delta E_{ij} \leq 0 \end{cases} \quad (6.1)$$

beträgt die attempt-to-hop Frequenz ν_0 = 10^{13} s^{-1} für Sprünge zwischen Kristalliten und innerhalb von Korngrenzen und ν_0 = 10^{15} s^{-1} für Sprünge innerhalb der Kristallite und die inverse Lokalisierungslänge γ = 5 nm^{-1}. Sprünge der Ladungsträger werden bis zum übernächsten Nachbar in jeder Richtung berücksichtigt.

Für die Morphologie der Halbleiterschicht verwende ich den Algorithmus aus Kapitel 5.4.1. Ich untersuche damit polykristalline Strukturen mit unterschiedlicher Breite der Korngrenzen, sowie unterschiedlicher Anzahl an Kristalliten, was direkt mit deren Größe korreliert. Die Breite der Korngrenzen variiert von 1 über 3 nach 5 nm. Für die Anzahl der Kristallite wird 3x3, 7x7 und 10x10 verwendet. Dies ist in Abbildung 6.2 dargestellt. Dabei ergeben sich mittlere Korngrößen von 7 bis 39 nm, je nach Korngrenzenbreite und Anzahl der Kristallite, wie in Tabelle 6.1 aufgelistet ist.

Tabelle 6.1 Mittlere Korngrößen berechnet unter der Annahme von kubischen Kristalliten. Die Korngröße beschreibt dabei die Kantenlänge der Kristallite.

		d_{GB} (nm)		
		1	3	5
N_{seeds}	9	39	37	35
	49	16	14	12
	100	11	9	7

Für die Energien in den Kristalliten und Korngrenzen nehme ich jeweils Gaußverteilungen an, wobei die energetische Unordnung der Standardabweichung der jeweiligen Verteilung entspricht. Die energetische Unordnung in den Kristalliten habe ich zu σ_{cryst} = 10 meV gewählt. Dies ist etwas größer als typische intermolekulare Phononenenergien, die im Bereich von 5 meV liegen.[5] Für die Korngrenzen hingegen habe ich einen Wert von σ_{GB} = 50 meV gewählt, was eine höhere energetische Unordnung in den Korngrenzen widerspiegelt. Das Verhältnis der energetischen Unordnungen in den Kristalliten und Korngrenzen entspricht dabei dem, das Mohan et al. für die Simulation von inhomogenen organischen Halbleitern verwendet haben.[90] Die Zentren der beiden Energieverteilungen können sich dabei unterscheiden. In der vorliegenden Untersuchung wird die mittlere Energie der Kristallite auf null gesetzt, sodass die

Differenz der beiden Verteilungen der mittleren Energie der Korngrenzen entspricht. Im Folgenden wird diese Differenz als *Energie der Korngrenzen* bezeichnet. Sie wird im Bereich -0.6 bis 0.4 eV verändert. Die Werte für die energetischen Unordnungen σ_{cryst} und σ_{GB} bleiben im Folgenden unverändert.

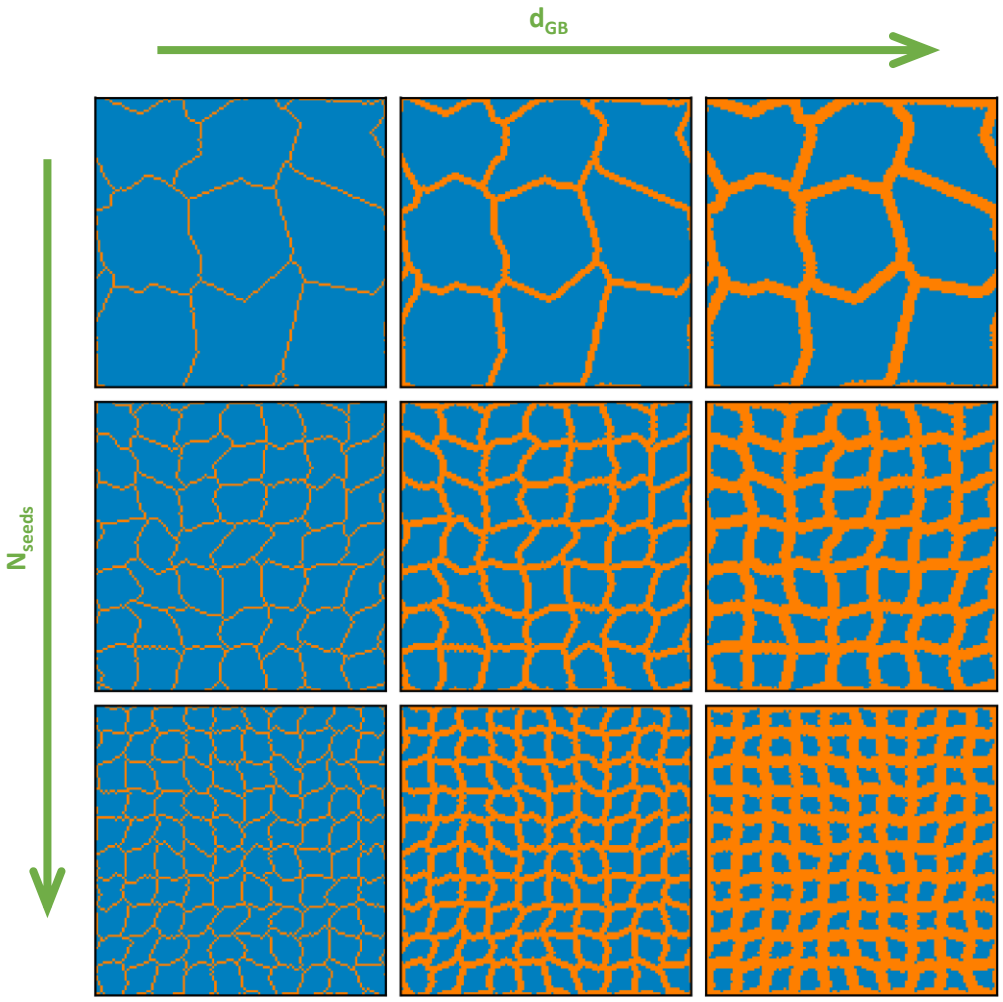

Abbildung 6.2 Verwendete Morphologien der Halbleiterschicht. Mit blau werden kristalline Bereiche dargestellt, die Korngrenzen sind orange. Jede Schicht in z-Richtung des Halbleiters entspricht dabei der gezeigten Morphologie. Von links nach rechts nimmt die Breite der Korngrenzen von 1 nach 3 nach 5 nm zu, von oben nach unten steigt die Anzahl der Startpunkte für Kristallitwachstum von 3x3 nach 7x7 nach 10x10.

6.3. Ergebnisse der Ladungstransportsimulation

Im Folgenden werden die Ergebnisse der Monte-Carlo-Simulation dargestellt. Zunächst untersuche ich die Ladungsträgermobilität für verschiedene Morphologien als Funktion der Energie in den Korngrenzen. Anschließend analysiere ich deren Temperaturabhängigkeit und schließlich betrachte ich den Einfluss des Anteils der Korngrenzen für den Fall, dass die Energie in den Korngrenzen im Mittel der in den Kristalliten entspricht.

6.3.1. Ladungsträgermobilität als Funktion der Energie der Korngrenzen

In Abbildung 6.3 ist die nach Gleichung (3.5) berechnete Ladungsträgermobilität als Funktion der Energie in den Korngrenzen bei Raumtemperatur dargestellt. Um eine bessere Vergleichbarkeit zu erzielen, sind alle Werte auf den Wert der Mobilität einer einkristallinen Morphologie mit σ_{cryst} = 10 meV normiert. Verschiedene Bereiche der Energien in den Korngrenzen sind mit unterschiedlichen Farben hinterlegt, wie noch später im Text erläutert wird.

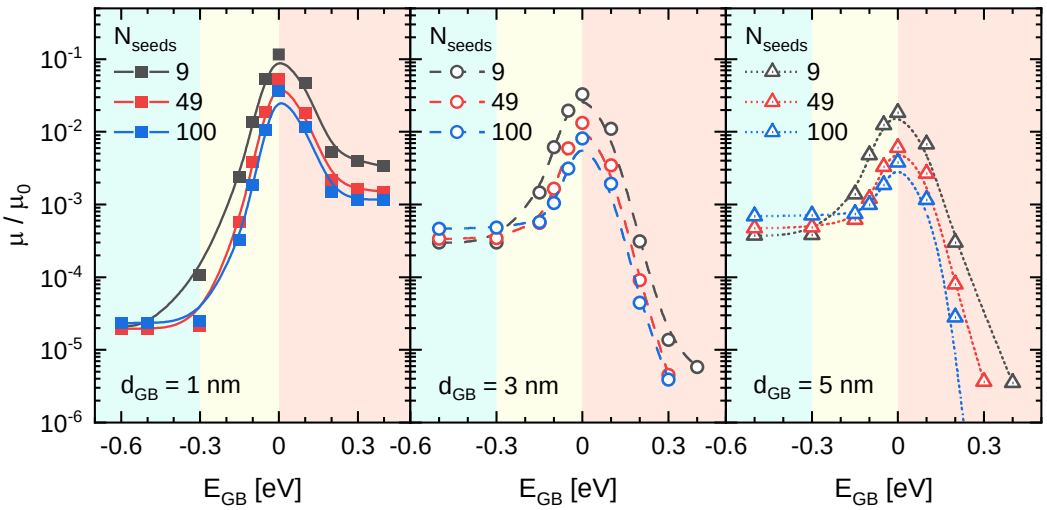

Abbildung 6.3 Berechnete Ladungsträgermobilität aus Monte-Carlo-Simulationen als Funktion der Energie der Korngrenzen für verschiedene Anzahl an Kristalliten N_{seeds} wie angegeben. Die Breite der Korngrenzen variiert von links nach rechts wie angegeben. Die Temperatur beträgt jedes Mal 300 K. Die Linien dienen dabei nur der Veranschaulichung. Mit den verschiedenen Hintergrundfarben werden unterschiedliche Ladungstransportbereiche hervorgehoben; eine genauere Beschreibung findet sich im Text. Aus Ref.[108]

Die Mobilität nimmt ein Maximum an, wenn die mittlere Energie in den Korngrenzen und Kristalliten gleich ist, das heißt für E_{GB} = 0 eV. In beide Richtungen, für größere und kleinere Energie der Korngrenzen, fällt die Mobilität je nach Breite der Korngrenzen ab. Ich betrachte nun zunächst das Verhalten der Mobilität zu kleineren Energien der Korngrenzen hin. Hierbei fällt die Mobilität um einige Größenordnungen ab, wobei der Abfall mit der Breite der Korngrenzen abnimmt und ebenfalls von der Anzahl an Kristalliten abhängig ist. Dies lässt sich übersetzen in einen relativen Anteil der Korngrenzen, wobei breitere Korngrenzen und eine größere Anzahl an Kristalliten einen höheren relativen Anteil der Korngrenzen bewirken. Für die breitesten Korngrenzen und die meisten Kristallite ist der Abfall in der Mobilität am geringsten. Es zeigt sich außerdem, dass die Werte der Mobilität unterhalb von E_{GB} ≤ -0.3 eV, was in etwa einer Energie von 12 k_BT für T = 300 K entspricht, konstant sind. Der Wert der Mobilität ist dabei von der Breite der Korngrenzen abhängig, wobei sie sich von d_{GB} = 1 nm nach 3 nm um einen Faktor von 20 und von d_{GB} = 3 nm nach 5 nm nur noch um einen Faktor von maximal 1.5 erhöht. Außerdem steigt die Mobilität in diesem Bereich leicht mit der Anzahl der Kristallite. Oberhalb von E_{GB} > -0.3 eV ist es genau umgekehrt, hier sinkt die Mobilität durchgängig mit der Anzahl der Kristallite, das bedeutet mit kleiner werdenden Kristalliten. Für positive Werte der Korngrenzen, das heißt für Barrieren, fällt die Mobilität exponentiell ab und nimmt im Fall von d_{GB} = 1 nm einen konstanten Wert an.

6.3.2. Temperaturabhängigkeit der Ladungsträgermobilität

Als nächstes untersuche ich die Temperaturabhängigkeit der Ladungsträgermobilität für vier verschiedene Morphologien, nämlich N_{seeds} = 9 oder 100 und d_{GB} = 1 oder 5 nm. Der Temperaturverlauf der Mobilität im Bereich T = 100 bis 400 K ist in Abbildung 6.4 gezeigt, wobei ich eine Arrheniusdarstellung, eine halblogaritmische Darstellung gegen die reziproke Temperatur, gewählt habe. Dies dient dazu, eventuelle Abhängigkeiten der Form $\propto \exp(-E_A/k_BT)$ leicht erkennen zu können. Die Daten habe ich anschließend mit einer Arrheniusfunktion angepasst nach der Form:

$$\mu(T) = \mu_\infty \cdot \exp\left(-\frac{E_A}{k_BT}\right) \tag{6.2}$$

wobei E_A die Aktivierungsenergie ist, was in der gewählten Darstellung eine Gerade liefert.

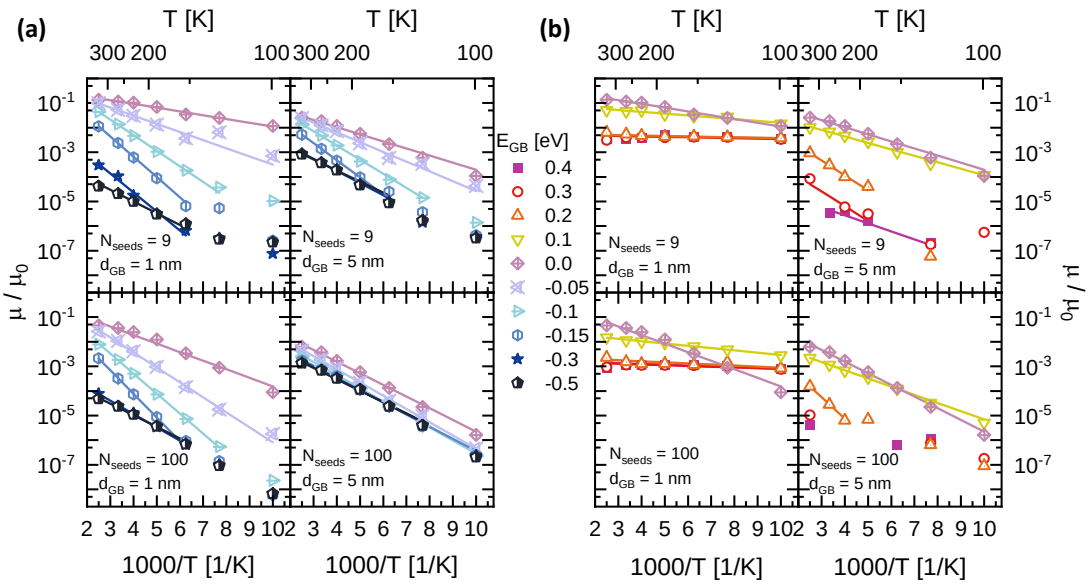

Abbildung 6.4 Temperaturverlauf der Ladungsträgermobilität aus der Monte-Carlo-Simulation für vier verschiedene Morphologien wie angegeben. Der besseren Lesbarkeit halber sind die Energien in den Korngrenzen in (a) negative und (b) positive Werte aufgeteilt (jeweils inklusive E_{GB} = 0 eV). Die Geraden stellen Arrheniusfits dar wie im Text beschrieben und sind in dem Bereich, in dem sie angewendet wurden, gezeichnet. Aus Ref.[108]

Betrachtet werden zunächst negative Energiewerte der Korngrenzen. Im Fall der Morphologie mit den breitesten Korngrenzen und kleinsten Kristalliten, d_{GB} = 5 nm und N_{seeds} = 100, liegt für alle Energien in den Korngrenzen eine arrheniusförmige Abhängigkeit vor. Die Aktivierungsenergie befindet sich dabei im Bereich 91 bis 106 meV für E_{GB} = -0.5 bis -0.05 eV. Für den Fall d_{GB} = 1 nm und N_{seeds} = 9 zeigt sich ebenfalls ein arrheniusförmiges Verhalten. Die Aktivierungsenergie ist dabei 95 meV für E_{GB} = -0.5 eV, durchläuft ein Maximum von 164 meV bei E_{GB} = -0.15 eV und nimmt dann wieder ab. Die beiden übrigen Morphologien zeigen ein ähnliches Verhalten, wobei sie sich zwischen den beiden beschriebenen Extremfällen bewegen.

Für positive Werte der Energie in den Korngrenzen weist die Temperaturabhängigkeit im Fall von d_{GB} = 1 nm für E_{GB} = 0.1 eV eine relativ niedrige Aktivierungsenergie von 15 bzw. 19 meV für große (N_{seeds} = 9) bzw. kleine (N_{seeds} = 100) Kristallite auf, die für höhere Energien in den Korngrenzen noch weiter abnimmt. Im Fall von dicken Korngrenzen fällt die Mobilität stark mit der Energie in den Korngrenzen bei allen Temperaturen ab.

Abbildung 6.5 zeigt die Werte der Aktivierungsenergien, die durch Anpassung der Daten aus Abbildung 6.4 mittels Arrheniusfunktionen bestimmt wurden.

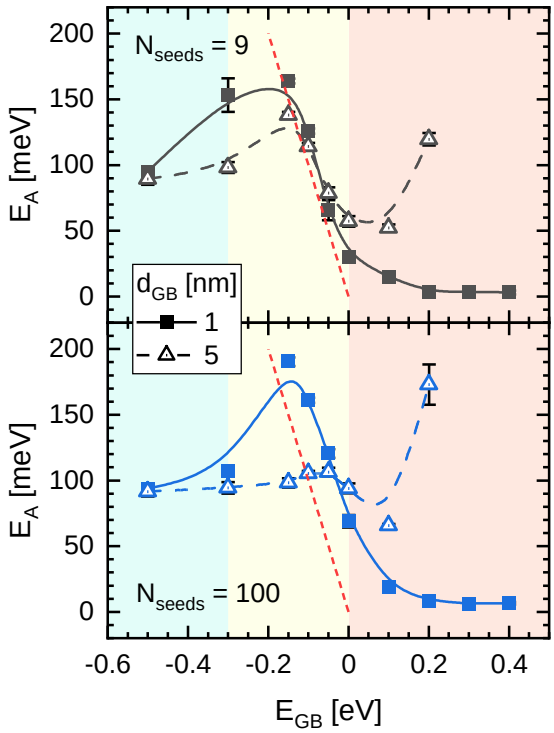

Abbildung 6.5 Aktivierungsenergien der Ladungsträgermobilität, die durch Anpassung der Daten mittels Arrheniusfunktionen bestimmt wurden, für verschiedene Morphologien wie angegeben. Der Linienstil und die Symbole bestimmen dabei die Breite der Korngrenzen. Die rot gestrichelte Linie besitzt eine Steigung von -1 und dient der Veranschaulichung. Ebenso sind die Trendlinien nur als Orientierungshilfe gedacht. Die Hintergrundfarbe symbolisiert die verschiedenen Transportbereiche, wie im Text näher erläutert wird. Aus Ref.[108]

Erkennbar ist hier das bereits vorgestellte Verhalten der Aktivierungsenergie. Das Maximum der Aktivierungsenergie tritt dabei bei E_{GB} = -0.15 eV auf ist am stärksten ausgeprägt für d_{GB} = 1 nm. Bei E_{GB} = -0.5 eV nähert sich die Aktivierungsenergie asymptotisch etwa einem Wert von (92 ± 2) meV an. Für den Fall von breiten Korngrenzen weicht die Aktivierungsenergie für negative Energien in den Korngrenzen nur geringfügig von diesem asymptotischen Wert ab.

6.3.3. Einfluss des Anteils der Korngrenzen auf die Ladungsträgermobilität

Um besser zu verstehen, wie sich die Kristallite und die Korngrenzen auf den Ladungstransport auswirken, ist es hilfreich die Temperaturabhängigkeit der Ladungsträgermobilität für Morphologien mit unterschiedlichem Anteil an Korngrenzen zu untersuchen. In Abbildung 6.6

ist dies gezeigt für acht verschiedene Morphologien. Die Energie in den Korngrenzen beträgt dabei immer E_{GB} = 0 eV. Dies dient dazu, den Einfluss von Kristalliten und Korngrenzen möglichst isoliert zu betrachten.

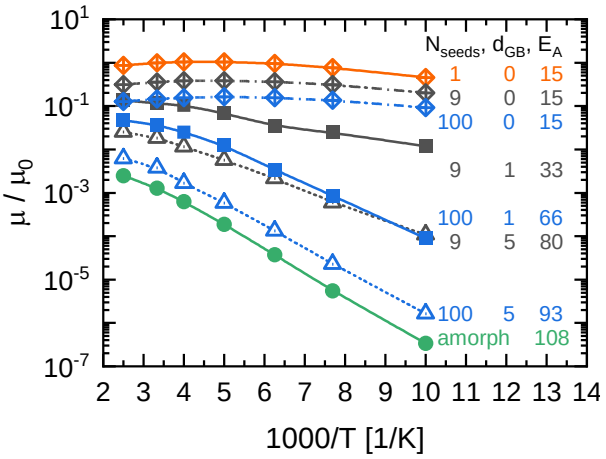

Abbildung 6.6 Temperaturabhängigkeit der Ladungsträgermobilität für Morphologien mit unterschiedlichem Anteil an kristalliner Phase. Die Energie der Korngrenzen beträgt dabei immer E_{GB} = 0 eV. Die Anzahl der Kristallite, die Breite der Korngrenzen sowie die aus Arrheniusfits ermittelten Aktivierungsenergien sind für jede Kurve angegeben. Der Fall N_{seeds} = 1 (orange Kurve) entspricht einem Einkristall, das heißt es liegen auch keine Korngrenzen vor. Die grüne Kurve ist für den Fall einer vollständig amorphen Morphologie. Für die Kurven dazwischen gibt die Farbe die Anzahl der Kristallite an, schwarz bedeutet N_{seeds} = 9, blau N_{seeds} = 100, und die Symbole die Dicke der Korngrenzen, gekreuzte Rauten haben d_{GB} = 0 nm, ausgefüllte Quadrate d_{GB} = 1 nm und gepunktete Dreiecke d_{GB} = 5 nm. Die Linien dienen nur als Orientierungshilfe. Aus Ref.[108]

Die Temperaturabhängigkeit zeigt für alle Kurven einen arrheniusförmigen Verlauf und weicht nur bei hohen Temperaturen leicht davon ab. Auf Letzteres gehe ich in Kapitel 8 noch näher ein. Die Aktivierungsenergien für die Fälle ohne Korngrenzen sind identisch bei 15 meV, was mit der energetischen Breite der Kristallite vergleichbar ist. Die Mobilität verringert sich durch das Einfügen von zusätzlichen Unterbrechungen der Kristallite, da in der Simulation die Sprünge innerhalb von Kristalliten um einen Faktor von 100 wahrscheinlicher sind. Der Wert der Mobilität der einkristallinen Morphologie bei T = 300 K wurde für alle Mobilitäten als Normierungswert verwendet. Für den vollständig amorphen Fall beträgt die Aktivierungsenergie 108 meV. Zwischen diesen beiden Extremfällen nimmt die Aktivierungsenergie mit steigendem Anteil an Korngrenzen zu, wie in der Tabelle 6.2 weiter unten ersichtlich ist.

6.4. Diskussion

Anhand der dargestellten Ergebnisse sollen nun Aussagen über die Ladungstransportmechanismen im polykristallinen Halbleiter unter dem Einfluss der Parameter Energie, Breite und relativem Anteil der Korngrenzen getroffen werden. Dabei ziehe ich sowohl die räumliche Verteilung der Ladungsträger im Halbleiter als auch deren energetische Lage innerhalb der Zustandsdichte mit in Betracht, bevor ich drei verschiedene Transportbereiche in Abhängigkeit von der Energie der Korngrenzen identifiziere.

6.4.1. Ladungsträgerverteilung in der untersten Schicht des Halbleiters

Zunächst untersuche ich die Ladungsträgerkonzentration in der untersten Schicht des Halbleiters für verschiedene charakteristische Energien der Korngrenzen. Für vier Morphologien ist diese Konzentration in Abbildung 6.7 dargestellt. Gezeigt wird für jede Morphologie Energien der Korngrenzen von E_{GB} = -0.5, -0.1, 0 und 0.1 eV. Die Bilder zeigen die mittlere Besetzung der Gitterplätze über die gesamte Simulation hinweg, das heißt, dass bei einem Wert von 1 dieser Platz für die gesamte Simulationszeit mit einem Ladungsträger besetzt ist.

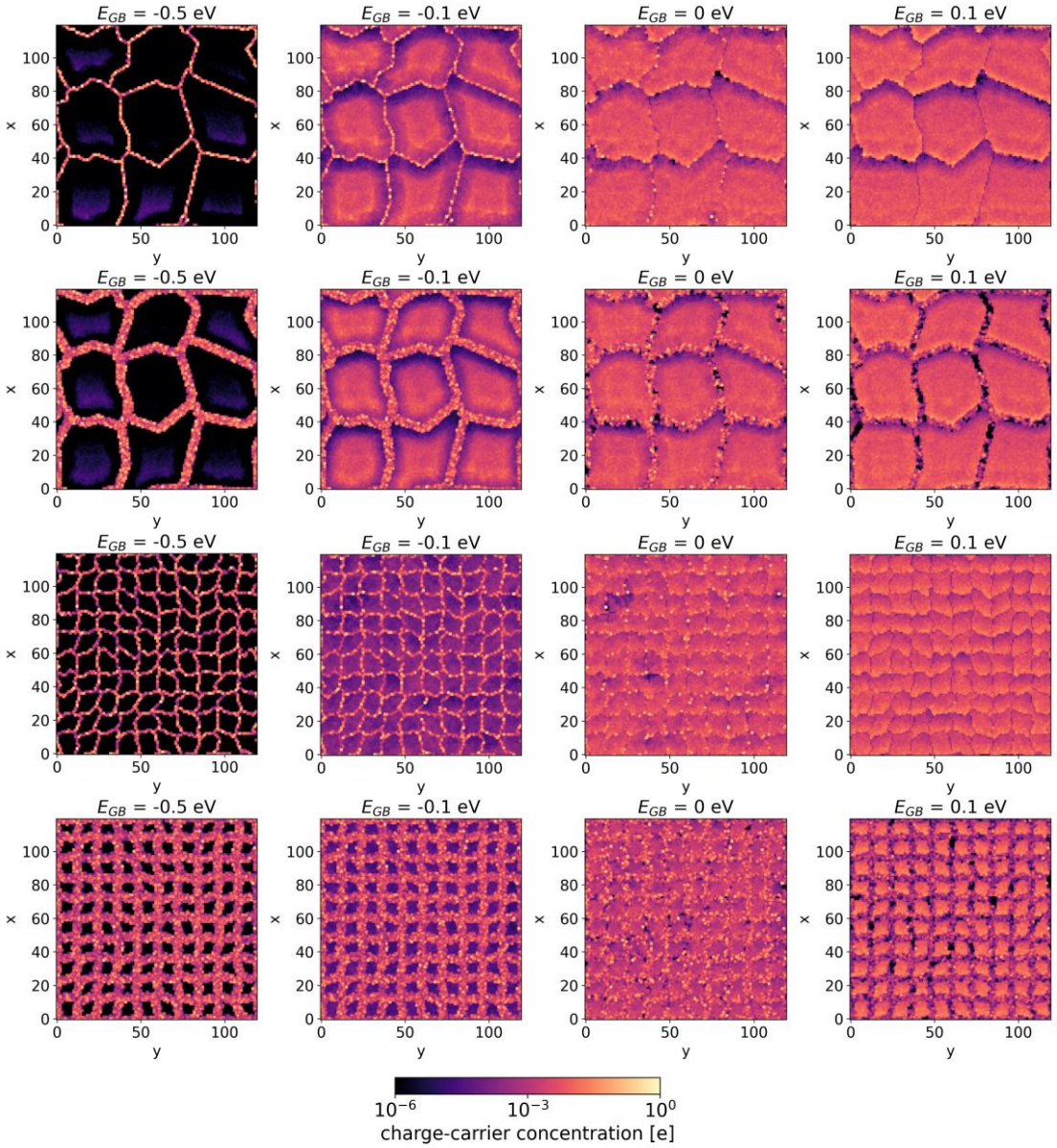

Abbildung 6.7 Ladungsträgerdichte in der untersten Schicht des Halbleiters für vier verschiedene Energien der Korngrenzen wie angegeben bei T = 300 K. Die Morphologien von oben nach unten sind: 1. Zeile d_{GB} = 1 nm, N_{seeds} = 9, 2. Zeile d_{GB} = 5 nm, N_{seeds} = 9, 3. Zeile d_{GB} = 1 nm, N_{seeds} = 100, 4. Zeile d_{GB} = 5 nm, N_{seeds} = 100. Der Farbcode gibt die Ladungsträgerkonzentration in Einheiten der Elementarladung an, wobei gelb und orange Bereiche hoher Konzentration darstellen. Aus Ref.[108]

Angefangen bei E_{GB} = -0.5 eV, was bei T = 300 K etwa 20 k_BT entspricht, befinden sich zunächst so gut wie alle Ladungsträger in den Korngrenzen bei allen vier Morphologien. Bei E_{GB} = -0.1 eV ist die Konzentration an Ladungsträgern in den Korngrenzen immer noch hoch, jedoch befinden sich auch Ladungsträger in den Kristalliten. Zu beachten ist hierbei jedoch die logarithmische Skalierung, sodass die Konzentration in den Korngrenzen noch deutlich höher ist als in den Kristalliten. E_{GB} = -0.1 eV entspricht dabei etwa 4 k_BT bei T = 300 K. Für E_{GB} ≤ -0.1 eV ist die

Konzentration der Ladungsträger in den Kristalliten bei kleineren Kristalliten (N_{seeds} = 100) geringer als bei größeren (N_{seeds} = 9). Dies kann durch die Coulombabstoßung durch die Ladungsträger in den Korngrenzen erklärt werden, welche eine erhöhte Konzentration an Ladungsträgern in den Kristalliten verhindern. In der Simulation wird die Coulombwechselwirkung bis zu einem Radius von 10 nm exakt berücksichtigt, was dazu führt, dass sich bei N_{seeds} = 100 fast keine Ladungsträger in den Kristalliten befinden. Für E_{GB} = 0 eV nimmt die Besetzung in den Korngrenzen deutlich ab und es entsteht ein schuppenförmiges Bild. Dieser Verlauf, der andeutungsweise auch bei E_{GB} = -0.5 eV für N_{seeds} = 9 und d_{GB} = 1 nm zu erkennen ist, stammt daher, dass in negative x-Richtung das elektrische Feld der Source- und Drainelektrode anliegt. Dadurch entsteht ein Gradient in negativer x-Richtung und Ladungsträger sammeln sich vor Korngrenzen an. Diese Schuppenstruktur ist am deutlichsten bei E_{GB} = 0.1 eV ausgeprägt, bei der sich fast keine Ladungsträger mehr in den Korngrenzen befinden.

Somit lässt sich zunächst zusammenfassen, dass die energetischen Lage der Korngrenzen beeinflusst, ob sich die Ladungsträger entweder in den Korngrenzen, den Kristalliten oder in beiden befinden.

6.4.2. Lage des Ferminiveaus

Abhängig vom Füllgrad der Zustandsdichte ergeben sich unterschiedliche Ladungstransportarten. Deswegen analysiere ich nun die energetische Verteilung der Ladungsträger in der Zustandsdichte.

Für den Fall einer quasi-leeren Zustandsdichte, wie es beispielsweise bei einem time-of-flight-Experiment der Fall ist, bewegen sich die Ladungsträger nach einer gewissen Anfangsrelaxation auf Gitterplätzen in der Nähe der thermischen Gleichgewichtsenergie[47]

$$\varepsilon_\infty = -\frac{\sigma^2}{k_B T} \tag{6.3}$$

wobei σ die energetische Unordnung des Halbleiters ist. Die Diffusion folgt damit einer Abhängigkeit der Form $D \propto \exp(-T_0/T)^2$. Sie ist mit der Mobilität über die Einstein-Beziehung

$$\mu = \frac{eD}{k_B T} \tag{6.4}$$

verknüpft.[127] Für σ = 50 meV ergibt sich bei T = 300 K ein Wert der Gleichgewichtsenergie von ε_∞ = -100 meV. Dieses Verhalten gilt nur, solange die Ladungsträgerkonzentration hinreichend gering ist, was dem Fall $\varepsilon_\infty > \varepsilon_F$ entspricht, wobei ε_F das Ferminiveau ist.

Ist die Zustandsdichte jedoch zu einem gewissen Grad gefüllt, bildet sich ein Ferminiveau aus, und es liegt eine andere Temperaturabhängigkeit der Ladungsträgermobilität vor. Dies ist der Fall für $\varepsilon_F > \varepsilon_\infty$. Das Ferminiveau ist ein Maß für den Füllgrad der Zustandsdichte. Für die Berechnung des Ferminiveaus wird hierbei angenommen, dass die Zustandsdichte φ aus einer einfachen Gaußfunktion besteht:

$$\varphi(\varepsilon) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(\varepsilon - \varepsilon_0)^2}{2\sigma^2}\right) \tag{6.5}$$

Über die Verteilungsfunktion

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) dt = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \equiv n \tag{6.6}$$

wobei n die Ladungsträgerdichte ist, und mit Kenntnis der Fehlerfunktion $\operatorname{erf}(x)$ lässt sich numerisch, beispielsweise durch eine Nullstellensuche mit Hilfe des Newton-Verfahrens,[128] das Ferminiveau $\varepsilon_F = x\sigma + \varepsilon_0$ berechnen. In diesem Fall bewegen sich die Ladungsträger nicht von Gitterplätzen mit einer Energie in der Nähe der thermischen Gleichgewichtsenergie auf Plätze mit einer Energie des Transportniveaus, welches in der Nähe des Zentrums der Zustandsdichte liegt.[42] Stattdessen befinden sich die Ladungsträger bereits auf der Höhe des Ferminiveaus und werden auf das Transportniveau angehoben. Die Temperaturabhängigkeit der Ladungsträgermobiliät beträgt hierbei $\mu \propto \exp(-T_0/T)$. Das Ferminiveau wird dabei als konstant angenommen, wobei freilich angemerkt werden muss, dass es auch in geringem Maße von der Temperatur abhängt, vor allem jedoch die Aufweichung der Fermikante durch die Temperatur.

In der vorliegenden Untersuchung zum Ladungstransport in Korngrenzen ist die Berücksichtigung des Ferminiveaus für den Ladungstransport wichtig, da das Ferminiveau und die thermische Gleichgewichtsenergie in der gleichen Größenordnung sind. Wie bereits in Abbildung 6.7 gesehen, befinden sich die Ladungsträger für energetisch tiefe Korngrenzen in denselben. Dadurch steigt lokal die Ladungsträgerkonzentration stark an. Für eine Gatespannung von 5 V, wie sie in der Simulation verwendet wurde, ergibt sich eine

Ladungsträgerkonzentration von 1.1 % pro Gitterpunkt (nach Gleichung (3.6)), wenn sich alle Ladungsträger in der untersten Schicht befinden würden. Tatsächlich befinden sich lediglich 50 – 80 % der Ladungsträger in der untersten Schicht,[63,102] was jedoch durch den verstärkenden Effekt der Konzentration der Ladungsträger in den Korngrenzen immer noch bedeutsam ist. Für eine Ladungsträgerkonzentration von 1 % ergibt sich ein Ferminiveau von etwa -2.3σ, was in der Nähe der thermischen Gleichgewichtsenergie bei Raumtemperatur liegt (siehe oben). Durch die Ansammlung der Ladungsträger in den Korngrenzen für $E_{GB} \leq 0$ eV und der damit steigenden Ladungsträgerkonzentration liegt nun das Ferminiveau deutlich oberhalb der thermischen Gleichgewichtsenergie.

Mit Hilfe der Konzentration der Ladungsträger in den Korngrenzen und der Zustandsdichte in der Simulation kann ich nun die Lage des Ferminiveaus berechnen. Die Zustandsdichten für die verschiedenen Morphologien und Energien in den Korngrenzen sind in Abbildung 6.8 dargestellt. Dabei habe ich die Zustandsdichte jeder einzelnen Schicht im Halbleiter ohne Berücksichtigung des Potentials durch die Gatespannung aufsummiert.

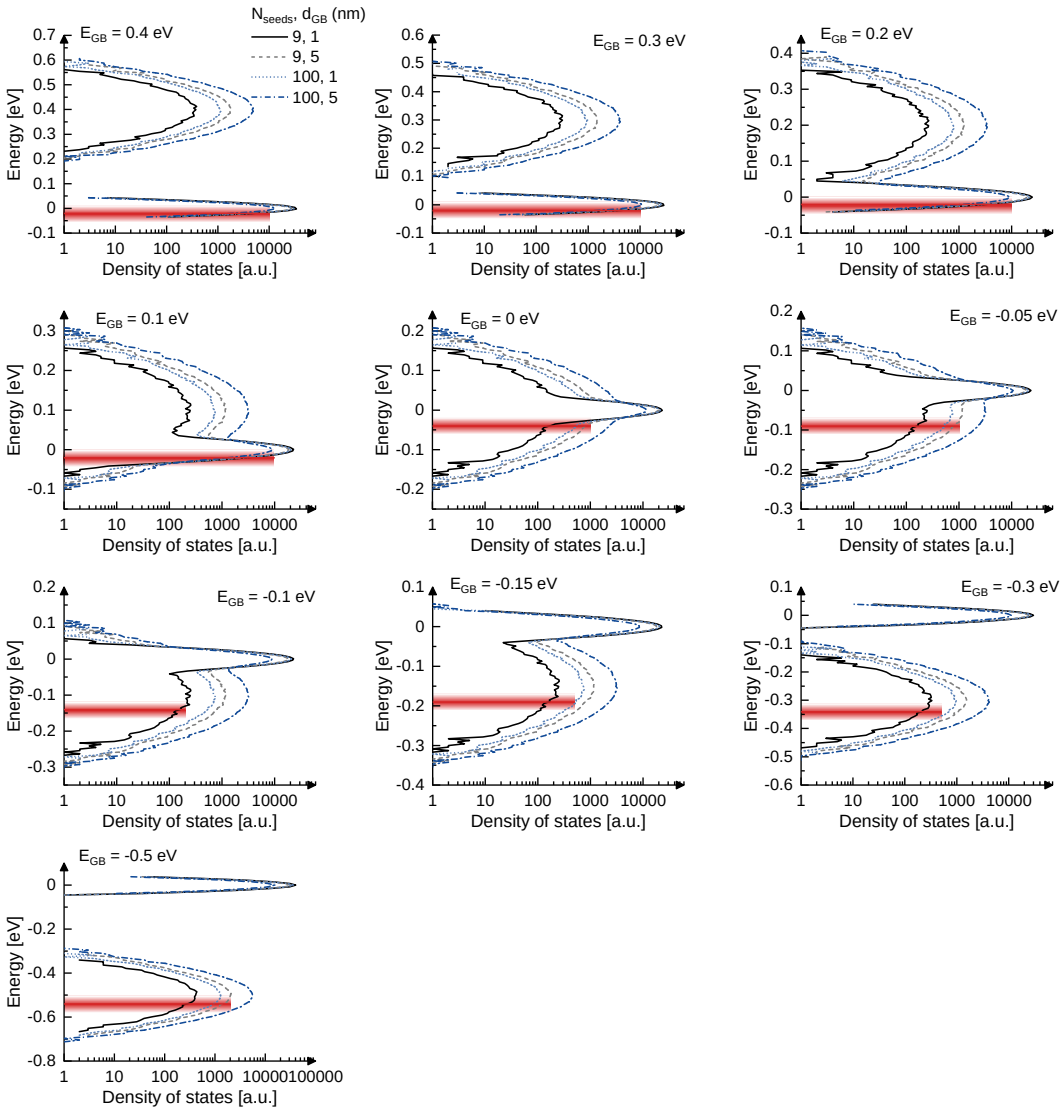

Abbildung 6.8 Simulierte Zustandsdichten der Halbleiterschicht für verschiedene Morphologien und Energien der Korngrenzen wie angegeben. Die Zustandsdichten sind dabei summiert über alle Einzellagen im Halbleiter. Die Lage des Ferminiveaus (siehe Tabelle 6.2) und dessen energetische Verbreiterung wird dabei durch den verschwommenen roten Strich dargestellt. Aus Ref.[108]

Die daraus resultierenden Ferminiveaus, sowie die Konzentration der Ladungsträger in den Korngrenzen sind in Tabelle 6.2 aufgelistet. Für die Besetzung der Korngrenzen habe ich vereinfachend angenommen, dass sich alle Ladungsträger in den Korngrenzen befinden. Mit Hilfe der Gleichung (6.6) und unter Verwendung des Newton-Verfahrens habe ich anschließend numerisch das Ferminiveau unter der Annahme einer einfachen Gaußverteilung der Zustandsdichte berechnet. Zudem sind in Tabelle 6.2 auch die Werte des Ferminiveaus für eine vollständig amorphe und eine einkristalline Halbleiterschicht eingetragen.

Tabelle 6.2 Anteil der Korngrenzen in der Morphologie und Ferminiveau unterhalb der mittleren Energie der Korngrenzen bzw. unterhalb der mittleren Energie der einkristallinen oder vollständig amorphen Phase $E_{GB} - \varepsilon_F$ für die verschiedenen Morphologien. Zum Vergleich ist die Aktivierungsenergie für E_{GB} = 0 eV aus Abbildung 6.6 in der rechten Spalte angegeben. Für die Besetzung der Korngrenzen (a) wurde angenommen, dass sich alle Ladungsträger in den Korngrenzen befinden. Nach Ref.[108]

N_{seeds}	d_{GB} (nm)	Anteil der Korngrenzen (%)	Besetzung der Korngrenzen (%)[(a)]	$E_{GB} - \varepsilon_F$ (meV)	$\frac{E_{GB} - \varepsilon_F}{\sigma}$	E_A (meV)
9	1	4.8	23.4	36.1	0.7	33
	3	14.2	7.7	71.0	1.4	-
	5	23.6	4.7	83.8	1.7	66
49	1	10.6	10.4	62.9	1.3	-
	3	31.0	3.5	90.2	1.8	-
	5	49.0	2.2	100.2	2.0	-
100	1	14.8	7.4	72.1	1.4	80
	3	42.1	2.6	97.0	1.9	-
	5	64.4	1.7	105.9	2.1	93
Einkristalline Phase, 1.1 % Besetzung				22.9	2.3	15
Vollständig amorphe Phase, 1.1 % Besetzung				114.5	2.3	108

6.4.3. Identifikation unterschiedlicher Ladungstransportbereiche

Ausgehend von der Betrachtung der Ladungsträgermobilität als Funktion morphologischer Parameter und der Temperatur, sowie in Kombination mit den Ergebnissen des Ferminiveaus, kann ich nun den Ladungstransportmechanismus untersuchen. Wie oben erläutert, ist die Temperaturabhängigkeit der Ladungsträgermobilität abhängig von der Lage des Ferminiveaus. In Abbildung 6.4 zeigen alle Kurven ein annähernd lineares Verhalten mit einer leichten Krümmung zu hohen Temperaturen hin für E_{GB} = 0 eV. Daraus kann gefolgert werden, dass sich ein Ferminiveau ausbildet, von dem aus die Ladungsträger thermisch auf eine Transportenergie angehoben werden. Die leichte Abweichung bei höheren Temperaturen kann durch die Einsteinbeziehung $\mu = eD/k_BT$ erklärt werden, wodurch die Temperaturabhängigkeit der Mobilität nicht strikt linear der der Diffusion folgt, insbesondere ergibt sich bei hohen Temperaturen eine

leichte Abweichung. Darauf gehe ich noch genauer in Kapitel 8 ein. Die ermittelten Aktivierungsenergien stimmen dabei gut mit dem Ferminiveau überein, was bedeutet, dass sich die Transportenergie nahe des Zentrums der Zustandsdichte befindet.

Gesondert betrachte ich den Fall E_{GB} = 0 eV, für den ich in Abbildung 6.6 eine detaillierte Analyse der Abhängigkeit von der Morphologie durchgeführt habe. Ausgehend von der einkristallinen Morphologie nimmt die Mobilität um bis zu einer Größenordnung allein dadurch ab, dass die Morphologie polykristallin wird, ohne dass die Korngrenzen eine räumliche Breite besitzen. Dies entspricht dem Fall, dass sich die Kristallorientierung an den Korngrenzen abrupt ändert und wurde hier in der Simulation dadurch imitiert, dass die Hüpfrate über Korngrenzen hinweg um einen Faktor 100 kleiner ist als in den Kristalliten. In einem realen System hängt der Abfall in der Mobilität mit der Änderung des Transferintegrals zusammen, welches für die Übergangswahrscheinlichkeit zwischen Molekülen verantwortlich ist, wie beispielsweise von Rivnay et al. experimentell für Perylendiimid gezeigt wurde.[34] Mit steigender Breite der Korngrenzen reduziert sich die Mobilität noch weiter, da nun der Transport durch die Korngrenzen bestimmt wird, welche eine größere energetische Unordnung besitzen.

Abhängig von der Energie der Korngrenzen lassen sich nun drei Ladungstransportbereiche unterscheiden: Energetisch tiefe Korngrenzen, energetisch flache Korngrenzen und Barrieren als Korngrenzen. Sie werden insbesondere dadurch charakterisiert, wo sich die Ladungsträger bevorzugt befinden, was Auswirkung auf die Abhängigkeit von morphologischen Parametern hat. Diese drei Bereiche sind in Abbildung 6.3 und Abbildung 6.5 mit unterschiedlichen Farben hinterlegt, mit den folgenden Energiebereichen:

(i) E_{GB} < -0.3 eV: Für energetisch tiefe Korngrenzen ist es fast unmöglich, dass die Ladungsträger thermisch aktiviert zurück auf die Kristallite gelangen, da eine thermische Energie von mehr als 10 $k_B T$ bei T = 300 K nötig wäre. Der Ladungstransport findet somit ausschließlich in den Korngrenzen statt. Dies ist nicht nur eindeutig aus Abbildung 6.7 ersichtlich, sondern zeigt sich auch dadurch, dass sich die Aktivierungsenergie in diesem Bereich einem konstanten Wert von 80 – 90 meV annähert, was in etwa dem Transport in einer Zustandsdichte mit Breite 50 meV entspricht (siehe Tabelle 6.2). Da der Transport nicht mehr über die Kristallite stattfindet, ist es auch unerheblich wie energetisch tief die Korngrenzen sind. Das Ferminiveau befindet sich etwa zwischen $1\sigma_{GB}$ und $2\sigma_{GB}$, was bedeutet, dass es hier den Ladungstransport bestimmt. In Abbildung 6.9 ist die Ladungsträgermobilität als Funktion der energetischen Unordnung der Korngrenzen für verschiedene Morphologien dargestellt. Daraus ist auch ersichtlich, dass die energetische Unordnung in den Korngrenzen den Ladungstransport bestimmt. Dabei zeigt sich,

dass die Mobilität um eine Größenordnung zunimmt, wenn die energetische Unordnung in den Korngrenzen von 50 auf 10 meV verringert wird.

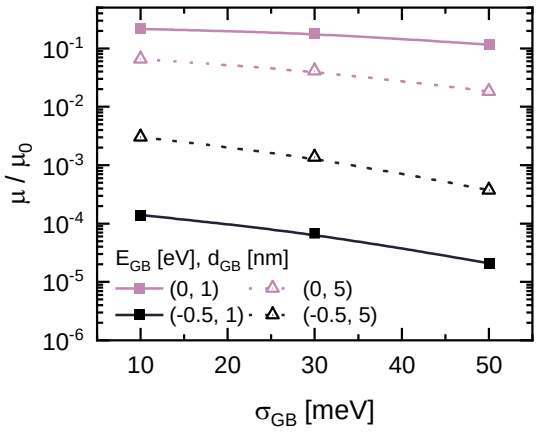

Abbildung 6.9 Ladungsträgermobilität als Funktion der energetischen Unordnung in den Korngrenzen für verschieden Morphologien wie angegeben bei Raumtemperatur. Durchgezogene Linien haben dabei eine Breite der Korngrenzen von 1 nm, gestrichelte haben 5 nm. Die Farbe gibt die Energie der Korngrenzen wider. Die Linien sind dabei nur Orientierungshilfen. Aus Ref.[108]

Ein weiteres wichtiges Merkmal, wenn der Ladungstransport ausschließlich in den Korngrenzen stattfindet, ist die Dimensionalität des Transportwegs. Durch eine Verbreiterung der Korngrenzen kann zunächst erwartet werden, dass die Ladungsträgermobilität abnimmt, da das Ferminiveau wie oben erklärt absinkt. Jedoch steigt die Mobilität etwa um einen Faktor von 20, wenn die Breite der Korngrenzen von 1 auf 3 nm erhöht wird. Dies kann dadurch erklärt werden, dass bei 1 nm breiten Korngrenzen der Transportweg in einer Halbleiterschicht eindimensional ist. Bei einer Besetzung der Korngrenzen von über 20 % ist hier eine hohe Coulombabstoßung zwischen den Ladungsträgern zu erwarten. Mit steigender Breite der Korngrenzen wird der Transportweg in der Ebene zweidimensional, wodurch einzelne feststeckende Ladungsträger in energetisch tiefen Position leichter umgangen werden können.

(ii) -0.3 eV < E_{GB} < 0 eV: Bei energetisch flachen Korngrenzen wird die höchste Ladungsträgermobilität und die niedrigste Aktivierungsenergie für den Fall von isoenergetischen Korngrenzen erreicht. Mit sinkender Energie der Korngrenzen steigt die Aktivierungsenergie im selben Maß linear an. Im Bereich -0.15 eV < E_{GB} < 0 eV stimmt die Aktivierungsenergie näherungsweise mit dem Betrag des Ferminiveaus plus dem Betrag der Energie der Korngrenzen überein, wie aus Tabelle 6.2 ersichtlich und in Abbildung 6.5 durch die rot gestrichelten Linien angedeutet

ist. Das bedeutet, dass sich die Ladungsträger in den Korngrenzen befinden, aber für den Transport auf die Kristallite thermisch aktiviert angehoben werden können und diese somit für den Transport zur Verfügung stehen. Der lineare Anstieg der Aktivierungsenergie bedeutet gleichzeitig einen exponentiellen Abfall der Mobilität wie er in Abbildung 6.3 auftritt. Die größte Aktivierungsenergie wird bei E_{GB} = -0.15 eV erreicht und nimmt für kleinere Energie der Korngrenzen wieder ab. Dies ist ein Hinweis darauf, dass bei dieser Energie ein Übergang von einem Transport auf den Kristalliten, der durch die Korngrenzen als Fallen limitierten wird, zu einem Transport fast ausschließlich in den Korngrenzen stattfindet. Im Gegensatz zum Transport, der ausschließlich in den Korngrenzen stattfindet, ist es hier hilfreich, dass die Korngrenzen möglichst dünn sind. Dies ist bereits gut dokumentiert für perylendiimid-basierte OFETs.[39]

(iii) 0 eV < E_{GB}: Für positive Werte der Energie der Korngrenzen stellen diese Barrieren für den Ladungstransport dar. Die Mobilität nimmt dann mit steigender Energie der Korngrenzen exponentiell ab, da die Aktivierungsenergie, die für die Überwindung dieser Barrieren nötig ist, zunimmt. Für die Aktivierungsenergie wird erwartet, dass sie sich aus der Summe aus dem Betrag des Ferminiveaus, etwa 20 meV (siehe Tabelle 6.2), und der Höhe der Barriere zusammensetzt. In Abbildung 6.5 ist jedoch ersichtlich, dass der Wert der Aktivierungsenergie nochmal 30 – 40 meV niedriger liegt. Dies lässt vermuten, dass die Ladungsträger die Barriere über die energetisch tiefer liegenden Zustände in den Korngrenzen überwinden.

Dünne Korngrenzen mit d_{GB} = 1 nm bilden einen Spezialfall, da hier ein Tunneln durch die Barriere möglich ist. Dies zeigt sich daran, dass die Mobilität für E_{GB} > 0.15 eV einen konstanten Wert annimmt, was sich aus dem hier verwendeten Modell für die Übergangsraten (Gleichung (6.1)) ergibt. Prinzipiell ist auch bei breiteren Korngrenzen ein Tunneln durch die Barrieren zu erwarten, jedoch ist die Wahrscheinlichkeit dafür vernachlässigbar gering, weswegen in der Simulation Sprünge mit Distanzen größer gleich 3 Gitterplätzen nicht berücksichtigt wurden. Ebenfalls nicht berücksichtigt in der Simulation ist die Tatsache, dass die Tunnelwahrscheinlichkeit von der Höhe der Barriere abhängt, wie es rein rechnerisch zu erwarten wäre.[129]

6.5. Zusammenfassung

In diesem Abschnitt meiner Arbeit habe ich eine Reihe von Monte-Carlo-Simulationen von polykristallinen OFETs durchgeführt, um den Einfluss der Korngrenzen auf den Ladungstransport zu untersuchen. Ich habe drei verschiedene Transportbereiche identifiziert, die abhängig von

der Breite und der Energie der Korngrenzen sind. Für energetisch tiefe Korngrenzen zeigt sich, dass der Ladungstransport fadenförmig fast ausschließlich in den Korngrenzen stattfindet. Dabei wird der Transport durch ein dichtes Netzwerk an breiten Korngrenzen mit geringer energetischer Unordnung begünstigt. Mit zunehmender Energie der Korngrenzen findet der Ladungstransport vor allem innerhalb der Kristallite statt, wird jedoch durch das Ferminiveau in den Korngrenzen begrenzt, welche als Fallenzustände wirken und aus denen die Ladungsträger thermisch aktiviert entkommen müssen. Dieses Entkommen aus den Fallen wird durch schmale Korngrenzen begünstigt, da dort das Ferminiveau höher ist. Handelt es sich bei den Korngrenzen um energetische Barrieren für die Ladungsträger, müssen diese durch Tunneln im Fall von hinreichend dünnen Korngrenzen oder durch thermische Aktivierung bei breiten Korngrenzen überwunden werden, was beide Male die Ladungsträgermobilität drastisch reduziert.

Durch diese Ergebnisse zeigt sich, dass das Ferminiveau eine wichtige Rolle spielt für die Ladungsträgermobilität in dem am wahrscheinlich häufigsten Fall von flachen Fallen in den Korngrenzen von polykristallinen OFETs. Liegen schmale, energetisch flache Korngrenzen mit geringer energetischer Unordnung vor, so ist die Abnahme der Mobilität im Vergleich zu der im Fall eines einkristallinen Halbleiters am geringsten. Häufig wird jedoch in Simulation die Lage des Ferminiveaus und die Besetzung der Gitterplätze vernachlässigt, obwohl dies einen Einfluss auf die Temperaturabhängigkeit und den Wert der Mobilität selbst hat. Meine Untersuchung bietet somit eine Grundlage dafür, warum in Diskussionen zur Ladungsträgermobilität in OFETs das Ferminiveau mit berücksichtigt werden sollte.

7. Der Einfluss der Morphologie von Oligomeren auf den Ladungstransport in organischen Feldeffekttransistoren

7.1. Einleitung

In kristallinen molekularen Halbleitern spielt neben der molekularen Struktur vor allem die Anordnung der Moleküle zueinander eine wichtige Rolle für den Überlapp der Orbitale und damit den Ladungstransport.[5,40] Typische Anordnungen von Molekülen zueinander sind schematisch in Abbildung 7.1 dargestellt.

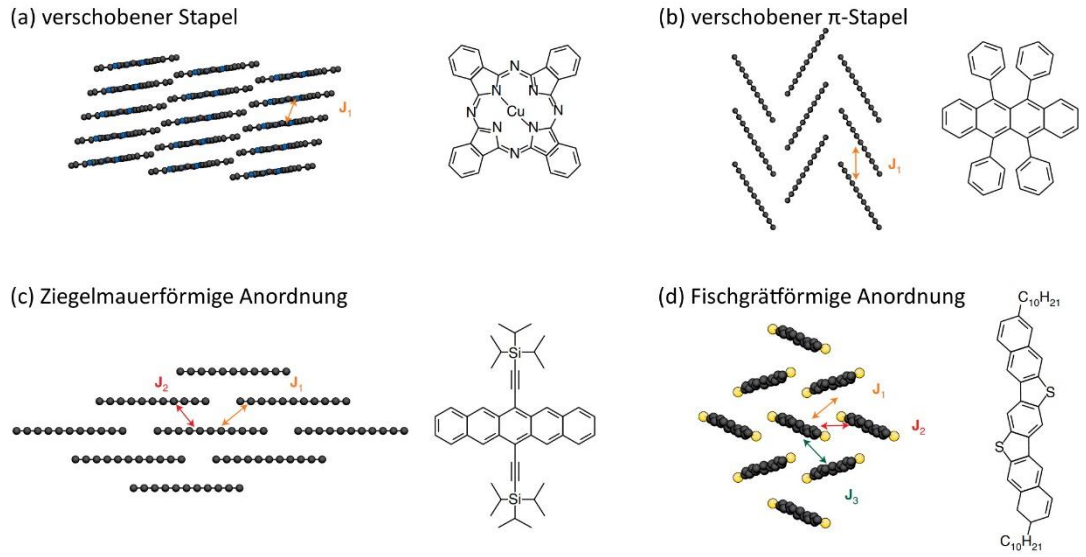

Abbildung 7.1 Typische Anordnungen der Moleküle zueinander in kristallinen organischen Halbleitern. Die chemischen Strukturen stellen typische Vertreter der jeweiligen Anordnung dar. (a) α-Phase von Kupferphthalocyanin, (b) Rubren, (c) TIPS-Pentacen, (d) C_{10}-DNBDT-NW. Die farbigen Pfeile stellen die Haupttransferintegrale J zwischen den Molekülen dar. Angepasst nach Ref.[5]

In konjugierten semikristallinen Polymeren kann durch einen hohen Ausrichtungsgrad der Moleküle, der als nematisch flüssigkristalline Anordnung aufgefasst werden kann,[84] im Allgemeinen

eine höhere Ladungsträgermobilität erzielt werden.[25,57,81,82,130] Mittels einer smektisch flüssigkristallinen Anordnung, bei der die Moleküle zusätzlich zur nematisch flüssigkristallinen Anordnung in Schichten angeordnet sind, können ebenfalls hohe Ladungsträgermobilitäten von über 20 $cm^2 V^{-1} s^{-1}$ erreicht.[131] Die ziegelmauerförmige Anordnung von Molekülen (Abbildung 7.1c) ist ebenfalls ein Spezialfall einer nematisch flüssigkristallinen Anordnung.

Neben der Anordnung der Moleküle zueinander, kann auch die Konjugationslänge von konjugierten organischen Halbleitern einen Einfluss auf die Ladungsträgermobilität haben. Dieser Einfluss zeigt sich am Beispiel von linearen Acenen, die aus leiterförmig angeordneten verbundenen Benzolringen bestehen.[132] Dabei hat sich experimentell als auch durch Dichtefunktionaltheorie gezeigt, dass mit steigender Anzahl an Benzolringen die Ladungsträgermobilität zunimmt.[27,132] Bei bestimmten Oligomeren gibt es jedoch auch Abweichungen von der direkten Korrelation zwischen Konjugationslänge und Ladungsträgermobilität. So weist beispielsweise das Oligomer DPP-6T eine geringere Mobilität auf als das kürzere Pendant DPP-4T, was jedoch in diesem Fall durch eine höhere Anzahl an Konfigurationsfreiheitsgraden von DPP-6T erklärt werden kann, was wiederum zu einem häufigeren Auftreten von Defekten in der Kristallstruktur beim Kristallwachstum führt und damit die Mobilität verringert.[133] Für bestimmte Polymere gilt außerdem, nachdem sie durch die Prozessierung ausgerichtet worden sind, dass die Ladungsträgermobilität nicht mehr durch die Länge der Polymere beeinflusst wird.[57]

Diese experimentellen Beobachtungen zeigen, dass die Anordnung der Moleküle zueinander und die Konjugationslänge von konjugierten semikristallinen Systemen einen wesentlichen Einfluss auf die Ladungstransporteigenschaften haben. Um Vorhersagen über die notwendige chemische Struktur von hocheffizienten organischen Halbleitermaterialien treffen zu können, ist es hilfreich den Einfluss der Anordnung und der Konjugationslänge der Moleküle auf den Ladungstransport zu verstehen.

Im Folgenden untersuche ich deswegen die Ladungstransporteigenschaften verschiedener flüssigkristalliner Anordnungen von Oligomeren als Funktion des Versatzes benachbarter Oligomere und der Länge der Oligomere. Dabei verwende ich folgende Nomenklatur: Kettenlänge bezeichnet die Länge eines Oligomers und ein Oligomerstrang ist eine Sequenz von Oligomeren, die die konjugierten Bereichen in einem Polymer darstellen. Ich beschränke mich hier auf den einfachen Fall von Oligomeren, da bei Polymeren zusätzlich Faltungseffekte der Polymere auftreten können, die die Ladungsträgermobilität beeinflussen.[134] Außerdem treten bei Polymeren

aus Entropiegründen immer auch amorphe Bereiche auf. Neben dem reinen Einfluss der Anordnung von isoenergetischen Oligomeren untersuche ich zudem den Fall, dass die Energien der Oligomere diskret gaußförmig verteilt sind.

Nach meinem besten Wissen existiert noch keine systematische Untersuchung des Zusammenhangs zwischen der Anordnung der Moleküle und den Ladungstransporteigenschaften von konjugierten semikristallinen Molekülen.

7.2. Verwendete Simulationsparameter

Die Simulation von Ladungstransport in polymeren organischen Halbleitern habe ich wie im vorangegangenen Kapitel mit der in Kapitel 5.3 beschriebenen Software Bumblebee durchgeführt. Ich habe einen Feldeffekttransistor wie in Abbildung 7.2 dargestellt modelliert.

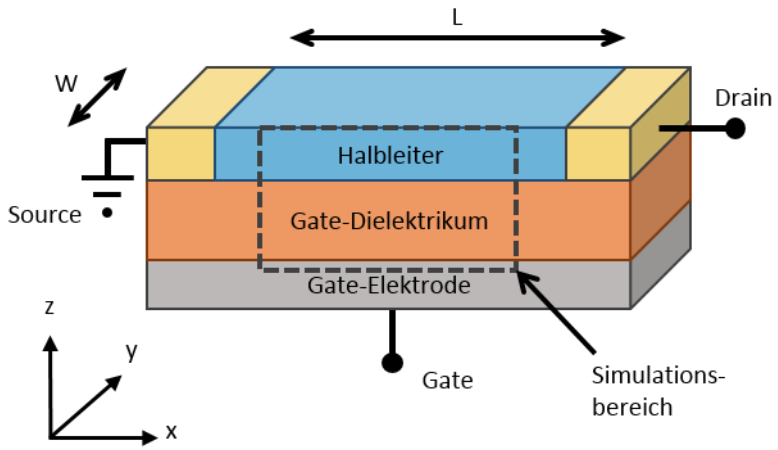

Abbildung 7.2 Schematische Darstellung eines Feldeffekttransistors.

Die Größe der Simulationsbox des organischen Halbleiters, soweit nicht anders angegeben, entspricht 180 x 180 x 10 Gitterpunkten mit einem Gitterabstand von 1 nm in allen Raumrichtungen. Für die Source-Drain-Spannung U_{SD} habe ich, soweit nicht anders angegeben, 0.72 V verwendet. Dadurch ergibt sich eine elektrische Feldstärke von $6 \cdot 10^4$ V cm^{-1} in Source-Drain-Richtung. Die Dicke des Gatedielektrikums beträgt 100 nm. Sprünge der Ladungsträger werden bis zum übernächsten Nachbar in jeder Richtung berücksichtigt. Als Wert für die Dielektrizitätskonstante im Halbleiter und im Gatedielektrikum wurde 4 verwendet. Ladungsträgerinjektion und -extraktion werden nicht simuliert. Dabei gelten für die Wahl der Simulationsparameter die gleichen Überlegungen wie in Kapitel 6.2 in der vorangegangenen Untersuchung.

Die Hüpfrate k_{ij} zwischen zwei Gitterplätzen i und j nach Miller-Abrahams beträgt (siehe Kapitel 5.3):[99]

$$k_{ij} = \nu_0 \exp(-2\gamma|r_{ij}|) \begin{cases} \exp\left(-\frac{\Delta E_{ij}}{kT}\right) & \Delta E_{ij} > 0 \\ 1 & \Delta E_{ij} \leq 0 \end{cases} \tag{7.1}$$

Dabei beträgt die attempt-to-hop Frequenz ν_0 = 10^{13} s^{-1} für Sprünge zwischen Oligomeren und ν_0 = 10^{15} s^{-1} für Sprünge innerhalb der Oligomere bzw. des Kristallits. Die inverse Lokalisierungslänge beträgt γ = 5 nm^{-1} und als Temperatur wurde durchgehend T = 300 K verwendet.

Für die Erstellung von oligomeren Strukturen im Halbleiter habe ich den in Kapitel 5.4.2 beschriebenen Algorithmus verwendet. Die daraus resultierenden Morphologien sind in Abbildung 7.3 dargestellt.

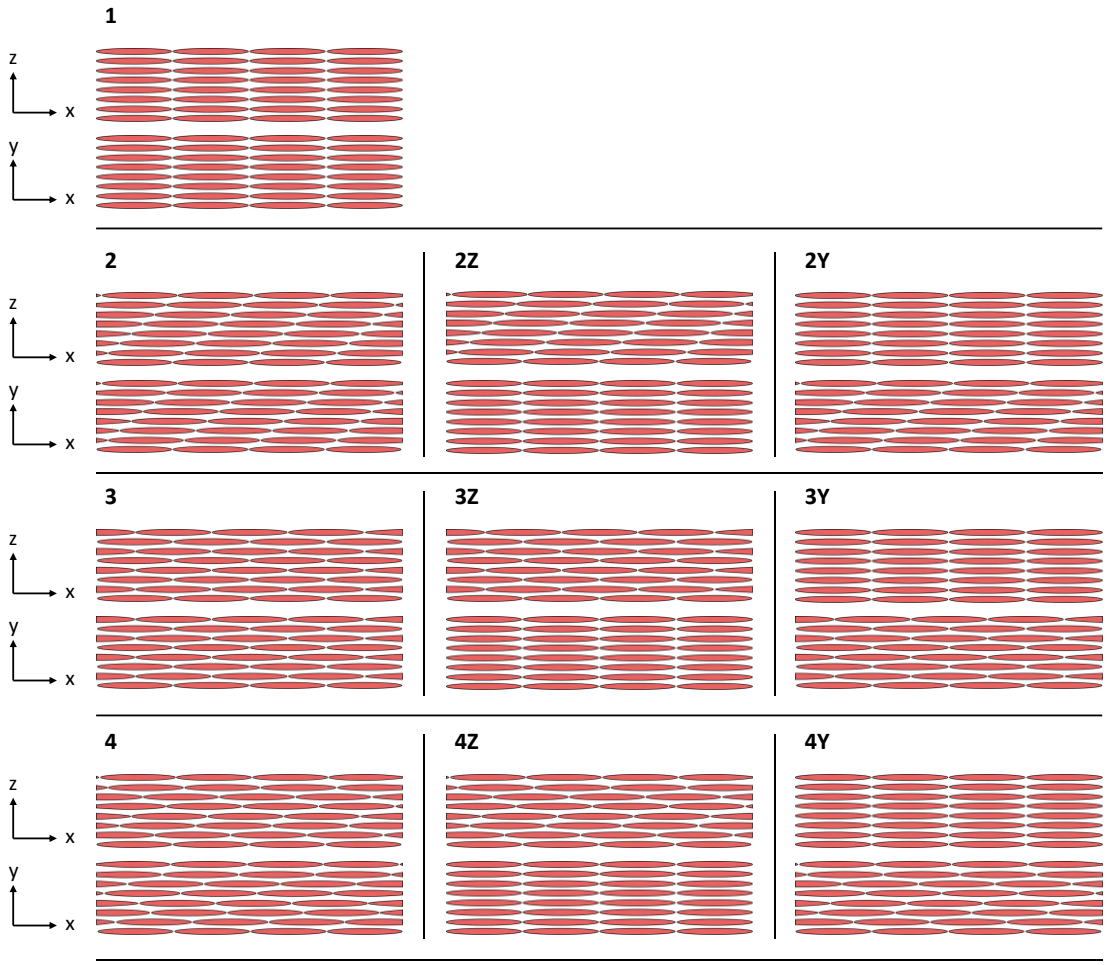

Abbildung 7.3 Verwendete Morphologien der Oligomere in der Halbleiterschicht des Feldeffekttransistors. Für jede Morphologie sind zwei Querschnitte gezeigt, einmal in der x-z-Ebene, d.h. als Querschnitt

durch den Film, (jeweils oben) und einmal in der x-y-Ebene, d.h. als Draufsicht auf den Film (jeweils unten), wie durch das Koordinatensystem auf der linken Seite dargestellt ist. Die roten Ellipsen repräsentieren dabei die Positionen der Oligomere und ihren räumlichen Versatz zueinander. Für eine leichtere Zuordnung werden die verschiedenen Morphologien von oben nach unten durchnummeriert. Der Zusatz „Z“ bzw. „Y“ bei jeder Nummer bedeutet, dass bei dieser Morphologie die Oligomere nur in der x-z- bzw. x-y-Ebene einen Versatz zueinander aufweisen.

Die Oligomere sind bei jeder Morphologie in x-Richtung, der Source-Drain-Richtung, ausgerichtet. Ich habe vier verschiedene Typen von Morphologien untersucht, welche für eine einfachere Zuordnung durchnummeriert werden. Nummer 1: Die Oligomere sind parallel in y- und z-Richtung angeordnet. Nummer 2: Die Oligomere weisen in y- oder z-Richtung oder in beiden einen festen Versatz zwischen benachbarten Oligomeren auf. Nummer 3: Die Oligomere weisen in y- oder z-Richtung oder in beiden einen Versatz benachbarter Oligomere auf, der genau der Hälfte der Kettenlänge entspricht und somit abhängig von der Kettenlänge ist. Nummer 4: Die Oligomere weisen in y- oder z-Richtung oder in beiden einen Versatz benachbarter Oligomere auf, der jedoch vollkommen zufällig gewählt wird. Diese Morphologien habe ich gewählt, um den Einfluss des Versatzes benachbarter Oligomere auf den Ladungstransport untersuchen zu können. Die verschiedenen Anordnungen entsprechen dabei unterschiedlichen flüssigkristallinen Phasen. Zudem unterscheide ich zwischen einem Versatz benachbarter Oligomere nur in y- oder z-Richtung bzw. in beiden Richtungen, da in z-Richtung aufgrund der anliegenden Gatespannung zusätzlich ein Feldgradient vorliegt und somit ein anderes Verhalten bei einem Versatz benachbarter Oligomere lediglich in y-Richtung als bei einem Versatz benachbarter Oligomere in z-Richtung erwartet werden kann.

Neben dem Fall, dass die Oligomere alle die gleiche Energie besitzen, betrachte ich auch den Fall einer annähernd gaußförmigen Verteilung von Energien der Oliogmere. Dafür verwende ich das in Kapitel 5.4.2 beschriebene Verfahren einer diskreten Gaußverteilung der Energien. Dazu unterteile ich eine Gaußverteilung mit einer Standardabweichung von 50 bzw. 100 meV in neun äquidistante Intervalle, die so gewählt sind, dass 99 % der Energiewerte innerhalb dieser Intervalle liegen. Diese Aufteilung ist in Abbildung 7.4 dargestellt.

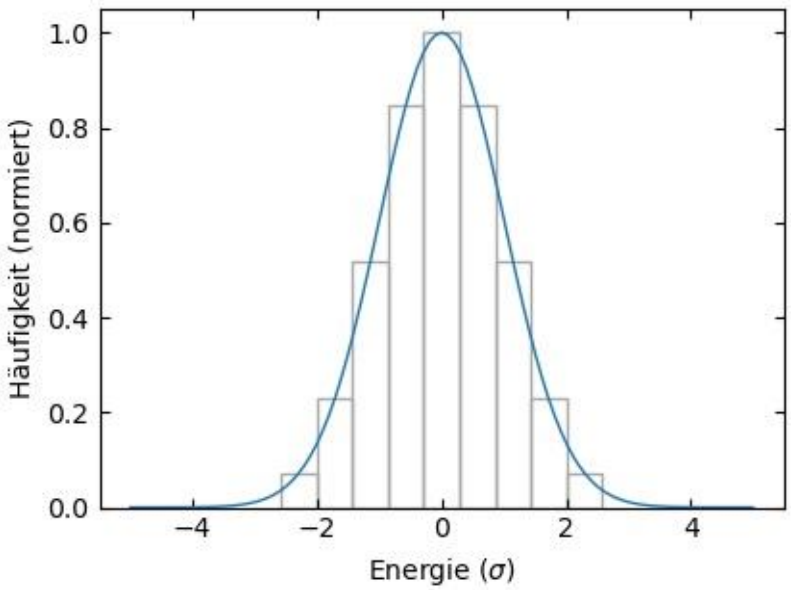

Abbildung 7.4 Aufteilung einer Gaußverteilung in äquidistante Intervalle, deren Bereich 99 % aller Energiewerte abdeckt. Die Energieachse ist in Einheiten der Standardabweichung der Gaußverteilung angegeben.

Oligomere können nur Energien aus dieser diskreten Verteilung annehmen mit den entsprechenden Wahrscheinlichkeiten der Gaußverteilung.

7.3. Ladungstransportsimulation für isoenergetische Oligomere

7.3.1. Ergebnisse der Stromdichte

Ich untersuche zunächst den Fall von isoenergetischen Oligomeren, also bei dem jedes Oligomer die gleiche Energie besitzt. Dadurch kann der Ladungstransport nur unter dem Einfluss der Morphologie betrachtet werden. Abbildung 7.5 zeigt die Stromdichte als Funktion der Kettenlänge der Oligomere für drei der in Abbildung 7.3 gezeigten Morphologien, sowie einer weiteren (3*), bei der der Versatz benachbarter Oligomere genau einem Drittel der Kettenlänge entspricht. Letztere habe ich gewählt, um Morphologie 3 mit einer weiteren Morphologie vergleichen zu können, bei der der Versatz benachbarter Oligomere nicht genau der Hälfte entspricht und somit eine zusätzliche Asymmetrie vorliegt.

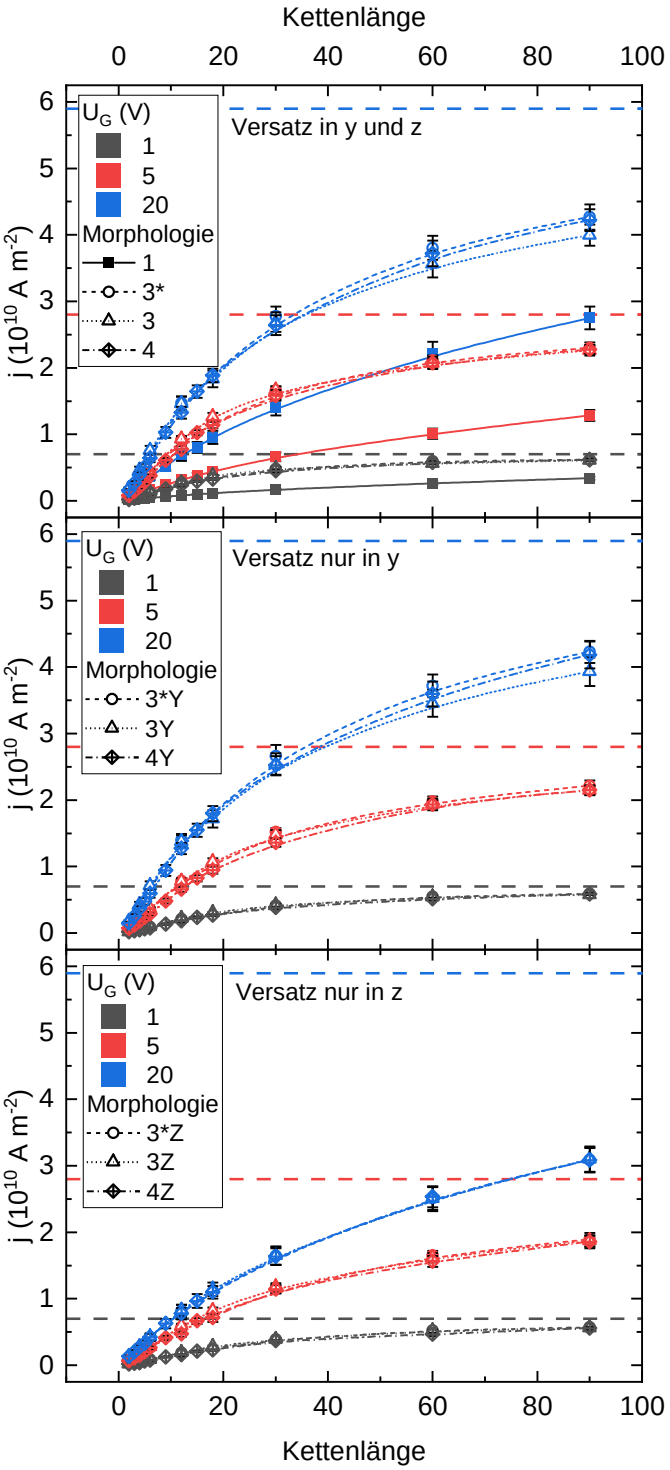

Abbildung 7.5 Stromdichte als Funktion der Kettenlänge der Oligomere für verschiedene Morphologien und Gatespannungen wie angegeben. Morphologie 3* bzw. 3*Y und 3*Z bezeichnet hier eine Morphologie, bei der benachbarte Oligomere genau um ein Drittel ihrer Kettenlänge zueinander versetzt sind. Die horizontalen gestrichelten Linien in jedem Diagramm zeigen zu jeder Gatespannung die Stromdichte, die sich bei unendlich langen Oligomeren ergibt, wobei die zugehörigen Werte in Tabelle 7.1 aufgeführt sind. Die eingezeichneten Linien dienen lediglich der Veranschaulichung.

Zunächst fällt auf, dass die Stromdichte mit steigender Kettenlänge zunimmt. Dies ist der Fall für alle Morphologien und tritt bei jeder Gatespannung auf. Der Zuwachs in der Stromdichte ist dabei bei kleinerer Kettenlänge stärker als bei größerer, das heißt die Kurven sind rechtsgekrümmt. Die Reihenfolge der Werte der Stromdichte folgt zudem der Gatespannung, das heißt mit zunehmender Gatespannung steigt die Stromdichte bei konstanter Kettenlänge. Werden die einzelnen Morphologien untereinander verglichen, so zeigt sich, dass die Morphologie 1 stets die kleinsten Werte für die Stromdichte aufweist. Die Werte der Stromdichte der Morphologie 3*, 3 und 4 hingegen sind in etwa gleich, mit Unterschieden, die innerhalb der Fehlertoleranzen liegen. Auf diese Unterschiede wird später noch genauer eingegangen.

Zur besseren Einordnung der Werte der Stromdichten habe ich eine weitere Simulation durchgeführt, bei der die Kettenlänge der Oligomere der Größe der Simulationsbox in x-Richtung entspricht. Durch die periodische Randbedingung in x-Richtung entstehen unendlich lange Oligomere. Der Wert, der sich bei dieser Simulation ergibt, kann zur Bestimmung des Maximalwertes der Stromdichte dienen, der mit Oligomeren bei den simulierten Parametern möglich ist. Die Ergebnisse der unendlich langen Oligomere sind in Tabelle 7.1 dargestellt und in der Abbildung 7.5 als gestrichelte horizontale Linien farblich entsprechend der Gatespannung eingezeichnet. In Tabelle 7.1 sind zudem auch die Werte für die Ladungsträgermobilität des unendlich langen Oligomers angegeben, die ich wie weiter unten beschrieben berechnet habe.

Tabelle 7.1 Stromdichte j_∞ und Ladungsträgermobilität μ_∞ eines unendlich langen Oligomers bei verschiedenen Gatespannungen U_G.

U_G (V)	j_∞ (A m^{-2})	μ_∞ (cm^2 V^{-1} s^{-1})
1	$(7.06 \pm 0.16) \cdot 10^9$	331 ± 7
5	$(2.80 \pm 0.08) \cdot 10^{10}$	263 ± 6
20	$(5.90 \pm 0.17) \cdot 10^{10}$	139 ± 4

Es zeigt sich, dass der Wert für die Stromdichte bei unendlich langen Oligomeren stets größer ist als bei kurzer Kettenlänge. Die Kurven mit endlicher Kettenlänge nähern sich den horizontalen Linien für den Wert bei unendlich langer Kettenlänge augenscheinlich asymptotisch an.

Beim Vergleich der Morphologien mit einem Versatz benachbarter Oligomere nur in einer Dimension zeigt sich, dass ein Versatz benachbarter Oligomere nur in der y-Richtung, d.h. innerhalb einer Lage, eine größere Stromdichte bewirkt als ein Versatz benachbarter Oligomere nur

in der z-Richtung, d.h. zwischen den Lagen. Ein Versatz benachbarter Oligomere in beiden Richtungen ergibt nochmal einen größeren Wert für die Stromdichte im Vergleich zum Versatz benachbarter Oligomere nur in der y-Richtung, jedoch ist der Unterschied nicht so groß wie beim Versatz benachbarter Oligomere nur in y- oder z-Richtung.

Die Werte der Stromdichte unterscheiden sich bei den Morphologien 3*, 3 und 4 nur geringfügig voneinander. Um den Einfluss des Versatzes benachbarter Oligomere noch genauer zu untersuchen, habe ich deswegen eine Simulation bei fester Kettenlänge mit sich änderndem Versatz benachbarter Oligomere durchgeführt. Abbildung 7.6 zeigt den Verlauf der Stromdichte als Funktion des Versatzes benachbarter Oligomere bei einer konstanten Kettenlänge von 12 für die Morphologien 2, 2Y und 2Z.

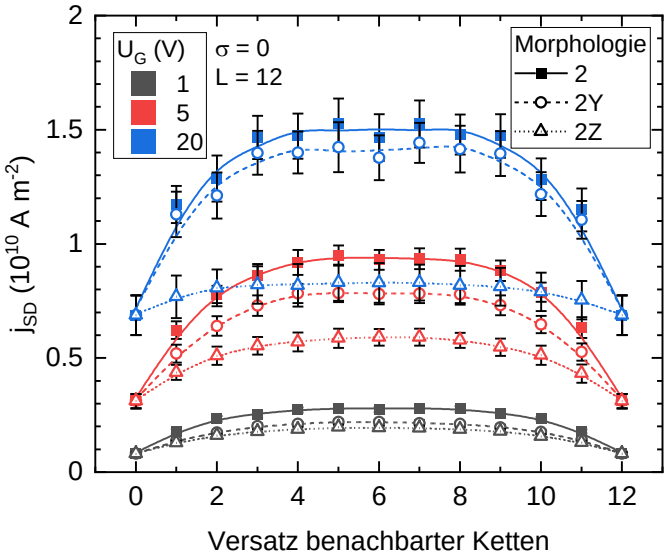

Abbildung 7.6 Stromdichte als Funktion des Versatzes benachbarter Oligomere für verschiedene Gatespannungen und unterschiedliche Morphologien wie angegeben. Die Kettenlänge der Oligomere beträgt stets 12.

Es zeigt sich, dass die Stromdichte einen Maximalwert bei einem Versatz benachbarter Oligomere von ungefähr der Hälfte der Kettenlänge annimmt. Der Verlauf der Stromdichte als Funktion des Versatzes benachbarter Oligomere ist bezüglich der Mitte bei allen Morphologien und Gatespannungen in etwa symmetrisch.

Hier zeigt sich nochmal deutlicher, dass ein Versatz benachbarter Oligomere nur in y-Richtung unabhängig vom gewählten Versatz höhere Werte für die Stromdichte liefert als ein Versatz

benachbarter Oligomere nur in z-Richtung, jedoch niedriger, wenn der Versatz benachbarter Oligomere in beiden Dimensionen vorliegt.

7.3.2. Ergebnisse der Ladungsträgermobilität

Um das Verständnis für den Ladungstransportmechanismus weiter zu vertiefen, habe ich aus den Werten für die Stromdichte die Ladungsträgermobilität berechnet. Dafür habe ich die Anzahl der Ladungsträger während einer Simulation genommen und durch das gesamte Halbleitervolumen geteilt, um so die mittlere Ladungsträgerdichte zu bestimmen. Die Mobilität berechnet sich dann wie in Kapitel 5.3.3 beschrieben aus der Stromdichte gemäß

$$\mu = \frac{j}{\rho F} \tag{7.2}$$

Die Werte für die Ladungsträgerdichte, die ich bei dieser Umrechnung verwendet habe, sind in Tabelle 7.2 aufgelistet. In Tabelle 7.1 habe ich zudem die Stromdichte für unendlich lange Oligomere in eine Ladungsträgerdichte umgerechnet.

Tabelle 7.2 Mittlere Ladungsträgerdichten ρ, die sich bei den verwendeten Gatespannungen U_G aus Gleichung (3.6) ergeben. Die Einheit nm^{-3} entspricht dabei der Einheit pro Gitterpunkt, da jeder Gitterpunkt ein Volumen von 1 nm^3 einnimmt.

U_G (V)	ρ (10^{-3} nm^{-3})
1	0.222
5	1.10
20	4.42

Abbildung 7.7 zeigt die sich daraus ergebende Ladungsträgermobilität als Funktion der Kettenlänge für die in Abbildung 7.5 durchgeführten Simulationen.

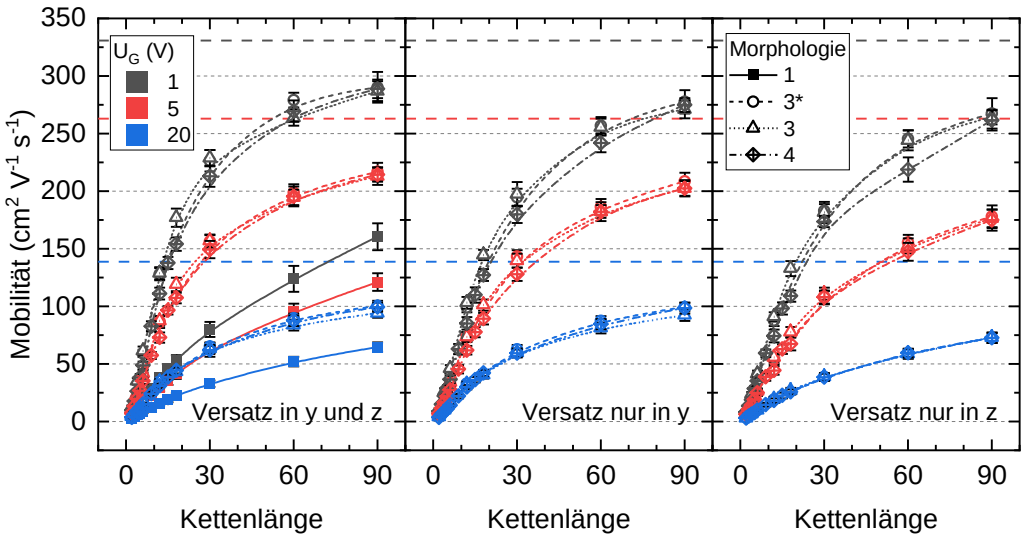

Abbildung 7.7 Ladungsträgermobilität als Funktion der Kettenlänge der Oligomere für verschiedene Morphologien und Gatespannungen wie angegeben. Die Morphologie 3* bezeichnet hier eine Morphologie, bei der benachbarte Oligomere genau um ein Drittel ihrer Kettenlänge zueinander versetzt sind. Die horizontalen gestrichelten Linien zeigen zu jeder Gatespannung die Ladungsträgermobilität, die sich bei unendlich langen Oligomeren ergibt. Die übrigen Linien dienen lediglich der Veranschaulichung.

Der Verlauf der Mobilität als Funktion der Kettenlänge ist wie bei der Stromdichte streng monoton steigend. Im Gegensatz zur Stromdichte nimmt die Mobilität mit steigender Gatespannung ab, das heißt bei kleinerer Gatespannung werden höhere Mobilitätswerte erzielt. Das bedeutet, dass die Mobilitäten bei kleinerer Ladungsträgerdichte größer sind. In Übereinstimmung mit der Stromdichte nähert sich auch die Mobilität mit steigender Kettenlänge augenscheinlich asymptotisch dem Wert für unendlich lange Oligomere an.

Analog zur Untersuchung der Stromdichte habe ich auch die Mobilität als Funktion des Versatzes benachbarter Oligomere bei einer Oligomerlänge von 12 berechnet. In Abbildung 7.8 ist die Mobilität als Funktion des Versatzes benachbarter Oligomere für verschiedene Morphologien gezeigt.

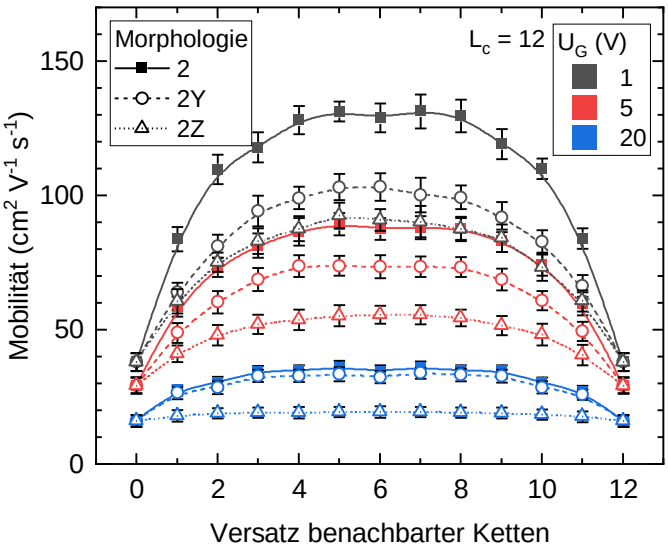

Abbildung 7.8 Mobilität als Funktion des Versatzes benachbarter Oligomere für verschiedene Gatespannungen und unterschiedliche Morphologien wie angegeben für isoenergetische Oligomere. Die Kettenlänge der Oligomere L_c beträgt stets 12. Die Linien dienen lediglich der Veranschaulichung.

Die Mobilität weist das gleiche symmetrische Verhalten bezüglich dem Versatz benachbarter Oligomere wie die Stromdichte auf. Außerdem zeigt sich hier, dass die Mobilität für einen Versatz benachbarter Oligomere lediglich in z-Richtung durchgängig kleinere Werte liefert als bei einem Versatz benachbarter Oligomere lediglich in y-Richtung. Dies ist am deutlichsten bei einer Gatespannung von 20 V ausgeprägt.

7.3.3. Analyse der Stromdichte und Mobilität

Im Folgenden möchte ich auf die Beobachtungen der Simulation der Stromdichte und Mobilität im OFET für verschiedene morphologische Parameter und Gatespannungen eingehen. Dabei analysiere ich zunächst den funktionalen Zusammenhang zwischen Stromdichte bzw. Mobilität und der Kettenlänge, gehe anschließend auf den Einfluss des Versatzes benachbarter Oligomere und schließlich auf die Abhängigkeit der Stromdichte und Mobilität von der Gatespannung ein.

In Abbildung 7.5 war zu erkennen, dass die Stromdichte mit der Länge der Oligomere unabhängig von der Morphologie und der Gatespannung ansteigt. Dies ist leicht nachvollziehbar, wenn die Kopplungsparameter innerhalb und zwischen den Oligomeren betrachtet werden. Sprünge der Ladungsträger innerhalb der Oligomere finden mit einer um einen Faktor 100 höheren Rate

als Sprünge zwischen Oligomeren statt. Mit steigender Kettenlänge sinkt somit der Anteil an langsamen Sprüngen zwischen Oligomeren. Dadurch ist mit steigender Kettenlänge ein Anwachsen der Stromdichte zu erwarten, die sich immer mehr dem Wert der Stromdichte für unendlich lange Oligomere annähern muss. Um einen genaueren funktionalen Zusammenhang zwischen der Kettenlänge und der Stromdichte zu erhalten, betrachte ich den Fall eines eindimensionalen Oligomerstranges. Für diesen Fall kann mittels einer einfacher Abschätzung ein Zusammenhang zwischen Kettenlänge und Stromdichte bzw. Mobilität hergestellt werden. Für die Herleitung dieses Zusammenhangs wird ein Oligomerstrang wie in Abbildung 7.9 dargestellt angenommen, wobei n_m die Anzahl an Wiederholeinheiten in einem Oligomer und n_o die Anzahl an Oligomeren innerhalb des gesamten Oligomerstranges bezeichnet.

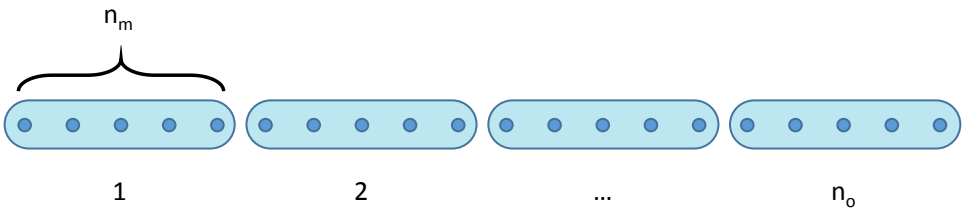

Abbildung 7.9 Schematische Darstellung eines eindimensionalen Oligomerstranges. n_m bezeichnet die Anzahl an Monomereinheiten innerhalb eines Oligomers, n_o die Anzahl an Oligomeren im gesamten Strang.

Die Anzahl aller Wiederholeinheiten in einem Strang ergibt sich zu:

$$N \equiv n_m n_o \tag{7.3}$$

In Analogie zu einem einfachen Stromkreis aus Widerständen kann der Widerstand, den ein Ladungsträger beim Durchlaufen eines Stranges von Oligomeren erfährt, als eine Serienschaltung von Widerständen betrachtet werden. Widerstände verursachen dabei die Sprünge innerhalb und zwischen den Oligomeren. Dies ist in Abbildung 7.10 dargestellt.

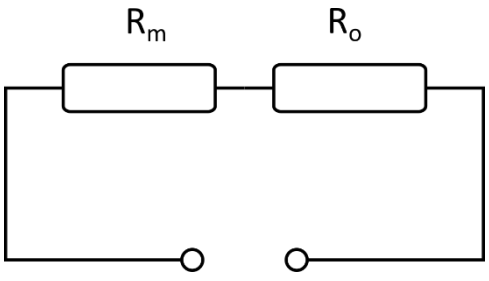

Abbildung 7.10 Ersatzschaltbild für die elektrische Analogie zu den Sprüngen in einem Oligomerstrang.

R_m bezeichnet dabei den Widerstand, den ein Ladungsträger aufgrund der Sprünge innerhalb von Oligomeren und R_o den Widerstand, den der Ladungsträger aufgrund von Sprüngen zwischen Oligomeren erfährt. Der Widerstand des gesamten Stranges von Oligomeren beträgt dann:

$$R = R_m + R_o \tag{7.4}$$

Der Widerstand eines elektrischen Leiters ist proportional zur Länge des Leiters und umgekehrt proportional zur Beweglichkeit der Ladungsträger. Die Beweglichkeit der Ladungsträger wiederrum ist proportional zur Übergangsrate zwischen den beteiligten Gitterpunkten. Übertragen auf die Sprünge in einem Oligomerstrang bedeutet dies:

$$R \propto \frac{n}{k} \tag{7.5}$$

wobei n der Länge des Leiters entspricht und k die zugehörige Hüpfrate ist. Für den Gesamtwiderstand gilt dann:

$$R \propto n_o \frac{n_m - 1}{k_m} + \frac{n_o}{k_o} \tag{7.6}$$

wobei k_m die Hüpfrate innerhalb eines Oligomers und k_o die Hüpfrate zwischen den Oligomeren ist. Die Reduktion um 1 bei der Anzahl der Wiederholeinheiten im Oligomer n_m habe ich deswegen gemacht, da ein Oligomer mit Kettenlänge 1 kein Oligomer, sondern nur ein einzelner Gitterpunkt ist und somit dann keine Sprünge mehr innerhalb eines Oligomers stattfinden können. Durch Umformung ergibt sich:

$$R \propto \frac{N}{k_o}\left(\frac{k_o}{k_m}\frac{n_m - 1}{n_m} + \frac{1}{n_m}\right) \tag{7.7}$$

λ wird als das Verhältnis zwischen den Hüpfraten innerhalb und zwischen den Oligomeren definiert:

$$\lambda \equiv \frac{k_m}{k_o} \tag{7.8}$$

Da die Beweglichkeit μ umgekehrt proportional zum Widerstand R ist, kann für die Beweglichkeit geschrieben werden:

$$\mu \propto \frac{1}{R} \propto \frac{k_o}{N}\left(\frac{1}{\lambda}\frac{n_m - 1}{n_m} + \frac{1}{n_m}\right)^{-1} = \frac{k_o}{N}\frac{\lambda n_m}{n_m - 1 + \lambda} \tag{7.9}$$

Für eine gegen unendlich strebende Kettenlänge $n_m \to \infty$ ergibt sich ein Mobilität μ_∞:

$$\mu_\infty = \frac{k_o \lambda}{N} = \frac{k_m}{N} \tag{7.10}$$

Dies stimmt damit überein, dass im Fall eines unendlich langen Oligomers nur noch die Hüpfrate innerhalb des Oligomers eine Rolle spielt. Liegen keine Oligomere, sondern nur einzelne Gitterpunkte, d.h. eine Sequenz von einzelnen Molekülen, vor, gilt $n_m = 1$ und damit für die Mobilität μ_1:

$$\mu_1 = \frac{k_o}{N} \tag{7.11}$$

Hier ist nur noch die Hüpfrate zwischen den Oligomeren relevant, was wiederum der Erwartung entspricht. Die Mobilität kann somit geschrieben werden als:

$$\frac{\mu}{\mu_\infty} = \frac{n_m}{n_m - 1 + \lambda} \tag{7.12}$$

Der Verlauf dieser Kurve ist in Abbildung 7.11a für verschiedene Verhältnisse der Hüpfraten, unter anderem von λ = 100 wie in der Simulation verwendet, dargestellt.

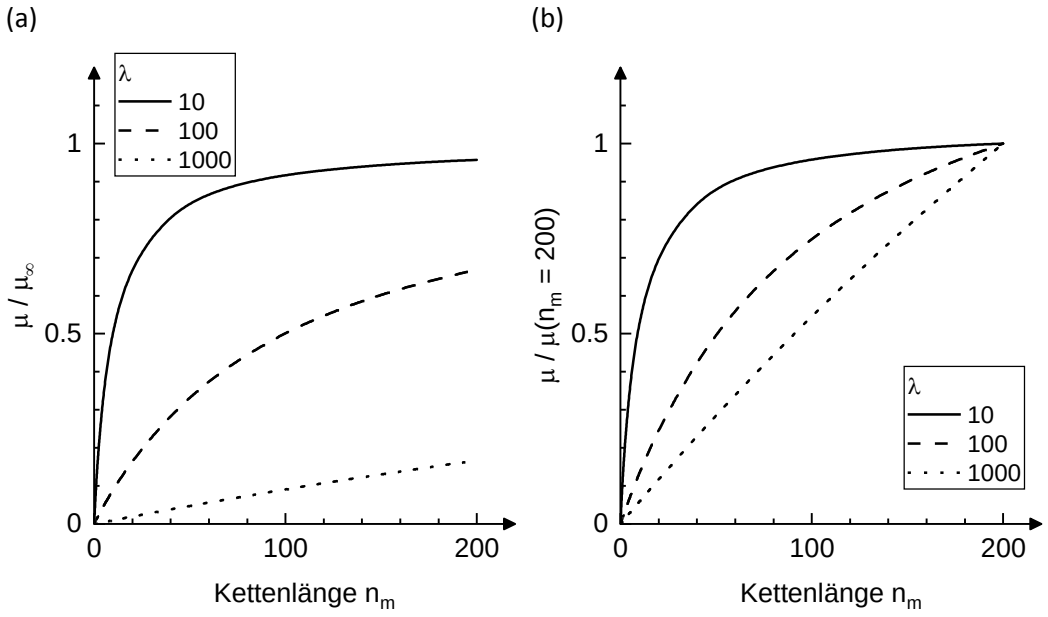

Abbildung 7.11 Theoretischer Verlauf der Mobilität eines eindimensionalen Strangs aus Oligomeren als Funktion der Kettenlänge n_m nach Gleichung (7.12) mit verschiedenen Verhältnissen der Hüpfraten λ wie angegeben. (a) μ geteilt durch μ_∞, (b) μ normiert auf den Wert von μ bei n_m = 200.

Die Kurve zeigt dabei ein ähnliches Verhalten wie die Stromdichte und die Mobilität als Funktion der Kettenlänge für Morphologie 1 in Abbildung 7.5 bzw. Abbildung 7.7. In Abbildung 7.11b, bei der Gleichung (7.12) auf den Wert bei n_m = 200 normiert wurde, ist zu erkennen, dass für μ = 1000 die Kurve einen nahezu linearen Verlauf hat. Um den funktionalen Zusammenhang in meiner Simulation noch genauer mit dem Verlauf der Mobilität eines eindimensionalen Oligomerstranges zu vergleichen, habe ich Gleichung (7.12) an den Verlauf der Mobilität als Funktion der Kettenlänge angepasst, was in Abbildung 7.12 in doppelt-logarithmischer Auftragung dargestellt ist. Die Parameter dieser Anpassung sind in Tabelle 7.3 aufgelistet.

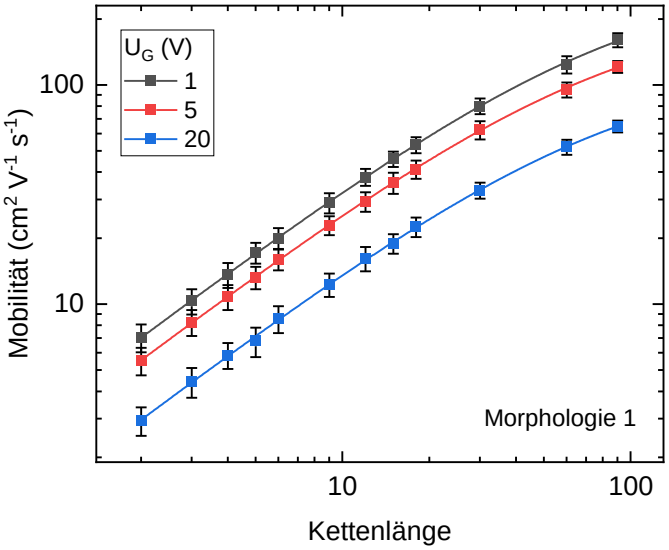

Abbildung 7.12 Mobilität als Funktion der Kettenlänge für Morphologie 1 in doppelt-logarithmischer Darstellung für verschiedene Gatespannungen wie angegeben. Die Daten entsprechen denen aus Abbildung 7.7. An die Daten wurde der funktionale Zusammenhang für einen eindimensionalen Oligomerstrang nach Gleichung (7.12) angepasst, deren Parameter in Tabelle 7.3 aufgetragen sind. Die Linien stellen die Ergebnisse dieser Anpassung dar.

Tabelle 7.3 Parameter der Anpassung der Gleichung (7.12) an die Daten in Abbildung 7.12, sowie die durch Simulation bestimmte Mobilität eines unendlich langen Oligomers μ'_∞ aus Tabelle 7.1.

U_G (V)	μ_∞ (cm^2 V^{-1} s^{-1})	λ	μ'_∞ aus Simulation (cm^2 V^{-1} s^{-1})
1	311 ± 5	87.9 ± 1.8	331 ± 7
5	226 ± 3	80.8 ± 1.5	263 ± 6
20	123 ± 2	82.2 ± 1.9	139 ± 4

Der Verlauf der Anpassung von Gleichung (7.12) an die Mobilität als Funktion der Kettenlänge passt augenscheinlich sehr gut. Die Werte für den Parameter μ_∞ stimmen hinreichend gut mit denen der Mobilität des unendlich langen Oligomers aus Tabelle 7.1 überein. Für den Parameter λ ergibt sich in der Anpassung in etwa ein Wert von 81 – 88. Das Verhältnis der Hüpfraten für Sprünge innerhalb und zwischen den Oligomeren beträgt 100. Somit ergibt sich in der Anpassung ein leicht geringerer Wert als in der Simulation, er kann jedoch als hinreichend übereinstimmend angesehen werden. Ein Grund für die leichte Abweichung kann in wechselseitigen

Abhängigkeiten der Anpassungsparametern liegen, wie sich später zeigen wird. Des Weiteren ist zu berücksichtigen, dass es sich bei der Simulation um ein dreidimensionales System handelt, die Berechnung aber nur ein eindimensionales System annimmt. Auch wenn kein Versatz benachbarter Oligomere vorliegt, können Sprünge in die y- oder z-Richtung möglicherweise zu einem scheinbar anderen Verhältnis der Hüpfraten führen.

Als nächstes gehe ich auf den Einfluss des Versatzes benachbarter Oligomere auf die Stromdichte ein. Der Vergleich der Morphologien untereinander hat gezeigt, dass Morphologie 1 durchwegs die kleinsten Werte der Stromdichte bei einer bestimmten Kettenlänge liefert. Die Morphologien 3*, 3 und 4 weisen in der Stromdichte nur vernachlässigbare Unterschiede auf. Das bedeutet zunächst, dass ein Versatz benachbarter Oligomere, egal welcher Art, prinzipiell förderlich für die Stromdichte ist. Erklärt werden kann dies dadurch, dass bei einer Morphologie mit Versatz benachbarter Oligomere ein Ladungsträger bei Sprüngen in y- oder z-Richtung ebenfalls Strecke in x-Richtung gewinnt, da er auf dem Oligomer, auf dem er sich dann befindet, weiter in x-Richtung vorankommen kann, ohne erneut zwischen Oligomeren springen zu müssen. Schematisch ist ein solcher möglicher Weg eines Ladungsträgers in Abbildung 7.13 dargestellt.

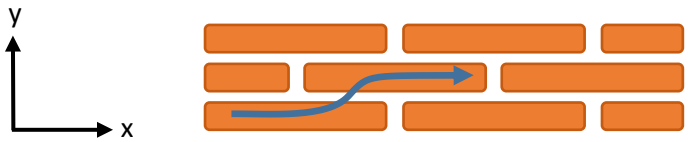

Abbildung 7.13 Schematische Darstellung des Ladungstransportweges bei Oligomeren, die einen Versatz benachbarter Oligomere entlang der y-Richtung aufweisen. In oranger Farbe sind die Oligomere angedeutet, der blaue Pfeil symbolisiert einen möglichen Weg eines Ladungsträgers.

Wird nun der Versatz benachbarter Oligomere in nur einer Richtung betrachtet, so hat sich gezeigt, dass mit einem Versatz benachbarter Oligomere nur in der y-Richtung eine höhere Stromdichte erreicht wird, als wenn der Versatz benachbarter Oligomere nur in z-Richtung vorliegt. Dies liegt daran, dass die Gatespannung ein elektrisches Feld in z-Richtung bewirkt, welches die Ladungsträger in den unteren Halbleiterschichten stärker konzentriert, in denen dann vorwiegend der Stromtransport stattfindet. Dies ist umso stärker, je größer die Gatespannung ist. Der oben beschriebene Streckengewinn eines Ladungsträgers in x-Richtung bei Sprüngen auf benachbarte Oligomere kann somit deutlich häufiger vorkommen, wenn der Versatz benachbarter Oligomere in y-Richtung vorliegt. Um den Einfluss des Versatzes benachbarter Oligomere auf

die Stromdichte quantitativ zu analysieren, wende ich folgendes vereinfachtes Modell an, welches in Abbildung 7.14 dargestellt ist.

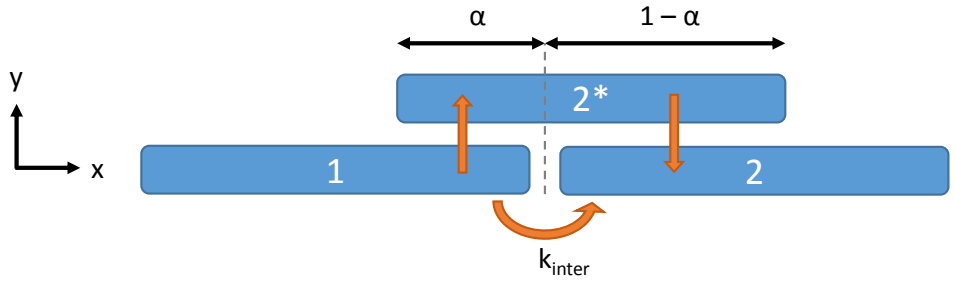

Abbildung 7.14 Skizze für den Übergang eines Ladungsträgers auf ein benachbartes Oligomer in x-Richtung über ein in y-Richtung benachbartes Oligomer.

Wenn ein Ladungsträger in x-Richtung von Oligomer 1 auf das benachbarte Oligomer 2 hüpft, so geschieht dies mit der Rate $k_{12} = k_{inter}$. Der Ladungsträger kann jedoch auch durch den Umweg über Oligomer 2* von Oligomer 1 nach Oligomer 2 gelangen. Da die Hüpfrate zwischen Oligomeren in y-Richtung ebenfalls k_{inter} beträgt, ergibt sich für die Rate k_{12^*} von Oligomer 1 nach Oligomer 2*

$$k_{12^*} = k_{inter} n_m \alpha \tag{7.13}$$

wobei n_m die Anzahl an Wiederholeinheiten im Oligomer und α der Anteil des Oligomers 2* ist, der mit Oligomer 1 in x-Richtung überlappt. Der Faktor $n_m \alpha$, mit dem die Rate multipliziert wird, ergibt sich dadurch, dass der Übergang zwischen den Oligomeren 1 und 2* innerhalb des überlappenden Bereiches der beiden Oligomere stattfinden kann. Für die Übergangsrate von Oligomer 2* auf Oligomer 2 ergibt sich analog:

$$k_{2^*2} = k_{inter} n_m (1 - \alpha) \tag{7.14}$$

Die Kehrwerte von Übergangsraten entsprechen Lebenszeiten ($k^{-1} = \tau$).[17] Für die Gesamtdauer von zwei aufeinanderfolgenden Prozessen müssen die einzelnen Lebenszeiten addiert werden. Die Gesamtrate von zwei aufeinanderfolgenden Prozessen ergibt sich damit aus dem Kehrwert der Summe der Kehrwerte der einzelnen Raten. Die gesamte Übergangsrate von Oligomer 1 über 2* auf Oligomer 2 beträgt somit:

$$k_{12^*2} = \left(\frac{1}{k_{inter} n_m \alpha} + \frac{1}{k_{inter} n_m (1-\alpha)}\right)^{-1} = k_{inter} n_m \alpha (1-\alpha) \tag{7.15}$$

Die gesamte Übergangsrate eines Ladungsträgers von Kette 1 nach 2, sowohl über den direkten Weg als auch über den Umweg über die Kette 2*, ergibt sich aus der Summe der beiden Raten:

$$k_{ges} = k_{inter} + k_{inter} n_m \alpha (1-\alpha) \tag{7.16}$$

Daraus kann das Verstärkungsverhältnis der Sprungraten berechnet werden:

$$\frac{k_{ges}}{k_{inter}} = 1 + n_m \alpha (1-\alpha) \tag{7.17}$$

Der parabelförmige Verlauf dieses Verhältnisses ist beispielhaft in Abbildung 7.15 dargestellt.

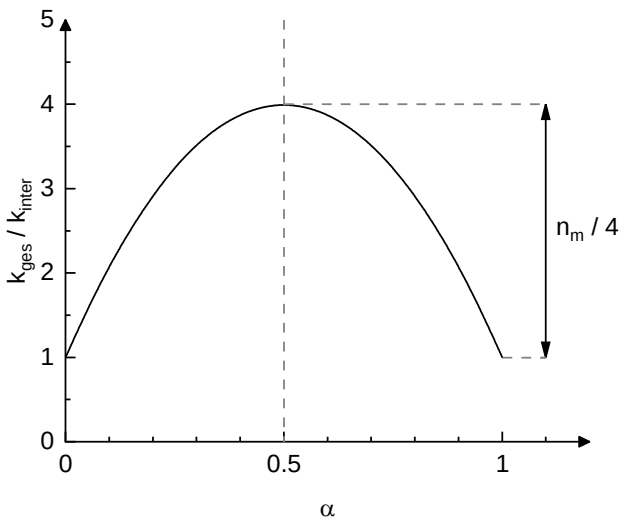

Abbildung 7.15 Verlauf des Verstärkungsverhältnisses der Sprungraten (Gleichung (7.17)) als Funktion des relativen Versatzes benachbarter Oligomere α beispielhaft für eine Kettenlänge von 12.

Das Verstärkungsverhältnis wird maximal für α = 0.5, wenn also die Oligomere in y-Richtung einen Versatz von genau der halben Kettenlänge aufweisen. Dies stimmt mit dem Maximum der Stromdichte als Funktion des Versatzes benachbarter Oligomere überein, welches sich bei der Hälfte der Kettenlänge befindet. Für eine Kettenlänge von 12 wird bei einem Versatz benachbarter Oligomere um genau die Hälfte der Kettenlänge ein Verstärkungsverhältnis von 4 erwartet. In Abbildung 7.6 ergibt sich bei einer Gatespannung von 1 V für Morphologie 2 ein

Verhältnis von 3.39 zwischen dem Maximum der Stromdichte und dem Wert der Stromdichte ohne Versatz benachbarter Oligomere, für Morphologie 2Y bei einer Gatespannung von 1 V ein Verhältnis von 2.72 und für Morphologie 2Y bei einer Gatespannung von 20 V ein Verhältnis von 2.00. Eine Abweichung von dem erwarteten Verhältnis von 4 kann daher kommen, dass in der Simulation zusätzlich Sprünge in z-Richtung auftreten, die aufgrund des elektrischen Feldes durch die Gatespannung sich anders verhalten wie Sprünge in der x-y-Ebene. Bei einer Gatespannung von 1 V ist dieser Effekt am geringsten. Wenn zudem kein Versatz benachbarter Oligomere in z-Richtung vorliegt, so kann das Verhältnis der Stromdichten nochmal stärker vom erwarteten Wert abweichen. Deswegen liegt das Verhältnis der Stromdichten bei der kleinsten Gatespannung mit Versatz benachbarter Oligomere in y- und z-Richtung am nächsten zum erwarteten Wert.

Die Stromdichte nimmt als Funktion des Versatzes benachbarter Oligomere erst merklich ab, sobald sich der Versatz deutlich von der Hälfte der Kettenlänge unterscheidet, also für besonders kleinen oder besonders großen Versatz benachbarter Oligomere. Das ist der Grund dafür, warum sich die Morphologien 3*, 3 und 4 untereinander nur geringfügig unterscheiden, da der Versatz benachbarter Oligomere nur für Randwerte deutliche Unterschiede liefert. Im Fall von Morphologie 4 treten auch die Randwerte des Versatzes benachbarter Oligomere auf, jedoch scheint dies statistisch nicht häufig genug der Fall zu sein, als dass die Stromdichte merklich beeinflusst würde. Insgesamt bedeutet dies aber, dass für das Erreichen hoher Stromdichten gar keine besonders geordnete Morphologie vorliegen muss, sondern die Morphologie in gewissem Rahmen robust gegenüber leichter Unordnung ist. Eine Vorzugsrichtung ist dabei jedoch notwendig.

Abschließend soll hier noch kurz auf die Abhängigkeit der Stromdichte und Mobilität von der Gatespannung eingegangen werden. Die Reihenfolge der Werte der Stromdichte folgt der Gatespannung, das heißt mit zunehmender Gatespannung steigt die Stromdichte. Dies liegt daran, dass mit höherer Gatespannung die Anzahl an Ladungsträgern gemäß Gleichung (3.6) steigt, die für den Stromfluss zur Verfügung stehen. Unter der Annahme, dass die Coulombwechselwirkung zwischen den Ladungsträgern zunächst vernachlässigbar ist, erhöht sich damit einfach die Anzahl an Ladungsträger, die pro Zeitintervall durch eine Querschnittsfläche des Halbleiters fließen, und somit eine höhere Stromdichte verursachen. Das Anwachsen der Stromdichte mit steigender Gatespannung wurde beispielsweise auch von Sharma et al. bei Monte-Carlo-Simulation von OFETs beobachtet.[102]

Wird die Stromdichte jedoch in eine Ladungsträgermobilität umgerechnet, so zeigt sich, dass die Mobilität mit der Gatespannung im hier simulierten Bereich bis 20 V abnimmt. Da die Ladungsträgerdichte in die Berechnung der Mobilität eingeht und die Stromdichte offensichtlich nicht im gleichen Maß wie die Anzahl der Ladungsträger mitwächst, lässt sich bereits hier vermuten, dass Coulombeffekte zwischen den Ladungsträgern eine wichtige Rolle spielen können. Darauf gehe ich detaillierter im nachfolgenden Kapitel 8 ein.

7.4. Ladungstransportsimulation für Oligomere mit gaußförmig verteilten Energien

Bisher habe ich lediglich den Fall von isoenergetischen Gitterplätzen betrachtet. In realen Systemen jedoch herrscht häufig eine gewisse Unordnung zwischen den einzelnen Molekülen. So können beispielsweise die Abstände zwischen den einzelnen Molekülen oder Polymeren statisch variieren, was durch Polarisationseffekte zwischen den Ladungsträgern und den sie umgebenden Medium eine statistische Verschiebung der Energien der Moleküle bewirkt.[17] Die sich ergebende Zustandsdichte ist dabei gaußförmig. Im nächsten Schritt untersuche ich nun den Fall von annähernd gaußverteilten Energien der Oligomere. Dafür verwende ich den in Kapitel 5.4.2 beschriebenen Algorithmus. Dabei haben alle Gitterplätze, die zum selben Oligomer gehören, die gleiche Energie, die Energien der Oligomere untereinander jedoch sind annähernd gaußverteilt. Die Standardabweichung der Energien beträgt dabei zunächst 50 meV. Analog zu der Untersuchung für den Fall von isoenergetischen Oligomeren untersuche ich auch hier den Zusammenhang zwischen der Stromdichte bzw. der Mobilität und der Kettenlänge bzw. dem Versatz benachbarter Oligomere für verschiedene Morphologien. In Abbildung 7.16 ist die Stromdichte als Funktion der Kettenlänge für verschiedene Gatespannungen und unterschiedliche Morphologien gezeigt.

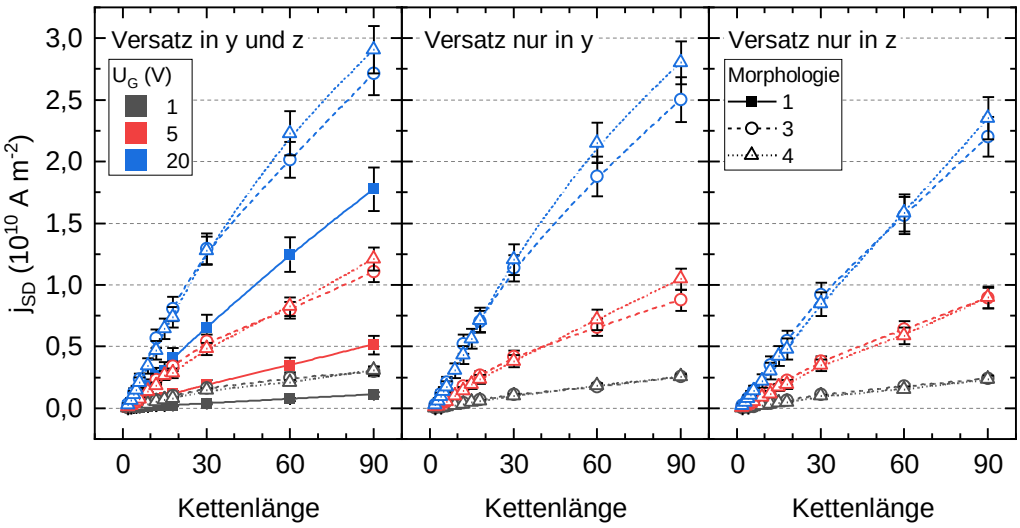

Abbildung 7.16 Stromdichte als Funktion der Kettenlänge der Oligomere für verschiedene Morphologien und Gatespannungen wie angegeben mit einer Standardabweichung der Energieverteilung der Oligomere von σ = 50 meV. Die eingezeichneten Linien dienen lediglich der Veranschaulichung.

Die Werte der Stromdichte folgen auch hier der Reihenfolge der Gatespannungen. Ebenfalls zeigt sich, dass ein Versatz benachbarter Oligomere in beide Richtungen eine größere Stromdichte bewirkt als ein Versatz benachbarter Oligomere nur in y-Richtung, der wiederum eine größere Stromdichte bewirkt als ein Versatz benachbarter Oligomere lediglich in z-Richtung. Der Verlauf der Stromdichte als Funktion der Kettenlänge verläuft augenscheinlich deutlich linearer als im Fall von isoenergetischen Oligomeren. Darauf wird weiter unten noch genauer eingegangen.

In Abbildung 7.17 ist der Verlauf der Stromdichte als Funktion des Versatzes benachbarter Oligomere dargestellt.

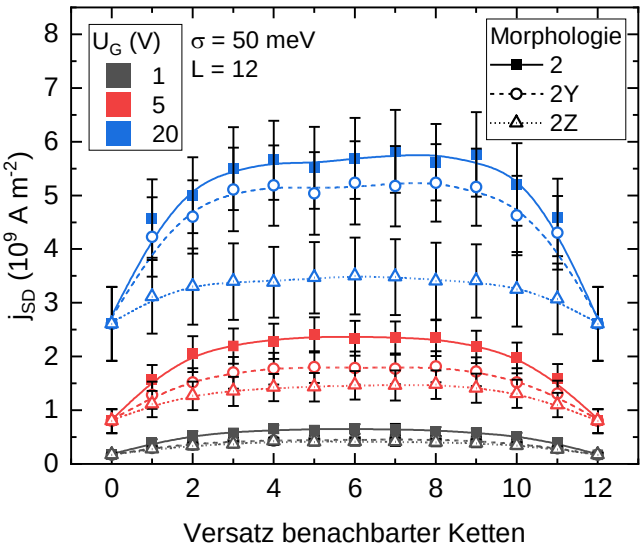

Abbildung 7.17 Stromdichte als Funktion des Versatzes benachbarter Oligomere für verschiedene Gatespannungen und unterschiedliche Morphologien wie angegeben, wobei die Breite der Energieverteilung der Oligomere 50 meV beträgt. Die Kettenlänge der Oligomere beträgt stets 12.

Auch hier zeigt sich ein ähnlicher Verlauf wie im Fall von isoenergetischen Oligomeren, wobei hier die Fehlerbalken der einzelnen Werte augenscheinlich größer sind als im Fall isoenergetischer Oligomere.

Die sich in Abbildung 7.16 ergebenden Werte der Stromdichte habe ich ebenfalls in eine Mobilität umgerechnet, wie in Abbildung 7.18 dargestellt ist.

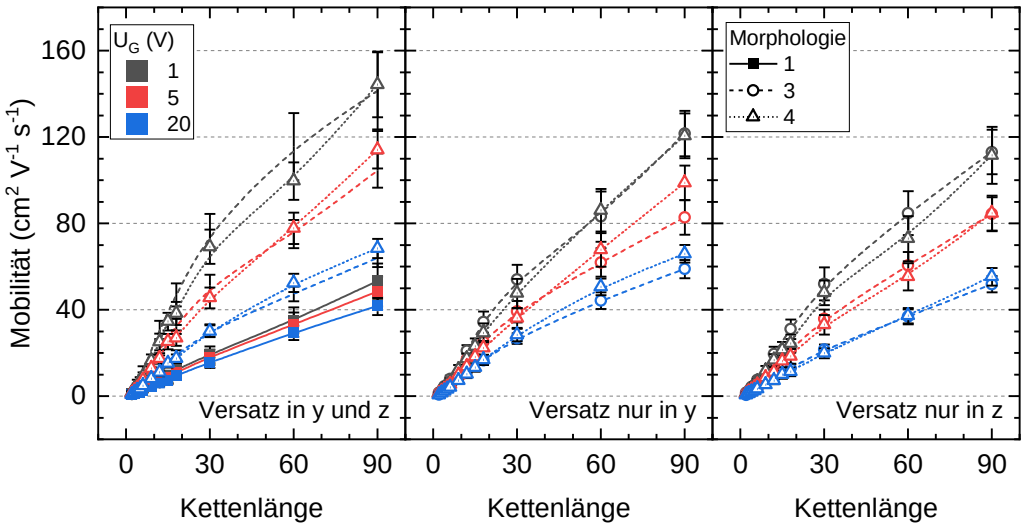

Abbildung 7.18 Mobilität als Funktion der Kettenlänge der Oligomere für verschiedene Gatespannungen und Morphologien wie angegeben, wobei die Breite der Verteilung der Energien der Oligomere 50 meV beträgt. Die eingezeichneten Linien dienen lediglich der Veranschaulichung.

Der Verlauf der Mobilität als Funktion der Kettenlänge ist hier augenscheinlich deutlich linearer als im Fall isoenergetischer Oligomere. Zudem nimmt die Mobilität insbesondere bei großen Kettenlängen und einer Gatespannung von 1 V merklich ab, wenn der Versatz zwischen benachbarten Oligomeren lediglich in y-Richtung statt in beiden Richtungen vorliegt, und nimmt weiter ab, wenn der Versatz benachbarter Oligomere nur in z-Richtung vorliegt. Der Unterschied der Mobilität zwischen einem Versatz benachbarter Oligomere in beiden Richtungen und lediglich in y-Richtung ist etwas größer als im Fall isoenergetischer Oligomere. Dies deutet darauf hin, dass im Fall einer Verteilung der Energien der Oligomere Sprünge in z-Richtung zunehmend relevanter werden, selbst wenn kein Versatz benachbarter Oligomere in dieser Richtung vorliegt. Ein Grund dafür kann sein, dass ein benachbartes Oligomer in z-Richtung energetisch günstiger liegt und somit für den Ladungstransport bevorzugt wird. Bei Morphologie 1 liegen die Kurven für die verschiedenen Gatespannungen nahe beieinander, sodass die Gatespannung hier nur einen geringen Einfluss auf die Mobilität hat. Dies zeigt, dass die Mobilität stärker durch die Verteilung der Energien der Oligomere beeinflusst wird als durch intrinsische Effekte wie beispielsweise die Coulombwechselwirkung.

Um den Einfluss der Breite der Verteilung der Energien der Oligomere besser quantifizieren zu können, habe ich zum Vergleich eine weitere Simulation mit einer größeren Standardabweichung der Energien der Oligomere durchgeführt. Die Energien der Oligomere sind ebenfalls aus einer diskreten Gaußverteilung mit einer Aufteilung der Energien in neun Intervalle entnommen. Die Breite der Verteilung beträgt jetzt jedoch 100 meV. In Abbildung 7.19 ist die Stromdichte als Funktion der Kettenlänge für verschiedene Gatespannungen und unterschiedliche Breiten der Verteilung der Energien der Oligomere dargestellt. Beispielhaft habe ich die Morphologien 1 und 4 gewählt.

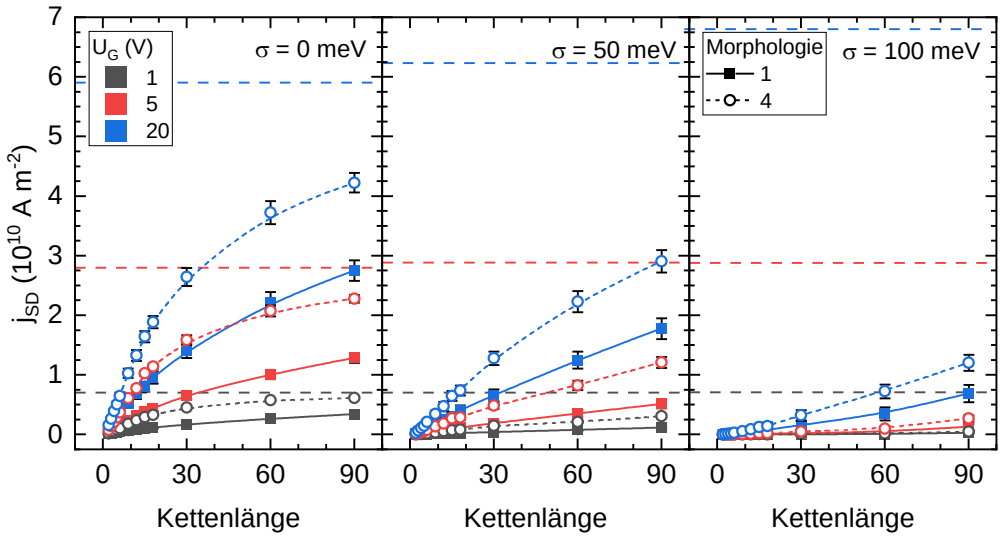

Abbildung 7.19 Stromdichte als Funktion der Kettenlänge der Oligomere für verschiedene Gatespannungen und Morphologien wie angegeben. Die Breite der Verteilung der Energien der Oligomere beträgt von links nach rechts 0, 50 und 100 meV wie angegeben. Die horizontalen gestrichelten Linien zeigen zu jeder Gatespannung die Stromdichte, die sich bei unendlich langen Oligomeren ergibt, wobei die zugehörigen Werte aus Tabelle 7.1 und Tabelle 7.4 entnommen sind. Die eingezeichneten Linien dienen lediglich der Veranschaulichung.

Es ist erkennbar, dass mit zunehmender Breite der Verteilung der Energien der Oligomere die Stromdichte abnimmt. Außerdem wird der Verlauf der Kettenlängenabhängigkeit mit zunehmender Breite der Verteilung der Energien der Oligomere augenscheinlich linearer, das heißt, die Krümmung der Kurven nimmt ab und wechselt bei σ = 100 meV für U_G = 1 und 5 V sogar das Vorzeichen. Der Vorzeichenwechsel der Krümmung ist nicht ohne weiteres nachvollziehbar und wird in dieser Arbeit nicht weiter diskutiert. Dies ist ein mögliches Forschungsfeld für weitere Arbeiten.

Um den Zusammenhang zwischen der Kettenlänge und der Ladungsträgermobilität zu untersuchen, habe ich eine weitere Simulation durchgeführt, bei der die Kettenlänge der Länge der Simulationsbox in x-Richtung entspricht, sodass durch die periodischen Randbedingungen unendlich lange Oligomere entstehen. Die sich daraus ergebenden Stromdichten und die mittels Tabelle 7.2 berechneten Mobilitäten sind in Tabelle 7.4 und Tabelle 7.5 aufgelistet. Zum Vergleich sind jeweils auch noch die Werte aus Tabelle 7.1 für isoenergetischen Oligomere angegeben.

Tabelle 7.4 Stromdichte für unendlich lange Oligomere mit unterschiedlicher Breite der Verteilung der Energien der Oligomere für verschiedene Gatespannungen wie angegeben.

U_G (V)	j_∞ (A m^{-2})		
	σ = 0 meV	σ = 50 meV	σ = 100 meV
1	$(7.06 \pm 0.16) \cdot 10^9$	$(7.1 \pm 0.2) \cdot 10^9$	$(7.11 \pm 0.17) \cdot 10^9$
5	$(2.80 \pm 0.08) \cdot 10^{10}$	$(2.89 \pm 0.08) \cdot 10^{10}$	$(2.88 \pm 0.08) \cdot 10^{10}$
20	$(5.90 \pm 0.17) \cdot 10^{10}$	$(6.23 \pm 0.17) \cdot 10^{10}$	$(6.8 \pm 0.2) \cdot 10^{10}$

Tabelle 7.5 Ladungsträgermobilität für unendlich lange Oligomere mit unterschiedlicher Breite der Verteilung der Energien der Oligomere für verschiedene Gatespannungen wie angegeben.

U_G (V)	μ_∞ (cm^2 V^{-1} s^{-1})		
	σ = 0 meV	σ = 50 meV	σ = 100 meV
1	331 ± 7	333 ± 11	333 ± 8
5	263 ± 6	271 ± 8	270 ± 8
20	139 ± 4	146 ± 4	160 ± 5

Dabei fällt auf, dass sowohl die Stromdichte als auch die Ladungsträgermobilität bei der jeweiligen Gatespannung nahezu die gleichen Werte für die verschiedenen Breiten der Verteilung der Energien der Oligomere aufweisen. Um die verschiedenen Standardabweichungen der Verteilungen der Energien der Oligomere miteinander und mit dem Modell für die Kettenlängenabhängigkeit nach Gleichung (7.12) zu vergleichen, habe ich in Abbildung 7.20 die Ladungsträ-

germobilität für den Fall von isoenergetischen Oligomeren, sowie mit einer Standardabweichungen von 50 bzw. 100 meV der Verteilung der Energien der Oligomere dargestellt und eine Kurvenanpassung nach Gleichung (7.12) an die Daten durchgeführt.

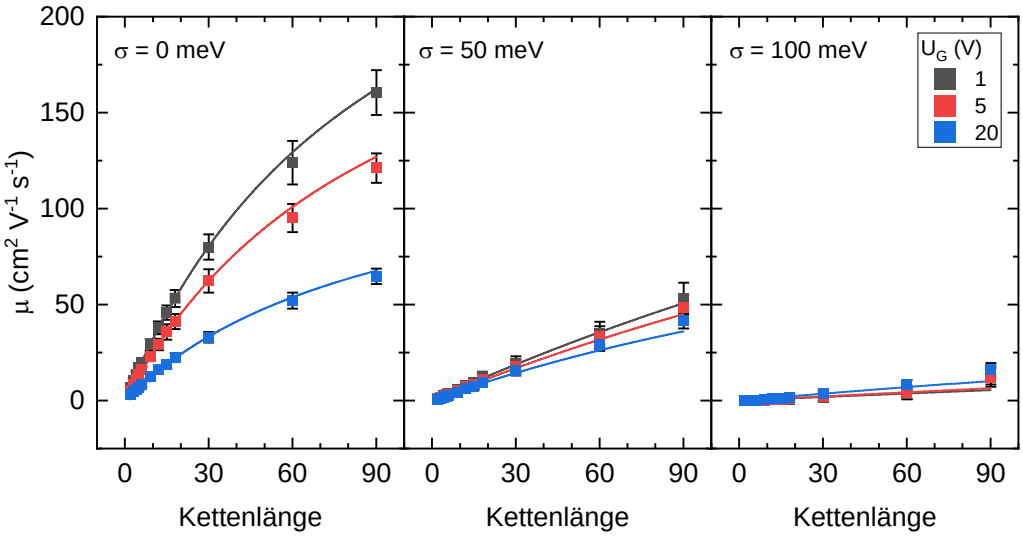

Abbildung 7.20 Mobilität als Funktion der Kettenlänge für verschiedene Gatespannungen und energetischen Breiten der Energieverteilung der Oligomere wie angegeben. Die Linien stellen dabei Anpassungen an die Daten nach Gleichung (7.12) unter Fixierung von μ_∞ auf den durch Simulation ermittelten Wert dar, deren Parameter in Tabelle 7.6 aufgeführt sind.

Bei der Anpassung an die Daten nach Gleichung (7.12) habe ich den Parameter μ_∞ auf den Wert des unendlich langen Oligomers für die jeweilige Breite der Verteilung der Energien der Oligomere fixiert und nur den Parameter λ angepasst. Dies liegt daran, dass bei den Anpassungen an die Daten mit einer Breite der Verteilung der Energien der Oligomere von σ = 50 meV und σ = 100 meV wechselseitige Abhängigkeiten zwischen den Parametern auftreten, sodass die Parameter nicht eindeutig bestimmt werden können. Um eine bessere Vergleichbarkeit zu erzielen, habe ich die Anpassung für die isoenergetischen Oligomere auch nochmal mit einer Fixierung von μ_∞ auf den durch Simulation ermittelten Wert durchgeführt. Die sich ergebenden Werte der Kurvenanpassung für den Parameter λ sind in Tabelle 7.6 eingetragen.

Tabelle 7.6 Parameter λ, der das Verhältnis der Hüpfraten innerhalb und zwischen den Oligomeren angibt, ermittelt aus der Anpassung von Gleichung (7.12) an die Daten in Abbildung 7.20. Der Parameter

μ_∞ wurde bei allen Breiten der Energieverteilung der Oligomere auf den Wert des unendlich langen Oligomers für die jeweilige Breite der Verteilung der Energien der Oligomere aus Tabelle 7.5 fixiert.

U_G (V)	λ		
	σ = 0 meV	σ = 50 meV	σ = 100 meV
1	94.5 ± 0.6	502 ± 8	(5.4 ± 1.0) · 10^3
5	98.4 ± 1.3	453 ± 14	(3.7 ± 0.6) · 10^3
20	96.0 ± 1.2	275 ± 16	(1.31 ± 0.19) · 10^3

Für isoenergetische Oligomere ergibt sich bei allen Gatespannungen ein Wert von 95 – 98 für den Parameter λ, annähernd dem Wert für das Verhältnis der Hüpfraten innerhalb und zwischen den Oligomeren. Kommt zusätzlich eine Verteilung der Energien der Oligomere hinzu, so steigt der Wert des Parameters λ deutlich mit zunehmender Breite der Verteilung der Energien der Oligomere an und nimmt für eine Verteilung der Energien der Oligomere mit steigender Gatespannung ab.

7.5. Diskussion

Ladungstransport in Polymeren, die geknäulte Ketten bilden können, wurde bereits früher untersucht.[84,135] In aktuellen OFETs, die neben einer hohen Ladungsträgermobilität auch gute Photolumineszenzeigenschaften aufweisen, werden jedoch lineare Oliogmere verwendet, sodass ein Verständnis das Ladungstransports in linearen Oligomeren relevant ist.[136] In diesem Kapitel habe ich verschiedene Aspekte des Ladungstransports von linearen Oligomeren adressiert. Zunächst habe ich den Einfluss der Kettenlänge von Oligomeren auf den Ladungstransport in OFETs untersucht. Die Stromdichte und Mobilität nimmt dabei mit der Kettenlänge bei allen untersuchten Morphologien streng monoton zu. Für den Fall eines eindimensionalen Oligomerstranges habe ich einen funktionalen Zusammenhang hergeleitet, der mit den Ergebnissen meiner Simulation sehr gut übereinstimmt.

Ebenso habe ich den Einfluss des Versatzes benachbarter Oligomere auf die Stromdichte und die Ladungsträgermobilität untersucht. In der Literatur gibt es nach meinem besten Wissen bis jetzt noch keine Untersuchung zum Einfluss des Versatzes benachbarter Oligomere auf die Transporteigenschaften. Der Versatz benachbarter Oligomere kann dabei verschiedene flüssig-

kristalline Phasen unterscheiden. Es ist bekannt, dass im Allgemeinen eine Ausrichtung der Polymere eine Erhöhung der Mobilität in Richtung der Ausrichtung, sowie gemittelt über alle Richtungen bewirkt.[84,130] Zudem ist auch bekannt, dass mittels einer smektischen Anordnung (Morphologie 1) hohe Ladungsträgermobilitäten erzielt werden können.[131,137] In meinen Untersuchungen habe ich herausgefunden, dass der Versatz benachbarter Oligomere unabhängig von der Art einen positiven Effekt auf den Ladungstransport hat. Ein Versatz benachbarter Oligomere lediglich in y-Richtung bewirkt dabei eine höhere Stromdichte und Mobilität als ein Versatz benachbarter Oligomere lediglich in z-Richtung. Der Versatz entlang der z-Richtung hat jedoch trotzdem eine nicht unerhebliche Auswirkung auf die Stromdichte, was bedeutet, dass auch die nächst höhere Schicht in z-Richtung mit in den Ladungstransport einbezogen ist. Dies ist im Einklang mit den Ergebnissen von Sharma et al.[102] Ob der Versatz benachbarter Oligomere regelmäßig im gesamten Halbleiter ist oder ein zufälliger Versatz benachbarter Oligomere vorliegt, hat dabei kaum einen Einfluss auf die Ladungstransporteigenschaften. Dieses Ergebnis kann durchaus von Nutzen sein für die nötige Sorgfalt bei der Herstellung von organischen Dünnschichthalbleitern, da beispielsweise die Prozessierung aus einer Lösung heraus komplex ist und somit eine genaue Kontrolle der Morphologie erschwert wird.[28] Ich habe zudem ein einfaches Modell für den Zusammenhang zwischen dem Versatz benachbarter Oligomere und der daraus resultierenden Verstärkung der Stromdichte aufgestellt. Das sich dabei ergebende Maximum der Stromdichte bei einem Versatz benachbarter Oligomere, der genau der Hälfte der Kettenlänge entspricht, stimmt mit meinem in der Simulation beobachteten Verlauf überein.

Außerdem habe ich den Einfluss einer zusätzlichen Verteilung der Energien der Oligomere auf den Ladungstransport untersucht. Sind die Energien der Oligomere verteilt mit einer gewissen Standardabweichung, so zeigt die Abhängigkeit der Stromdichte von der Kettenlänge einen Verlauf, der umso linearer wird, je höher die Standardabweichung der Verteilung der Energien der Oligomere ist. Der deutlich stärker lineare Verlauf der Mobilität im Fall von Oligomeren mit annähernd gaußförmig verteilten Energien kann dadurch erklärt werden, dass die energetische Verteilung die Sprünge zwischen den Oligomeren erschwert, da dabei zusätzlich eine thermische Aktivierung notwendig sein kann. Dies entspricht dem Fall eines größeren Verhältnisses zwischen den Sprüngen innerhalb und zwischen den Oligomeren, welches ich als Parameters λ nach Gleichung (7.8) definiert habe. In Abbildung 7.11b ist die Kurve für λ = 1000 deutlich linearer als die für λ = 100. Prinzipiell ist eine Abnahme der Mobilität mit steigender energetischer Unordnung des organischen Halbleiters zu erwarten. Für die Temperaturabhängigkeit der Mobilität gilt:[47]

$$\mu(T) = \mu_0 \exp\left(-\left(\frac{2}{3}\frac{\sigma}{k_B T}\right)^2\right) \tag{7.18}$$

Daraus folgt, dass bei sonst konstanten Parametern die Mobilität mit steigender energetischer Unordnung σ abnimmt. Diese Erwartung stimmt mit der Beobachtung in Abbildung 7.19 und Abbildung 7.20 überein, dass die Werte der Mobilität bei endlicher Kettenlänge mit steigender Breite der Verteilung der Energien der Oligomere abnimmt. Bei unendlich langen Oligomeren treten jedoch unabhängig von der Breite der Verteilung der Energien der Oligomere nahezu identische Werte auf. Dieses zunächst widersprüchliche Verhalten kann dadurch erklärt werden, dass bei unendlich langen Oligomeren in x-Richtung keinerlei Sprünge zwischen Oligomeren auftreten. Somit wird die Mobilität in dieser Richtung nicht durch Sprünge zwischen Oligomeren begrenzt, deren Rate von der Energiedifferenz zwischen den beteiligten Oligomeren und damit von der Breite der Verteilung der Energien der Oligomere abhängt.

Der Wert des Parameters λ kann mit dem verglichen werden, was als Eingabe in der Simulation verwendet wurde. Ich habe in der Simulation durchgehend ein Verhältnis von 100 verwendet. Im Fall von isoenergetischen Oligomeren stimmt der durch Kurvenanpassung ermittelte Wert gut mit diesem Verhältnis überein, wenn der Parameter μ_∞ auf den durch Simulation ermittelten Wert für unendlich lange Oligomere fixiert wird. Damit stimmt das Ergebnis der Simulation mit der erwarteten Abhängigkeit der Mobilität von der Kettenlänge überein. Wird jedoch eine Verteilung der Energien der Oligomere hinzugenommen, so steigt der Wert des Parameters λ deutlich über das in der Simulation vorgegebene Verhältnis der Sprungraten innerhalb und zwischen isoenergetischen Oligomeren an, und zwar umso mehr, je größer die Breite der Verteilung der Energien der Oligomere ist, was aus Tabelle 7.6 ersichtlich ist. Dies kann dadurch erklärt werden, dass für die Sprünge zwischen den Oligomeren zusätzlich eine energetische Barriere hinzukommen kann, die die Hüpfrate zwischen den Oligomeren reduziert und damit das Verhältnis der Sprünge innerhalb und zwischen den Oligomeren vergrößert. Die Zunahme des Parameters λ mit steigender Breite der Verteilung der Energien der Oligomere stimmt damit mit der Erwartung überein. Gleichzeitig nimmt bei Vorliegen einer Verteilung der Energien der Oligomere der Parameter λ mit steigender Gatespannung ab, was ebenfalls aus Tabelle 7.6 ersichtlich ist. Auffällig ist hierbei, dass bei unendlich langen Oligomeren die Stromdichte und die Ladungsträgermobilität nahezu unabhängig von der Breite der Verteilung der Energien der Oligomere ist. Das bedeutet, dass Sprünge zwischen den Oligomersträngen dabei keine Rolle spielen, sondern der Ladungstransport fast ausschließlich durch den Transport entlang eines Oligomerstranges in x-Richtung bestimmt wird. Bei endlicher Kettenlänge muss das nicht mehr not-

wendigerweise der Fall sein. Deswegen kann ein möglicher Grund für die Abnahme des Parameters λ mit steigender Gatespannung bei Vorliegen einer Verteilung der Energien der Oligomere sein, dass mit der Gatespannung das elektrische Feld in z-Richtung steigt und den Ladungstransport damit auf die unterste Schicht konzentriert. Der Ladungstransport wird dadurch zunehmend zweidimensional, wodurch weniger häufig Sprünge zwischen den Oligomeren in z-Richtung auftreten, bei denen zusätzlich eine Energiedifferenz auftreten kann. Dadurch nähert sich das effektive Verhältnis der Sprungraten wieder mehr dem intrinsischen Verhältnis der Sprünge innerhalb und zwischen den Oligomeren an. Ein Auffüllen von energetisch niedrigen Gitterplätzen muss hier nicht berücksichtigt werden, da in der Simulation nicht berücksichtigt wird, dass alle Gitterplätze eines Oligomers als besetzt markiert werden, sobald sich ein einziger Ladungsträger auf dem Oligomer befindet. In nachfolgenden Arbeiten sollte dies berücksichtigt werden.

7.6. Schlussfolgerung

In diesem Kapitel habe ich die Abhängigkeit der Ladungsträgermobilität von der Kettenlänge, dem Versatz benachbarter Oligomere und der energetischen Unordnung in oligomeren OFETs untersucht. Dabei habe ich herausgefunden, dass eine strenge Anordnung der Oligomere nicht nötig ist für eine hohe Ladungsträgermoblität, eine Vorzugsrichtung der Oligomere mit einem beliebigen Versatz benachbarter Oligomere genügt. Optimal ist ein Versatz benachbarter Oligomere, der genau der Hälfte der Kettenlänge entspricht. Für eine hohe Mobilität sind möglichst lange Oligomere (konjugierte Segmente) erforderlich mit einer niedrigen energetischen Unordnung der Oligomere. Ich habe zwei Modelle entwickelt, um zum einen den Zusammenhang zwischen der Mobilität und Kettenlänge in eindimensionalen Oligomersträngen und zum anderen die Verstärkung der Mobilität durch ein versetztes benachbartes Oligomer zu beschreiben. Meine Ergebnisse zeigen wie die Anordnung und die Kettenlänge von Oligomeren mit den Ladungstransporteigenschaften zusammenhängt und können als Grundlage für weitere Untersuchungen mit komplexeren Morphologien dienen. Ein Beispiel hierfür sind gestapelte Halbleiterschichten, in denen sich die Orientierung der Oligomere alternierend ändert.[138]

8. Der Einfluss der Coulombwechselwirkung auf die Transfercharakteristik von organischen Feldeffekttransistoren

8.1. Einleitung

Im vorangegangenen Kapitel habe ich beobachtet, dass die Ladungsträgermobilität mit steigender Gatespannung abnimmt, was zunächst der Erwartung widerspricht. Da möglicherweise Coulombeffekte eine Rolle spielen können, ist es naheliegend den Zusammenhang zwischen der Stromdichte bzw. der Mobilität und der Ladungsträgerdichte genauer zu untersuchen.

Die Ladungsträgermobilität wird normalerweise aus der Strom-Spannungskennlinie bestimmt, wobei das Shockley-Modell mit der sogenannten gradual-channel-Näherung zugrunde gelegt wird.[139] Eine Voraussetzung für eine akkurate Bestimmung der Mobilität im linearen bzw. Sättigungsbereich ist dabei ein linearer Zusammenhang zwischen dem Source-Drain-Strom und der Gatespannung bzw. der Quadratwurzel des Source-Drain-Stroms und der Gatespannung.[43] Bei der experimentellen Bestimmung der Ladungsträgermobilität können jedoch einige Stolpersteine auftreten. So zeigen beispielsweise über 55 % der veröffentlichten experimentellen Daten Abweichungen vom klassischen OFET Modell.[140] Die Ursachen für nichtlineare Strom-Spannungskennlinien von OFETs können unterschiedlicher Natur sein. Einige mögliche Gründe für bestimmte Typen von Abweichungen vom linearen Verhalten sind Kontaktwiderstandseffekte, Mobilitäten, die von der Ladungsträgerdichte abhängen, oder nicht gleichmäßige Source-Drain-Spannung in OFETs mit kurzem Kanal.[43] Als weiterer Grund für das Auftreten von Abweichungen vom linearen Verhalten der Strom-Spannungskennlinie wird das Einfangen von Ladungsträgern in Fallenzuständen an der Grenzschicht zum Gatedielektrikum oder in lokalisierten Zuständen am Rand der Zustandsdichte angeführt.[141] Für diese Arten von Abweichungen sind in Abbildung 8.1 charakteristische Strom-Spannungskennlinien gezeigt.

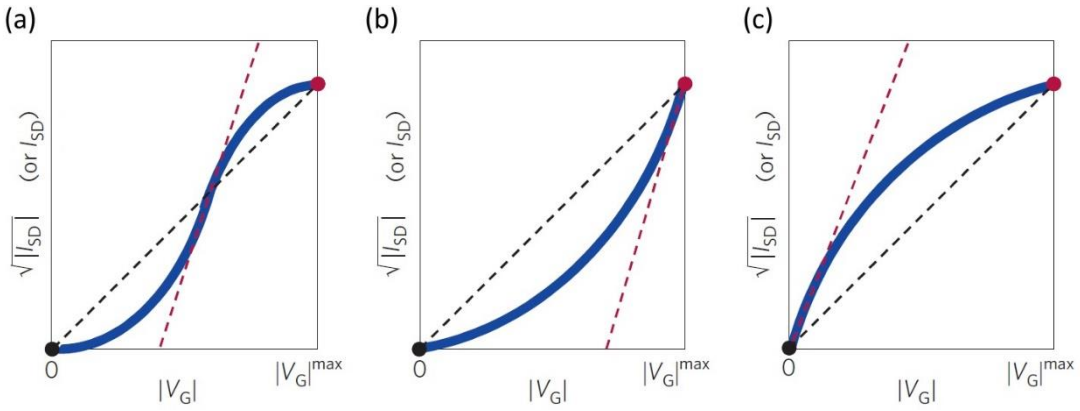

Abbildung 8.1 Einige der typischen Abweichungen vom linearen Verhalten der Strom-Spannungskennlinien in Feldeffekttransistoren. Mit $|V_G|^{max}$ ist die maximale im Experiment verwendete Gatespannung bezeichnet. Die rot gestrichelte Kurve zeigt mögliche Steigungen, die zur Bestimmung der Mobilität in der Literatur verwendet wurden. Schwarz gestrichelt ist die Transferkurve eines idealen Transistors angedeutet, der bei $|V_G|^{max}$ die gleiche maximale elektrische Leitfähigkeit hat. Angepasster genehmigter Abdruck.[43]

In Rubren-Einkristall-OFETs wurde zudem die Coulombwechselwirkung zwischen den Ladungsträgern im Transportkanal als eine Begründung dafür angeführt, dass eine Sättigung der Stromdichte bei hoher Ladungsträgerdichte mit stark polarisierbarem Dielektrikum beobachtet wurde,[44] was der Abweichung in Abbildung 8.1c entspricht.

Eine weitere Möglichkeit, die zu einem Auftreten eines Knicks in der Transfercharakteristik führen kann, ist ein quasi-eindimensionaler Ladungstransport, bei dem gleichzeitig die Ladungsträgerinjektion weit entfernt von der Halbleiter-Dielektrikum-Grenzschicht stattfindet wie beispielsweise in einem bottom-gate-top-contact-Transistors.[85] Dabei wurde eine große Dicke des Halbleiters sowie niedrige Hüpfraten senkrecht zur Kanalrichtung verwendet, was zu einem erhöhten Kontaktwiderstand führen kann.[9,142] Jedoch tritt die Abweichung von der linearen Strom-Spannungskennlinie bei hohen Gatespannungen auf, was eine Beteiligung der Coulombwechselwirkung bei diesem Effekt vermuten lässt.

Obwohl die Nichtlinearität der Strom-Spannungskennlinien bereits seit einiger Zeit untersucht wird, ist der Ursprung noch immer nicht vollständig verstanden.[9] Mit dem vorliegenden Kapitel will ich nun dazu beitragen das Verständnis hierfür zu vertiefen. Dazu untersuche ich den Einfluss der Coulombwechselwirkung, der Morphologie von Oligomeren und der energetischen Unordnung auf die Stromdichte und Ladungsträgermobilität mittels Monte-Carlo-Simulation in

einem OFET. Die Simulation des Ladungstransports ist hier eine geeignete Untersuchungsmethode, da dabei materialspezifische Effekte wie beispielsweise Degradationsprozesse, die bei der Messung der Strom-Spannungskennlinie auftreten können,[143] nicht berücksichtigt werden müssen. Ich verwende zudem eine Simulation ohne die Modellierung von Ladungsträgerinjektion und -extraktion und kann damit den Einfluss der Morphologie auf die Strom-Spannungskennlinie untersuchen ohne Effekte von Kontaktwiderständen berücksichtigen zu müssen. Dabei untersuche ich zunächst, wie sich verschiedene Berechnungsweisen der Coulombwechselwirkung auf die Strom-Spannungskennlinien auswirken, um auf den Einfluss der Coulombwechselwirkung schließen zu können. Anschließend untersuche ich verschiedene Morphologien mit der gleichen Berechnungsweise der Coulombwechselwirkung und deren Einfluss auf die Strom-Spannungskennlinien. Abschließend betrachte ich den Einfluss der energetischen Unordnung im organischen Halbleiter auf die Strom-Spannungskennlinien, sowie die Temperaturabhängigkeit der Ladungsträgermobilität.

8.2. Verwendete Simulationsparameter

Für die Untersuchung des Einflusses der Coulombwechselwirkung, der Morphologie und der energetischen Unordnung auf die Transfercharakteristik in OFETs verwende ich die kommerzielle Software Bumblebee und meine eigene Vielteilchensimulation Teil 2. Der schematische Simulationsbereich ist in Abbildung 8.2 dargestellt.

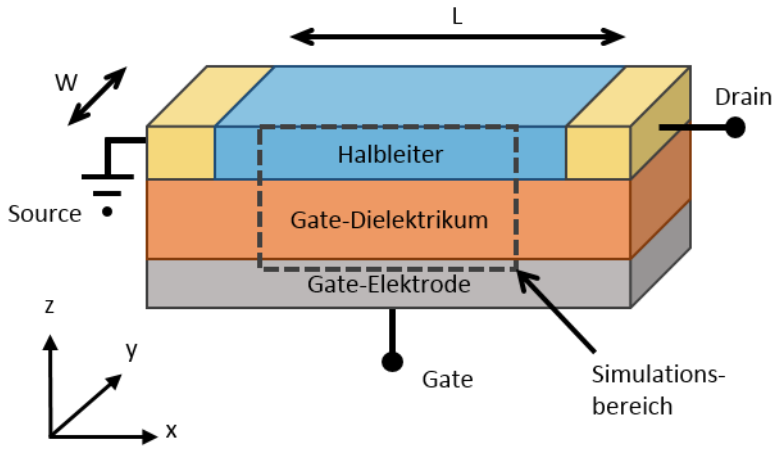

Abbildung 8.2 Schematische Darstellung eines Feldeffekttransistors inklusive des simulierten Bereichs.

Als Morphologien des Halbleiters in Kapitel 8.3 und 8.4 untersuche ich eine einkristalline Struktur und Oligomere mit unendlicher und endlicher Länge. Zur Untersuchung des Einflusses der energetischen Unordnung auf die Transfercharakteristik in Kapitel 8.5 verwende ich ein einfaches Punktgitter ohne weitere Morphologie wie ich es in Kapitel 5 verwendet habe. Der für die einkristalline Struktur verwendete Algorithmus ist in Kapitel 5.4.1 beschrieben. Dabei verwende ich den Algorithmus mit einem einzigen Kristallit, sodass alle Gitterpunkte des Halbleiters zum selben Kristallit gehören und keine Korngrenzen entstehen. Als Simulationsboxgröße habe ich hierfür 120 x 120 x 10 Gitterplätze gewählt. Für die Erzeugung der Morphologien mit Oligomeren verwende ich den in Kapitel 5.4.2 beschriebenen Algorithmus. Die Simulationsboxgröße beträgt dabei 180 x 180 x 10 Gitterplätze. Um Oligomere mit unendlich langer Kettenlänge zu erzeugen, verwende ich eine Kettenlänge von 180, was der Größe der Simulationsbox in x-Richtung entspricht, sodass durch die periodische Randbedingung in dieser Richtung keine Sprünge zwischen Oligomeren auftreten können. Die Kettenlänge der Morphologie mit endlich langen Oligomeren beträgt 12. In Abbildung 8.3 sind schematisch die Morphologien für einen einkristallinen Halbleiter, unendlich lange Oligomere und Oligomere mit endlicher Kettenlänge dargestellt.

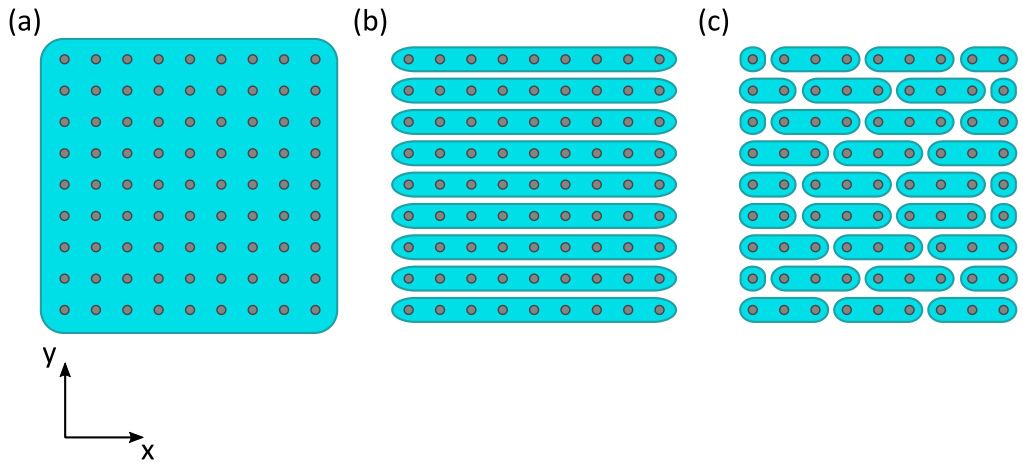

Abbildung 8.3 Schematische Darstellung der untersuchten Morphologien. (a) Einkristalliner Halbleiter, (b) Unendlich lange Oligomere durch periodische Randbedingung, (c) Oligomere mit endlicher Kettenlänge und zufälligem Versatz benachbarter Ketten. Die Gitterpunkte sind als graue Kreise angedeutet, die blauen Flächen markieren zusammenhängende (konjugierte) Bereiche innerhalb einer Morphologie.

Dabei sind konjugierte Bereiche innerhalb der Morphologie durch blaue Flächen dargestellt. Innerhalb dieser Bereiche finden Sprünge zwischen den Gitterpunkten mit einer Kopplungs-

konstante statt, die um einen Faktor 100 größer ist als für die Sprünge zwischen den konjugierten Bereichen. Dies ist bei allen drei Morphologien gleich. Bei der einkristallinen Morphologie treten nur Sprünge innerhalb des konjugierten Bereiches auf, bei unendlich langen Oligomere können in y- und z-Richtung zusätzlich auch Sprünge zwischen Oligomeren auftreten und bei endlich langen Oligomeren ebenfalls auch in x-Richtung.

Die weiteren Simulationsparameter lauten wie folgt: Als inverse Lokalisierungskonstante für Sprünge zwischen konjugierten Bereichen habe ich γ = 5 nm^{-1} bei einem Gitterabstand von 1 nm verwendet. Sprünge der Ladungsträger werden bis zum übernächsten Nachbar in jeder Richtung berücksichtigt. Die Dicke des Gatedielektrikums beträgt 100 nm, wobei die Dielektrizitätskonstante im Halbleiter und im Gatedielektrikum 4 beträgt. Ladungsträgerinjektion und -extraktion werden nicht simuliert. Die verwendeten Source-Drain-Spannungen sind im jeweiligen Abschnitt angegeben. Als Hüpfrate k_{ij} zwischen zwei Gitterplätzen i und j wurde die Miller-Abrahams-Hüpfrate verwendet (siehe Kapitel 5.3):[99]

$$k_{ij} = \nu_0 \exp\left(-2\gamma\left|r_{ij}\right|\right) \begin{cases} \exp\left(-\frac{\Delta E_{ij}}{kT}\right) & \Delta E_{ij} > 0 \\ 1 & \Delta E_{ij} \leq 0 \end{cases} \tag{8.1}$$

Dabei beträgt die attempt-to-hop Frequenz ν_0 = 2.203 · 10^{17} s^{-1} für Sprünge zwischen Oligomeren und ν_0 = 2.203 · 10^{19} s^{-1} für Sprünge innerhalb der Oligomere bzw. des Kristallits. Als Temperatur wurde durchgehend T = 300 K verwendet, sofern nicht anders angegeben. Der Cut-off-Radius, der bei der Berechnung der Coulombwechselwirkung verwendet wurde, beträgt 10 nm. Da in einem späteren Verlauf dieser Arbeit die Erkenntnis über die Wichtigkeit der korrekten Berücksichtigung der Coulombwechselwirkung gewonnen wurde (siehe Kapitel 5.3.4), habe ich die Transferkurve für die einkristalline Morphologie zusätzlich nochmal mit einem Cut-off-Radius von 40 nm berechnet. Obwohl sich der Verlauf der Transferkurven merklich unterscheidet, bleiben die wesentlichen Aussagen aus den Berechnungen mit einem Cut-off-Radius von 10 nm weiterhin gültig.

Für die Untersuchung des Einflusses der energetischen Unordnung auf die Transfercharakteristik in Kapitel 8.5 verwende ich teilweise Daten aus Kapitel 5.2.2, die ich bei einer Größe der Simulationsbox von 100 x 50 x 3 simuliert habe. Die attempt-to-hop Frequenz beträgt ν_0 = 10^{13} s^{-1}. Alle weiteren Parameter der Simulation sind identisch zu denen der Untersuchung des Einflusses der Coulombwechselwirkung und der Morphologie auf die Transferkurven in Kapitel 8.3 und 8.4. Um einen etwaigen Effekt der Größe der Simulationsbox zu untersuchen, habe ich in Abbildung 8.4 beispielhaft die Ladungsträgermobilität als Funktion der Temperatur für einen

einkristallinen Halbleiter mit meiner eigenen Vielteilchensimulation Teil 2 dargestellt. Wie später in Kapitel 8.5 erläutert, wurde die Mobilität der Anschaulichkeit halber mit einem Faktor von $2.203 \cdot 10^6$ skaliert, da für die attempt-to-hop Frequenz $\nu_0 = 10^{13}\,s^{-1}$ verwendet wurde. Als Größe der Simulationsbox habe ich einmal 100 x 50 x 3 wie in Kapitel 8.5 und einmal 100 x 100 x 10 verwendet.

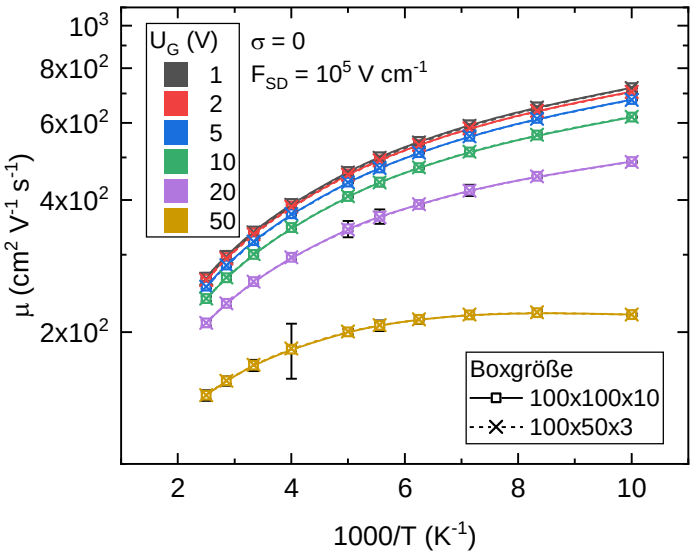

Abbildung 8.4 Ladungsträgermobilität als Funktion der Temperatur für zwei verschiedene Simulationsboxgrößen bei verschiedenen Gatespannungen wie angegeben. Die Größe der Simulationsbox beträgt einmal 100 x 100 x 10 und einmal 100 x 50 x 3. Die Linien dienen lediglich der Veranschaulichung.

Es zeigt sich, dass die Verläufe der Mobilität als Funktion der Temperatur nahezu deckungsgleich für beide untersuchten Größen der Simulationsbox sind. Die Größe der Simulationsbox hat somit bei meiner Simulationsmethode keinen Einfluss auf die Ergebnisse des Ladungstransports in OFETs.

8.3. Einfluss der Coulombwechselwirkung auf die Transferkurven

Im folgenden Abschnitt untersuche ich den Einfluss der Coulombwechselwirkung auf die Strom-Spannungskennlinie. Der Zusammenhang zwischen Stromdichte und Gatespannung wird auch als Transfercharakteristik oder Transferkurve bezeichnet. Dazu berechne ich Transferkurven ei-

nes einkristallinen OFETs mit verschiedenen Simulationsmethoden, die sich in der Art der Berücksichtigung der Coulombwechselwirkung unterscheiden. Ich verwende als Simulationsprogramm einmal Bumblebee mit zwei verschiedenen Cut-off-Radien und einmal meine eigene Vielteilchensimulation Teil 2, die ich miteinander vergleiche. Da ich bereits in Kapitel 5 die Berücksichtigung der Coulombwechselwirkung der verschiedenen Simulationsmethoden diskutiert habe, kann ich hier auf den Einfluss der Coulombwechselwirkung auf die Transferkurven schließen. Als Morphologie verwende ich hier nur einen einkristallinen Halbleiter, um Einflüsse durch die Morphologie auf den Ladungstransport ausschließen zu können. Die Größe der Simulationsbox beträgt bei der Simulation mit Bumblebee bei beiden Cut-off-Radien 120 x 120 x 10 und bei meiner eigenen Vielteilchensimulation Teil 2 100 x 100 x 10. Als Werte für die Source-Drain-Spannung habe ich bei Bumblebee 0.03, 0.06, 0.12, 0.24, 0.48 und 0.72 V verwendet und bei meiner eigenen Vielteilchensimulation Teil 2 0.025, 0.05, 0.1, 0.2, 0.4 und 0.6 V. Da ich Gatespannungen im Bereich 1 – 60 V bei Bumblebee bzw. 1 – 100 V bei meiner eigenen Vielteilchensimulation Teil 2 untersuche, ist die Voraussetzung für die Simulation eines Transistors im linearen Bereich, dass die Source-Drain-Spannung kleiner als die Gatespannung ist, durchwegs gegeben.

8.3.1. Ergebnisse für Simulationsmethoden mit unterschiedlicher Berücksichtigung der Coulombwechselwirkung

Zunächst berechne ich Transferkurven mit der Software Bumblebee unter Verwendung eines Cut-off-Radius für die Berechnung der Coulombwechselwirkung von 10 nm. Dies ist einerseits die Simulation, die bereits in einem früheren Stadium dieser Arbeit entstanden ist, und andererseits aufschlussreich, um die Auswirkung einer Überschätzung der Coulombwechselwirkung, wie sie bei einem Cut-off-Radius von 10 nm der Fall ist, zu untersuchen. In Abbildung 8.5 ist die Stromdichte und die Ladungsträgermobilität gemäß Gleichung (7.2) als Funktion der Gatespannung für verschiedene Source-Drain-Spannungen U_{SD} dargestellt. Die elektrische Feldstärke ergibt sich aus $F = U_{SD}/l_x$, wobei l_x die Größe der Simulationsbox in x-Richtung ist.

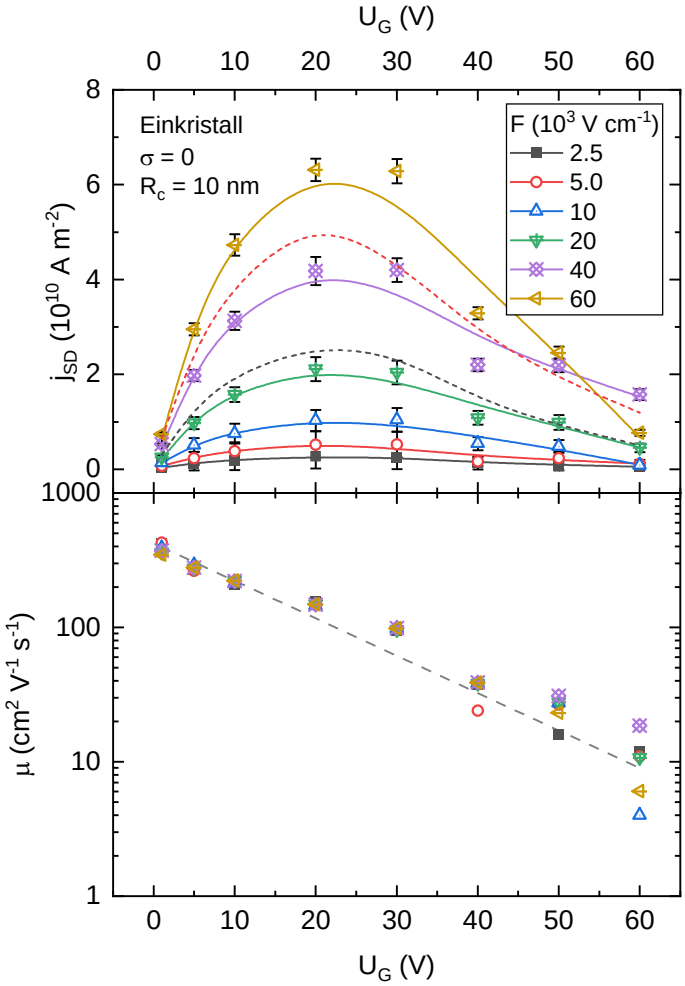

Abbildung 8.5 Stromdichte (oben) und Ladungsträgermobilität (unten) eines einkristallinen OFETs als Funktion der Gatespannung für verschiedene Source-Drain-Feldstärken wie angegeben. Der verwendete Cut-off-Radius in der Simulation beträgt R_C = 10 nm. Zur besseren Sichtbarkeit sind die beiden Linien der Stromdichte für eine Feldstärke von 2.5 und $5.0 \cdot 10^3$ V cm^{-1} nochmal gestrichelt mit einem Faktor 10 multipliziert eingezeichnet. Alle Linien, einschließlich der per Augenmaß durchgelegten Geraden für die Mobilität, dienen lediglich der Veranschaulichung.

Es zeigt sich, dass die Stromdichte bei allen Source-Drain-Spannungen zunächst mit steigender Gatespannung zunimmt, dann bei einer Gatespannung von 20 bis 30 V ein Maximum annimmt, und zu höherer Gatespannung hin wieder abnimmt. Dabei nehmen die Werte der Stromdichte mit steigender Source-Drain-Feldstärke zu. Für die Mobilität gilt, dass sie mit steigender Gatespannung annähernd exponentiell abnimmt, da sich in halblogarithmischer Darstellung ein annähernd linearer Abfall ergibt. Die Werte der Mobilität sind bei allen Source-Drain-Feldstärken in etwa gleich.

Da die Abnahme der Mobilität mit steigender Gatespannung darauf hindeutet, dass es einen möglichen Einfluss von Coulombwechselwirkung der Ladungsträger untereinander gibt, der mit steigender Gatespannung immer wichtiger wird, habe ich in Abbildung 8.6 die Ladungsträgerdichte, welche für die Berechnung der Mobilität wichtig ist, dargestellt. Dabei habe ich einmal den Mittelwert der Ladungsträgerdichte im gesamten Halbleiter aus der Anzahl der Ladungsträger berechnet und einmal den tatsächlichen Wert der Ladungsträgerdichte in der untersten Halbleiterschicht am Ende der Simulation ausgelesen. Die Ladungsträgerdichte habe ich für einen Wert der Source-Drain-Feldstärke von F_{SD} = 6 · 10^4 V cm^{-1} ausgelesen. Ich habe diesen Wert beispielhaft ausgewählt, da die Ladungsträgerdichte bei allen Source-Drain-Spannungen nahezu identisch ist. Die Unterscheidung zwischen der Ladungsträgerdichte gemittelt über den gesamten Halbleiter und ausgelesen in der untersten Schicht mache ich deshalb, da sich bei höherer Gatespannung die Ladungsträger vermehrt in der untersten Schicht befinden und somit eine höhere Konzentration an Ladungsträgern in der untersten Schicht vorliegt.[63] Das Verhältnis zwischen der Ladungsträgerdichte in der untersten Schicht und gemittelt über den gesamten Halbleiter ist ebenfalls in Abbildung 8.6 oben dargestellt. Für die Berechnung der Ladungsträgermobilität habe ich stets den Mittelwert der Ladungsträgerdichte über die gesamte Halbleiterschicht verwendet.

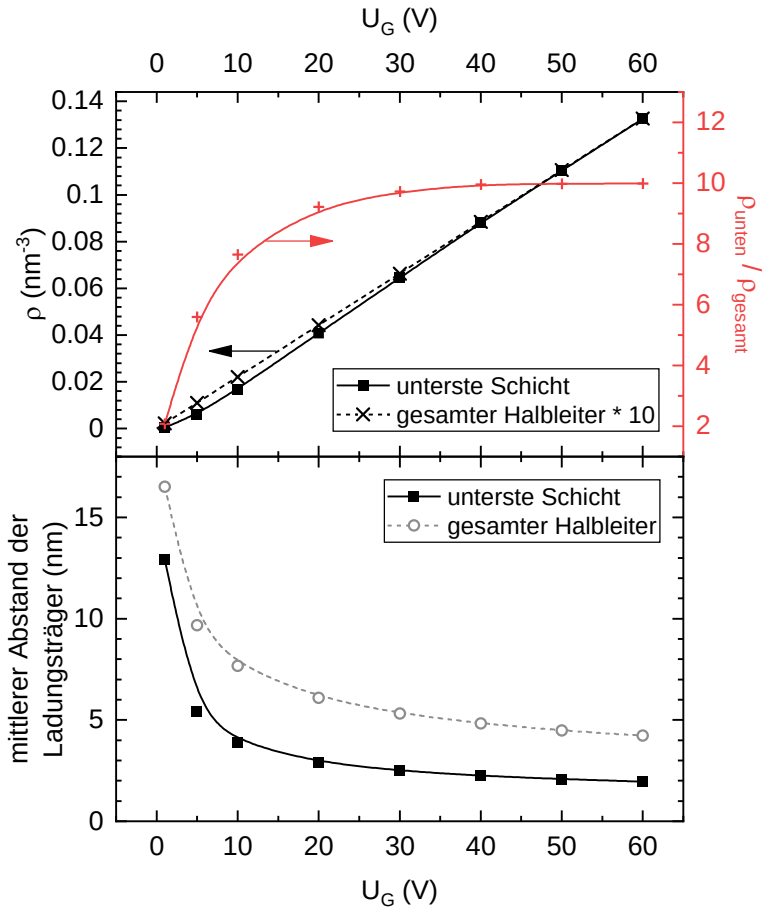

Abbildung 8.6 Ladungsträgerdichte (oben) und mittlerer Abstand der Ladungsträger (unten) im einkristallinen OFET als Funktion der Gatespannung einmal gemittelt über den gesamten Halbleiter und einmal ausgelesen in der untersten Halbleiterschicht bei einer Source-Drain-Feldstärke von F_{SD} = 6 · 10^4 V cm^{-1}. Die Ladungsträgerdichte, die sich gemittelt über den gesamten Halbleiter ergibt, wurde aus Anschaulichkeitsgründen in der oberen Abbildung mit einem Faktor 10 multipliziert, wobei die gestrichelte Linie eine Ursprungsgerade durch den Datenpunkt bei U_G = 60 V darstellt. Auf der rechten oberen y-Achse ist das Verhältnis zwischen der Ladungsträgerdichte in der untersten Schicht zu der im gesamten Halbleiter dargestellt. Alle eingezeichneten Linien dienen lediglich der Veranschaulichung.

Erkennbar ist, dass die Ladungsträgerdichte gemittelt über den gesamten Halbleiter linear mit der Gatespannung ansteigt. Die Ladungsträgerdichte in der ersten Schicht steigt dabei ebenfalls nahezu linear mit der Gatespannung an, nur bei kleiner Gatespannung gibt es eine leichte Abweichung vom linearen Verlauf. Das Verhältnis zwischen der Ladungsträgerdichte in der untersten Schicht und gemittelt über den gesamten Halbleiter nähert sich für hohe Gatespannungen

einem Wert von 10 an, was sich auch durch die annähernd gleich verlaufenden Kurven der Ladungsträgerdichte zeigt, wobei die im gesamten Halbleiter mit einem Faktor von 10 multipliziert wurde.

Um einen möglichen Einfluss von Coulombwechselwirkung der Ladungsträger untereinander bewerten zu können, habe ich den mittleren Abstand der Ladungsträger zueinander bestimmt. Dazu nehme ich an, dass sich die Ladungsträger gleichmäßig auf einem kubischen Gitter verteilen. Der mittlere Abstand a zwischen den Ladungsträgern beträgt dann:

$$a = (\rho)^{-\frac{1}{3}} \tag{8.2}$$

Die sich ergebenden Abstände zwischen Ladungsträgern für beide Fälle der Ladungsträgerdichte, gemittelt über den gesamten Halbleiter und ausgelesen für die unterste Halbleiterschicht, sind in Abbildung 8.6 unten dargestellt. Es ist erkennbar, dass mit zunehmender Gatespannung der mittlere Abstand schnell abfällt und dann immer mehr abflacht. Ab einer Gatespannung von 10 V wird ein Sättigungsverhalten beider Kurven beobachtet. Für die in der untersten Schicht ausgelesene Ladungsträgerdichte entspricht dies einem mittleren Abstand der Ladungsträger von etwa 4 nm. Die Gatespannung von 10 V ist auch der Wert, ab dem die Transferkurven in Abbildung 8.5 von einem linearen Verlauf abzuknicken beginnen.

Als nächstes berechne ich die Transferkurven eines einkristallinen OFETs mit der Software Bumblebee unter Verwendung eines Cut-off-Radius für die Berechnung der Coulombwechselwirkung von 40 nm. Dies ist wie in Kapitel 5.3.4 beschrieben ausreichend um die Coulombwechselwirkung in der Software Bumblebee annähernd korrekt zu berücksichtigen. In Abbildung 8.7 sind die Ergebnisse der Transferkurven mit einem Cut-off-Radius von 40 nm dargestellt, wobei alle anderen Parameter identisch zu denen in Abbildung 8.5 sind.

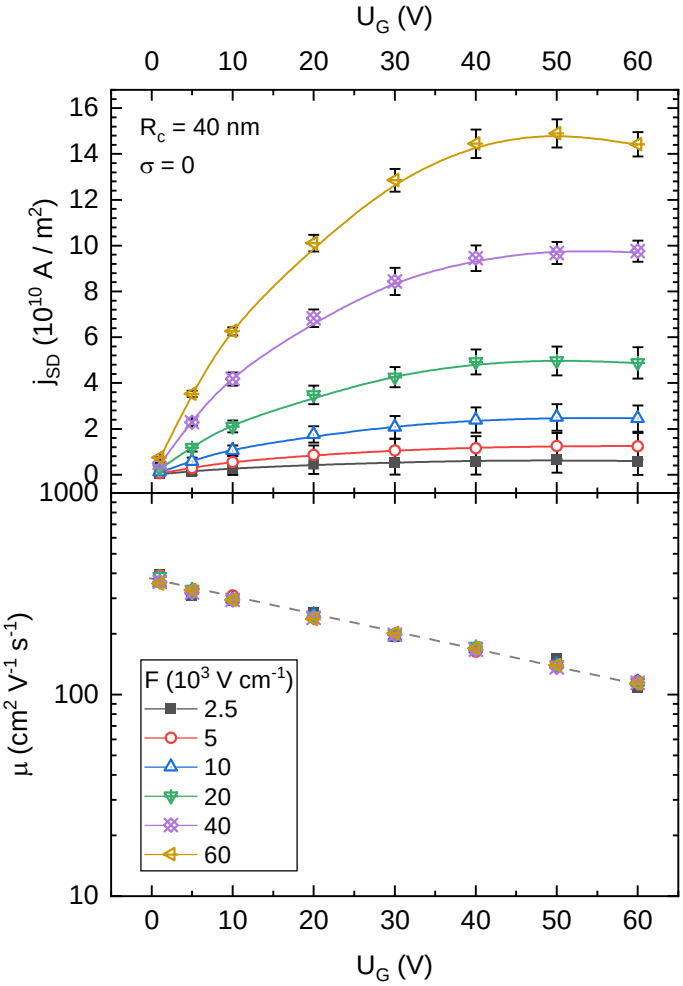

Abbildung 8.7 Stromdichte (oben) und Ladungsträgermobilität (unten) als Funktion der Gatespannung für einen einkristallinen OFET ermittelt mit der Software Bumblebee für verschiedene Source-Drain-Feldstärken wie angegeben. Der Cut-off-Radius beträgt R_c = 40 nm. Die Linien dienen lediglich der Veranschaulichung.

Der Verlauf der Stromdichte als Funktion der Gatespannung unterscheidet sich merklich von der mit einem Cut-off-Radius von 10 nm. Die Stromdichte nimmt zwar auch ein Maximum bei einer gewissen Gatespannung an, jedoch liegt dies deutlich höher bei einer Gatespannung von etwa 50 V. Der Maximalwert der Stromdichte ist ebenfalls deutlich größer. Die Mobilität weist zunächst augenscheinlich einen ähnlichen Verlauf wie bei einem Cut-off-Radius von 10 nm auf, lediglich die Steigung unterscheidet sich.

Die Ladungsträgerdichte gemittelt über den gesamten Halbleiter bei einer gewissen Gatespannung unterscheidet sich nicht bei verschiedenen Cut-off-Radien für die Berechnung der Coulombwechselwirkung. Da jedoch die Coulombwechselwirkung anders gewichtet wird, muss die Verteilung der Ladungsträger nicht notwendigerweise identisch sein. Um das zu prüfen,

habe ich wie bei einem Cut-off-Radius von 10 nm die Ladungsträgerdichte in der untersten Halbleiterschicht am Ende der Simulation ausgelesen. Diese ist für eine Source-Drain-Feldstärke von F_{SD} = 6 · 10^4 V cm^{-1} in Abbildung 8.8 dargestellt.

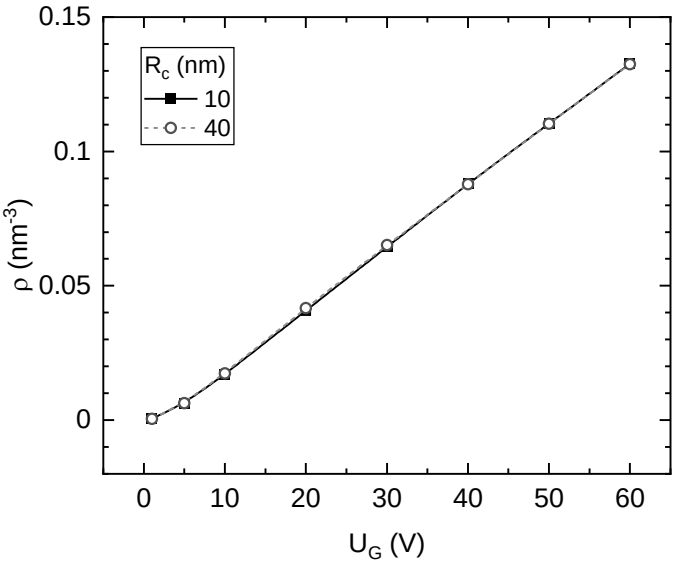

Abbildung 8.8 Ladungsträgerdichte in der untersten Halbleiterschicht ausgelesen aus der Simulation eines einkristallinen OFETs für die Simulationen mit einem Cut-off-Radius R_c von 10 und 40 nm. Die Ladungsträgerdichte wurde bei einer Source-Drain-Feldstärke von F_{SD} = 6 · 10^4 V cm^{-1} ausgelesen. Die Linien dienen lediglich der Veranschaulichung.

Die Ladungsträgerdichte bei der Verwendung eines Cut-off-Radius von R_c = 40 nm unterscheidet sich nicht von der mit einem Cut-off-Radius von R_c = 10 nm. Die mittels Abbildung 8.6 getroffenen Aussagen über den mittleren Abstand der Ladungsträger sind somit uneingeschränkt gültig.

In einem dritten Schritt berechne ich die Transferkurven mit meiner eigenen Vielteilchensimulation Teil 2, und vergleiche sie anschließend mit den beiden bisherigen Simulationsmethoden. Alle Parameter sind identisch zu denen der Simulation mit der Software Bumblebee. In Abbildung 8.9 sind die Ergebnisse der Transferkurven mit meiner eigenen Vielteilchensimulation Teil 2 dargestellt.

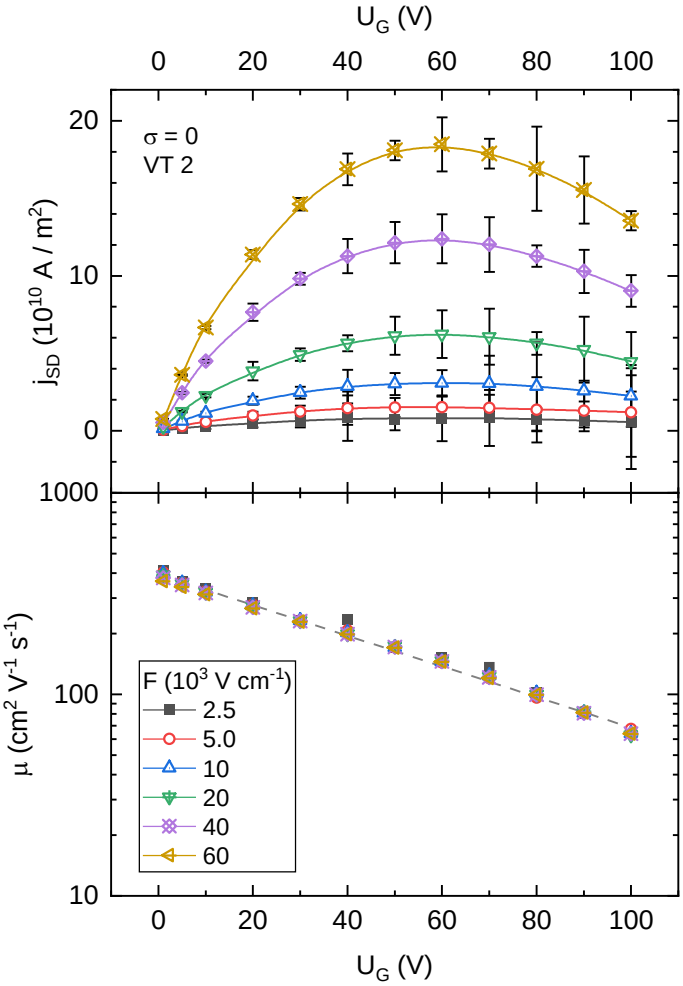

Abbildung 8.9 Stromdichte (oben) und Ladungsträgermobilität (unten) als Funktion der Gatespannung für einen einkristallinen OFET ermittelt mit meinem eigenen Simulationsprogramm für verschiedene Source-Drain-Feldstärken wie angegeben. Die Linien dienen lediglich der Veranschaulichung.

Der Verlauf der Stromdichte als Funktion der Gatespannung ist ähnlich zu der Simulation mit der Software Bumblebee mit einem Cut-off-Radius von 40 nm für die Berechnung der Coulombwechselwirkung. Das Maximum der Stromdichte liegt hier bei einer Gatespannung von 60 V. Um zu überprüfen, dass die Stromdichte für höhere Gatespannungen wieder abfällt, habe ich hier den Bereich der Gatespannung bis 100 V erweitert. Um die verschiedenen Simulationsmethoden besser miteinander vergleichen zu können, habe ich die Transferkurven der verschiedenen Simulationsmethoden bei der Source-Drain-Feldstärke $F_{SD} = 6 \cdot 10^4\ \mathrm{V\,cm^{-1}}$ in Abbildung 8.10 dargestellt.

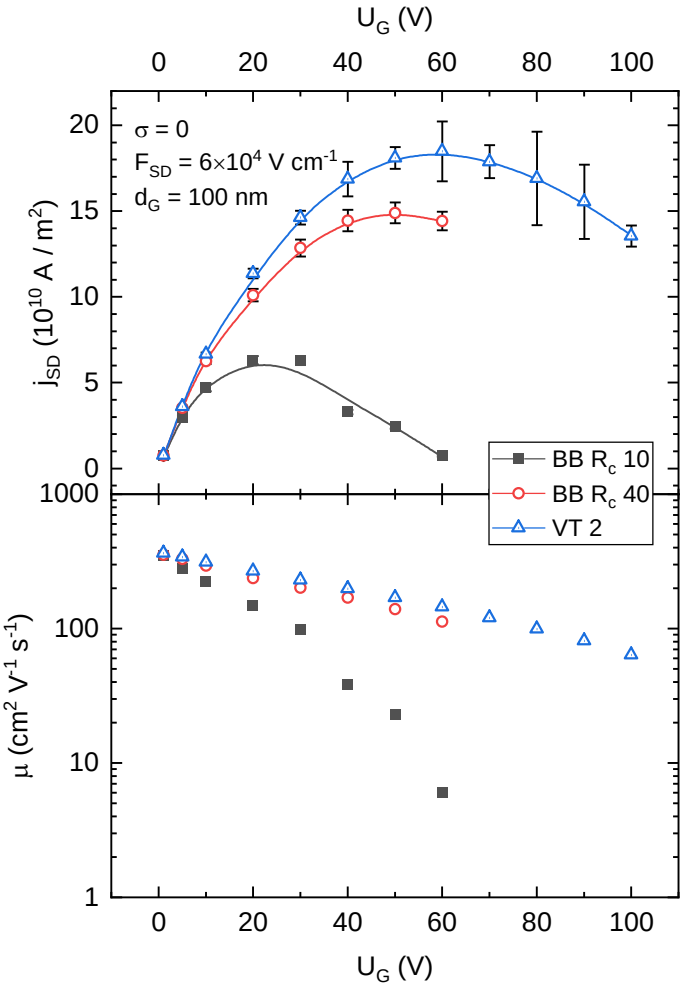

Abbildung 8.10 Stromdichte (oben) und Ladungsträgermobilität (unten) als Funktion der Gatespannung für einen einkristallinen OFET ermittelt mit den verschiedenen Simulationsmethoden: Bumblebee mit einem Cut-off-Radius von 10 nm (BB R_c 10), Bumblebee mit einem Cut-off-Radius von 40 nm (BB R_c 40) und mit meiner eigenen Vielteilchensimulation Teil 2 (VT 2). Die Linien dienen lediglich der Veranschaulichung.

Es zeigt sich, dass der Verlauf der Stromdichte und der Mobilität zwischen meiner eigenen Vielteilchensimulation Teil 2 und der Software Bumblebee mit einem Cut-off-Radius von 40 nm ähnlich ist. Der Verlauf der Stromdichte bei der Software Bumblebee liefert bei höherer Gatespannung leicht niedrigere Werte und das Maximum der Stromdichte wird augenscheinlich bei etwas niedrigerer Gatespannung angenommen. Die Simulation der Software Bumblebee mit einem Cut-off-Radius von 10 nm hingegen liefert einen Verlauf der Stromdichte, der bereits deutlich früher ein Maximum erreicht und dessen Werte merklich niedriger sind als bei den

beiden anderen Simulationsmethoden. Dies liegt wie in Kapitel 5.3.4 untersucht an der unzureichenden Berücksichtigung der Coulombwechselwirkung, was sich insbesondere bei hohen Gatespannungen bemerkbar macht.

8.3.2. Diskussion der Ergebnisse des Einflusses der Coulombwechselwirkung auf die Transferkurven

Die bei der Simulation mit der Software Bumblebee mit einem Cut-off-Radius von 10 nm beobachteten Transferkurven sind zunächst nicht das, was typischerweise für Transferkurven von OFETs beobachtet wird. In Abbildung 8.11 ist ein typischer Verlauf der Transferkurven im linearen und Sättigungsbereich des Transistors dargestellt.

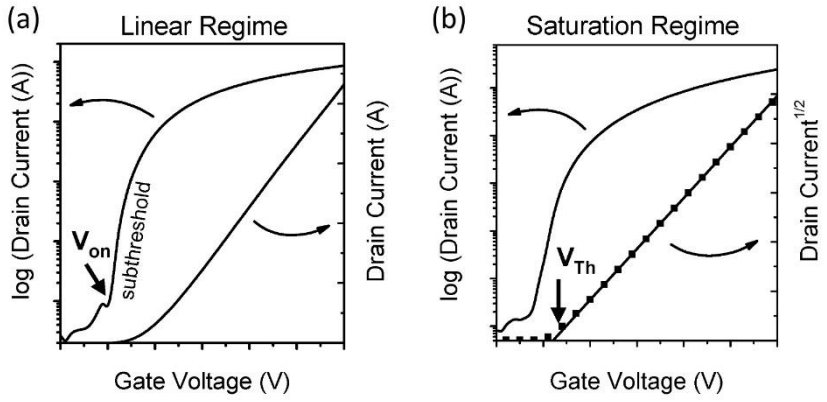

Abbildung 8.11 Typische Verlaufe der Source-Drain-Stromdichte als Funktion der Gatespannung. (a) Verlauf der Source-Drain-Stromdichte im linearen Bereich des Transistors, (b) Verlauf der Source-Drain-Stromdichte im Sättigungsbereich des Transistors. Der Verlauf der Stromdichte ist jeweils halblogarithmisch und linear bzw. quadratisch dargestellt. Genehmigter angepasster Abdruck.[144]

Im linearen Bereich, in dem sich meine Simulationen durchgehend befinden, steigt die Stromstärke ab einer gewissen Schwellenspannung linear mit der Gatespannung an. Dies ist im Gegensatz zu den Ergebnissen, die ich mit meiner Simulation mit der Software Bumblebee mit einem Cut-off-Radius von 10 nm erhalte. Aber auch bei den Simulationen mit einem Cut-off-Radius von 40 nm und mit meiner eigenen Vielteilchensimulation nimmt die Stromdichte bei einer größeren Gatespannung ein Maximum an, sodass die unzureichende Berücksichtigung der Coulombwechselwirkung bei einem Cut-off-Radius von 10 nm als alleiniger Grund für den abweichenden Verlauf der Stromdichte vom typischen Verhalten ausgeschlossen werden kann.

Experimentell wird jedoch für organische Halbleiter nicht immer der für anorganische Halbleiter typische lineare Verlauf der Stromstärke als Funktion der Gatespannung beobachtet. In Rubren-Einkristallen wurde bei hohen Gatespannung ein Sättigungsverhalten der Stromstärke beobachtet, wobei ein stark polarisierbares Gatedielektrikum verwendet wurde.[44,145] Der Verlauf der Stromstärke wird dabei durch ein Modell von Fröhlich-Polaronen mit Coulombwechselwirkung erklärt.[44] Aber auch in C_8-BTBT wird eine Abweichung vom linearen Verlauf der Stromstärke beobachtet und die Coulombwechselwirkung als Begründung dafür aufgeführt.[146] Wird ein Elektrolyt-Gatedielektrikum in Kombination mit verschiedenen organischen Halbleitern verwendet, so beobachtet man experimentell, dass die Stromstärke sogar ein Maximum annehmen kann, bevor sie bei höherer Gatespannung wieder abfällt, ähnlich zu meinen beobachteten Ergebnissen.[147–150] Koopmans et al. beobachten mittels Monte-Carlo-Simulation, dass die elektrische Leitfähigkeit, welche proportional zur Stromdichte ist, bei einer Ladungsträgerdichte von etwa 10^{-2} nm^{-3} ein Maximum annimmt.[103] Als Ursache für die Reduzierung der Leitfähigkeit bei hoher Ladungsträgerdichte identifizieren sie dabei die Coulombwechselwirkung. Dies stimmt qualitativ mit meinen Ergebnissen überein. Bei der Simulation verwenden sie jedoch einen Cut-off-Radius für die Coulombwechselwirkung, der bei einer Ladungsträgerdichte von 10^{-2} nm^{-3} 10 nm beträgt.[103] Wie ich bereits in Kapitel 5.3.4 gezeigt habe, hat der Cut-off-Radius insbesondere bei hohen Ladungsträgerdichten einen deutlichen Einfluss auf die Ladungsträgermobilität. In Abbildung 8.10 ist zudem erkennbar, dass die Position des Maximums der Stromdichte vom Cut-off-Radius abhängt. Die Coulombwechselwirkung hat somit einen Einfluss auf das Auftreten eines Maximums in der Stromdichte, jedoch sind dafür höhere Ladungsträgerdichten notwendig als von Koopmans et al. ermittelt.

Sharma et al. beobachten in ihren Monte-Carlo-Simulationen eines OFETs, dass bei einem Verhältnis von σ/kT = 2 die Mobilität mit steigender Ladungsträgerdichte bzw. Gatespannung abnimmt.[102] Die Abnahme der Mobilität beträgt dabei einen Faktor von etwa 1.3 bei einer Änderung der Gatespannung von 5 auf 60 V. In meiner Simulation mit der Vielteilchensimulation Teil 2 nimmt die Mobilität im gleichen Bereich der Gatespannung etwa um einen Faktor von 2.4 ab. Der Unterschied kommt dadurch zustande, dass ich einen Halbleiter ohne energetischer Unordnung (σ = 0) untersuche, bei dem keine Effekte durch eine energetische Unordnung berücksichtigt werden müssen. Zudem sei darauf hingewiesen, dass Sharma et al. in ihrer Untersuchung einen Cut-off-Radius von 15 nm verwenden, was das Ergebnis weiter verfälscht. Zhou et al. untersuchen einen Halbleiter in einer OLED-Struktur ohne energetischer Unordnung (σ = 0) mittels Monte-Carlo-Simulation und beobachten, dass die Ladungsträgermobilität für Ladungsträger-

dichten größer als 10^{-3} je Gitterpunkt abnimmt.[73] Bei der Berechnung der Coulombwechselwirkung zwischen den Ladungsträgern berücksichtigen sie alle Ladungsträger in den periodischen Fortsetzungen der Simulationsbox, sodass sich das Coulombpotential an einem Gitterpunkt aus Beiträgen des Ladungsträgers selbst und seiner periodischen Kopien zusammensetzt.[73] Dies ist ähnlich zu meiner Vorgehensweise bei der Vielteilchensimulation Teil 1 im Kapitel 5.2.1, was somit die Coulombwechselwirkung nicht korrekt berücksichtigt. Zudem verwenden sie eine Source-Drain-Feldstärke, die um mehr als eine Größenordnung über den von mir untersuchten Werten liegt, wodurch eine weitere Beeinflussung des Ergebnisses nicht ausgeschlossen werden kann. Liu et al. haben in Monte-Carlo-Simulationen von OLED-Strukturen ebenfalls festgestellt, dass die Coulombwechselwirkung bei Ladungsträgerkonzentrationen ab 10^{-3} pro Gitterpunkt und Source-Drain-Feldstärken unterhalb von 10^6 V cm^{-1} eine Verringerung der Mobilität in sandwichförmigen OLEDs bewirkt.[72] Bei einer Änderung der Ladungsträgerdichte von 10^{-3} auf 10^{-2} pro Gitterpunkt beobachten sie bei einer energetischen Unordnung von σ = 50 meV eine Abnahme der Mobilität um zwei Größenordnungen. Sie verwenden ebenfalls die Software Bumblebee mit einem Cut-off-Radius von 10 bzw. 15 nm je nach Ladungsträgerdichte.[72] Wie in Abbildung 8.10 erkennbar, führt dies zu einer stärkeren Reduktion der Mobilität als bei größerem Cut-off-Radius bzw. bei einer Simulation mit meiner Vielteilchensimulation Teil 2. Ein weiterer Nachteil ihrer Simulation von OLEDs ist, dass zur Erreichung von hohen Ladungsträgerdichten eine hohe Anzahl an Ladungsträgern notwendig ist, da sich diese über den gesamten Halbleiter verteilen. Dies erfordert einen hohen Rechenaufwand. Deswegen können Liu et al. auch bei einer Ladungsträgerdichte von 10^{-2} pro Gitterpunkt unterhalb einer Source-Drain-Feldstärke von 10^5 V cm^{-1} keine Mobilität mehr angeben. Bei der Simulation von OFETs werden jedoch leicht solche Ladungsträgerdichten erreicht, wie sich oben gezeigt hat.

Da für die Berechnung der Mobilität die Ladungsträgerdichte wichtig ist, gehe ich als nächstes kurz auf die Ladungsträgerdichten in meiner Simulation ein. Für den Fall, dass sich die Ladungsträgerdichte aus dem über den gesamten Halbleiter gemittelten Wert berechnet, ist zu erwarten, dass die Ladungsträgerdichte linear mit der Gatespannung gemäß Gleichung (3.6) ansteigt. Zur Erinnerung ist die Gleichung hier nochmal aufgeführt:

$$n = C_i U_G = \frac{\varepsilon_0 \varepsilon_G}{d_G} U_G \tag{8.3}$$

Die Beobachtung des linearen Verlaufs in Abbildung 8.6 stimmt mit dieser Erwartung überein. Bei der Ladungsträgerdichte in der untersten Schicht ist die Stärke des elektrischen Feldes durch

die Gatespannung im Vergleich zur Coulombwechselwirkung zwischen den Ladungsträgern entscheidend. Eine höhere Gatespannung bewirkt eine stärkere Konzentration der Ladungsträger in der untersten Schicht, aber auch eine stärkere Abstoßung der Ladungsträger untereinander durch die Zunahme in der Coulombwechselwirkung mit der Anzahl an Ladungsträgern. Der Unterschied zwischen der Ladungsträgerdichte, die sich gemittelt über den gesamten Halbleiter ergibt und dem in der untersten Halbleiterschicht ausgelesenen Wert am Ende der Simulation beträgt ab einer Gatespannung von 30 V ca. einen Faktor 10 bei der Simulation mit der Software Bumblebee. Das kann grob mit den Ergebnissen von Li et al. verglichen werden, die einen Faktor von etwa 8 zwischen der Ladungsträgerdichte in der ersten und in der zweiten Halbleiterschicht beobachten.[63] Allerdings verwenden sie einen organischen Halbleiter mit einer energetischen Unordnung von mindestens σ = 51 meV, wobei das Verhältnis der Ladungsträgerdichten mit steigender energetischer Unordnung sinkt.

Um die Ladungsträgerdichte in der untersten Halbleiterschicht einordnen zu können, habe ich in Abbildung 8.12 das Coulombpotential zwischen zwei Ladungsträgern als Funktion der Gatespannung aufgetragen. Den Abstand zwischen den beiden Ladungsträgern berechne ich nach Gleichung (8.2), wobei ich einmal die Ladungsträgerdichte gemittelt über den gesamten Halbeiter und einmal den in der untersten Halbleiterschicht ausgelesenen Wert verwendet habe. Außerdem ist die Potentialdifferenz zwischen zwei Halbleiterschichten in z-Richtung angegeben, die sich aus

$$\Delta\varphi_z = \frac{\varepsilon_G d_z}{2\varepsilon_{SC} d_G} U_G \tag{8.4}$$

ergibt. ε_G und ε_{SC} sind dabei die Dielektrizitätskonstanten des Gatedielektrikums und des organischen Halbleiters, d_z der Abstand zwischen zwei Halbleiterschichten in z-Richtung, d_G die Dicke des Dielektrikums und U_G die Gatespannung.

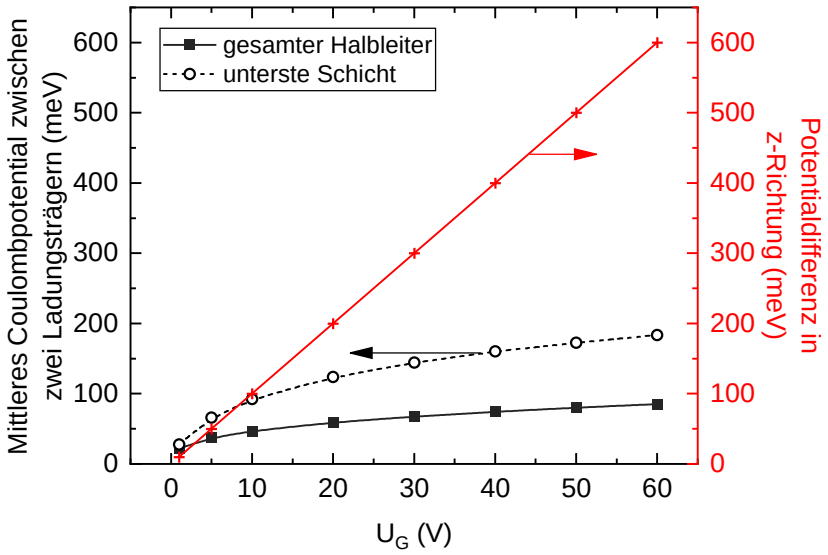

Abbildung 8.12 Coulombpotential zwischen zwei Ladungsträger im mittleren Abstand berechnet nach Gleichung (8.2) einmal unter der Verwendung der Ladungsträgerdichte, die sich gemittelt über den gesamten Halbeiter ergibt, und einmal unter Verwendung der Ladungsträgerdichte, die in der untersten Halbleiterschicht ausgelesen wurde. Zudem ist die Potentialdifferenz zwischen zwei Halbleiterschichten in z-Richtung dargestellt, die durch das elektrische Feld der Gateelektrode verursacht wird.

Bei einer Gatespannung von 10 V ergibt sich eine elektrische Feldstärke von $5 \cdot 10^5$ V cm^{-1} in z-Richtung im Halbleiter und eine Potentialdifferenz von 100 meV zwischen zwei Halbleiterschichten. Das ist somit viermal so hoch wie die thermische Energie bei Raumtemperatur von 25 meV. Bei dieser Gatespannung beträgt der mittlere Abstand der Ladungsträger 4 nm unter der Annahme, dass sich alle Ladungsträger in der untersten Halbleiterschicht befinden. Das Coulombpotential zwischen zwei Ladungsträgern beträgt bei diesem Abstand 90 meV unter der Annahme, dass sich alle Ladungsträger in der untersten Halbleiterschicht befinden, und liegt somit über der thermischen Energie bei Raumtemperatur und in der Größenordnung der Potentialdifferenz zwischen zwei Halbleiterschichten in z-Richtung. Da ab dieser Gatespannung die Potentialdifferenz größer als die mittlere Coulombwechselwirkung zwischen zwei Ladungsträgern ist, werden sich alle Ladungsträger vorzugsweise in der untersten Halbleiterschicht aufhalten, was den Effekt der Coulombwechselwirkung auf die Mobilität weiter verstärkt. Dies wird sich später in der Temperaturabhängigkeit der Mobilität eines einkristallinen Halbleiters zeigen.

Im späteren Verlauf meiner Arbeit hatte ich festgestellt, dass der Cut-off-Radius der Coulombwechselwirkung eine entscheidende Rolle bei der korrekten Bestimmung der Coulombwechselwirkung spielt. Mit der Software Bumblebee wird derselbe Trend für die Transferkurven wie in

meiner eigenen Vielteilchensimulation Teil 2 beobachtet. Die Coulombwechselwirkung wird jedoch in der Software Bumblebee etwas überschätzt, was sich daran zeigt, dass selbst bei einem Cut-off-Radius von 40 nm die Transferkurve etwas früher ein Maximum annimmt als mit meiner eigenen Vielteilchensimulation. Nichtsdestotrotz kann ich durch die unterschiedliche Berücksichtigung der Coulombwechselwirkung in den verwendeten Simulationsmethoden somit bestätigen, dass die Coulombwechselwirkung verantwortlich für die Nichtlinearität von Transferkurven sein kann.

8.4. Einfluss der Morphologie auf die Transferkurven

Im Folgenden habe ich die Stromdichte als Funktion der Gatespannung für verschiedene Morphologien bei unterschiedlichen Source-Drain-Feldstärken simuliert. Dafür habe ich die Software Bumblebee bei einem Cut-off-Radius von 10 nm für die Berücksichtigung der Coulombwechselwirkung verwendet, da dieser Teil bereits in einem früheren Stadium der Arbeit entstanden ist. Dieser ist wie bereits in Kapitel 5.3.4 erläutert nur unzureichend aussagekräftig für die korrekte Beschreibung der Coulombwechselwirkung zwischen den Ladungsträgern. Dies muss bei den Aussagen bezüglich der Coulombwechselwirkung berücksichtigt werden. Da ich hier jedoch vor allem auf Einflüsse der Morphologie im Halbleiter auf die Transferkurven eingehen möchte, ist diese Untersuchung zumindest dafür geeignet grundsätzliche Zusammenhänge zwischen der Morphologie und den Transferkurven festzustellen. Wie sich später zeigen wird, hat die Morphologie auf die Transferkurven keinen signifikanten Einfluss. Aus diesem Grund habe ich keine weitere Untersuchung mit einem größeren Cut-off-Radius durchgeführt, da sie mir als nicht besonders erfolgversprechend erschien. Als Morphologie des Halbleiters habe ich unendlich lange Oligomerketten, endlich lange, isoenergetische Oligomere, sowie endlich lange Oligomere mit einer Verteilung der Energien der Ketten untersucht. Die Größe der Simulationsbox beträgt dabei 180 x 180 x 10. Als Source-Drain-Spannungen habe ich 0.045, 0.09, 0.18, 0.36, 0.72 und 1.08 V verwendet, wobei die Bedingung für die Simulation eines Transistors im linearen Bereich bis auf die Kombination von Gatespannung 1 V bei einer Source-Drain-Spannung von 1.08 V immer gegeben ist. Dies muss bei Aussagen im letztgenannten Fall berücksichtigt werden.

8.4.1. Ergebnisse verschiedener Morphologien

Abbildung 8.13 zeigt die Stromdichte und die Mobilität für eine Morphologie mit unendlich langen Oligomeren, für eine Morphologie mit isoenergetischen Oligomeren der Kettenlänge 12, sowie für eine Morphologie mit Oligomeren der Kettenlänge 12 mit einer Breite der Verteilung der Energien der Ketten von 50 meV. Die teilweise relativ großen Fehlerbalken bei der Ladungsträgermobilität resultieren aus der Skalierung der Fehlerbalken der Stromdichte bei kleinen Source-Drain-Feldstärken mittels Fehlerfortpflanzung.

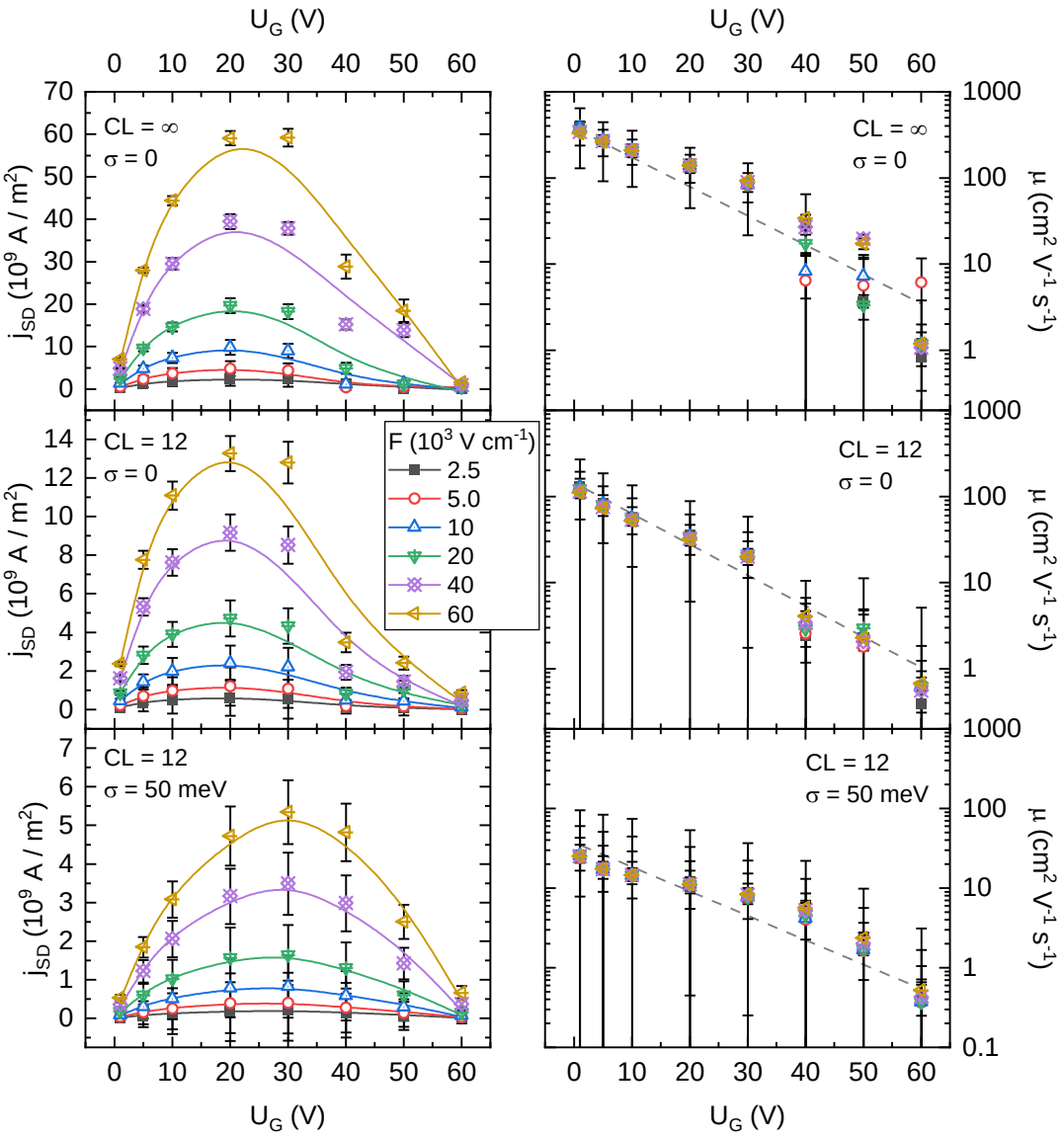

Abbildung 8.13 Stromdichte (links) und Ladungsträgermobilität (rechts) als Funktion der Gatespannung in einem Feldeffekttransistor für verschiedene Morphologien und Source-Drain-Feldstärken wie angegeben. Oben: Eine Morphologie mit unendlich langen Oligomeren. Mitte: Eine Morphologie mit isoenergetischen Oligomeren der Kettenlänge 12. Unten: Eine Morphologie mit Oligomeren der Kettenlänge 12 mit

einer Breite der Verteilung der Energien der Oligomere von 50 meV. Die Oligomere der mittleren und unteren Morphologie haben einen zufälligen Versatz benachbarter Ketten in y- und z-Richtung. Der verwendete Cut-off-Radius in allen Simulationen beträgt R_C = 10 nm. Alle Linien, einschließlich der per Augenmaß durchgelegten Geraden für die Mobilität, dienen lediglich der Veranschaulichung.

Es ist erkennbar, dass bei allen Morphologien ähnlich wie im Fall des einkristallinen Halbleiters mit steigender Gatespannung zunächst die Stromdichte steigt, dann ein Maximum erreicht, bevor sie wieder bis etwa zu ihrem ursprünglichen Wert abfällt. Dies ist insbesondere bei hohen Source-Drain-Feldstärken erkennbar. Die Werte der Mobilität sind bei allen Morphologien nahezu unabhängig von der Source-Drain-Feldstärke und fallen mit steigender Gatespannung auf halblogarithmischer Skala annähernd linear ab. Die Stromdichte und die Mobilität nimmt dabei von der Morphologie mit unendlich langen Oligomeren zur Morphologie mit endlich langen, isoenergetischen Oligomeren ab und verringert sich weiter zur Morphologie mit endlich langen Oligomeren mit einer Verteilung der Energien der Ketten. Bei der Morphologie mit endlich langen Oligomeren mit einer Verteilung der Energien der Ketten liegt das Maximum der Stromdichte zusätzlich leicht zu höheren Gatespannungen hin verschoben.

Um die Werte genauer miteinander vergleichen zu können, habe ich die Stromdichte und die Mobilität als Funktion der Gatespannung für die Source-Drain-Feldstärke F_{SD} = 6 · 10^4 V cm^{-1} in einem Diagramm zusammen mit der Transferkurve eines Einkristalls aus Abbildung 8.5 in Abbildung 8.14 dargestellt.

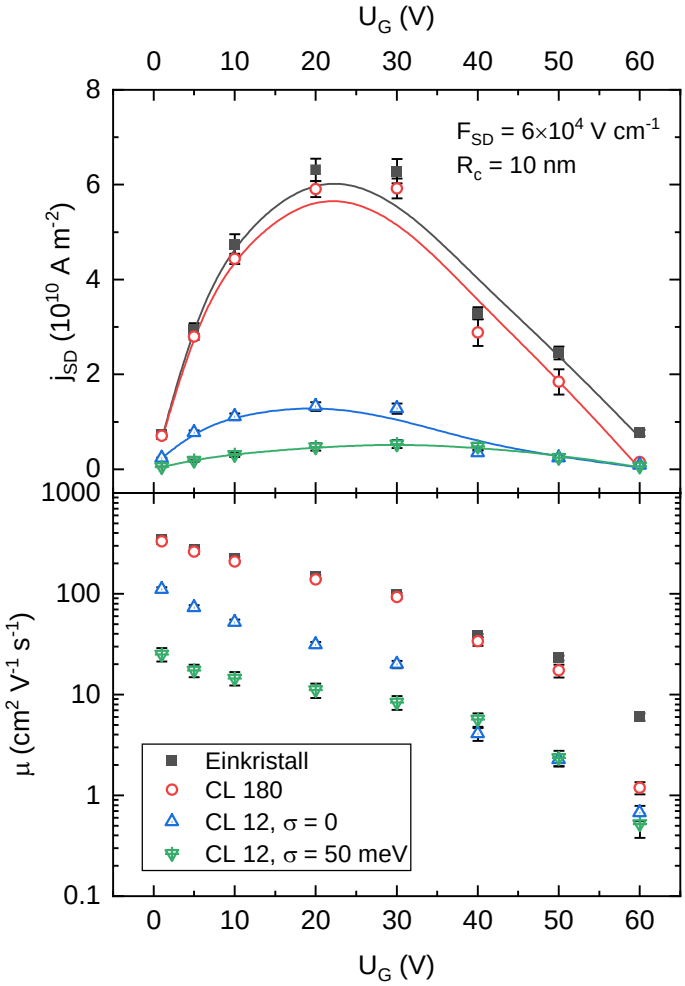

Abbildung 8.14 Vergleich der Stromdichte und Ladungsträgermobilität als Funktion der Gatespannung für verschiedene Morphologien wie angegeben. Die Source-Drain-Feldstärke beträgt für alle Kurven F_{SD} = 6 · 10^4 V cm^{-1}. Der Cut-off-Radius der Coulombwechselwirkung beträgt 10 nm. Die Linien dienen lediglich der Veranschaulichung.

Es zeigt sich, dass der Verlauf der Stromdichte beim Einkristall und bei unendlich langen Oligomeren nahezu identisch ist. Bei endlich langen Oligomeren mit und ohne Verteilung der Energien der Ketten liegen die Werte der Stromdichte und der Mobilität deutlich unterhalb der des Einkristalls und der unendlichen langen Oligomeren.

8.4.2. Diskussion der Ergebnisse der Transferkurven verschiedener Morphologien

Der funktionale Zusammenhang und die Werte der Stromdichte als Funktion der Gatespannung unterscheiden sich nahezu nicht zwischen der einkristallinen Morphologie und der Morphologie mit unendlich langen Oligomeren. Das bedeutet, dass die zusätzliche Vorzugsrichtung des Ladungstransports entlang der Feldrichtung im Fall der Morphologie mit unendlich langen Oligomeren die Strom-Spannungskennlinie nicht beeinflusst. Dies ist zunächst im Gegensatz zu der Beobachtung von Li und Brédas, die feststellen, dass die Strom-Spannungskennlinie durch einen eindimensionalen Ladungstransport, ähnlich dem der unendlich langen Oligomere, beeinflusst wird.[85] Allerdings führen sie als Voraussetzung für die Änderung in der Strom-Spannungskennlinie an, dass die Ladungsträgerinjektion und -extraktion weit entfernt von der eigentlichen Ladungstransportschicht stattfinden muss, wie dies beispielsweise bei einer bottom-gate-top-contact Struktur (siehe Abbildung 3.4) der Fall ist. Dies ist in meiner Simulation nicht der Fall, da Ladungsträgerinjektion und -extraktion nicht berücksichtigt werden. Für den Fall einer Ladungsträgerinjektion und -extraktion in die ladungstransportführende Schicht wie bei einer bottom-gate-bottom-contact Struktur beobachten auch Li und Brédas keine Veränderung der Strom-Spannungskennlinie.

Bei einer Hinzunahme von endlichen Oligomeren im Vergleich zur Morphologie mit unendlich langen Oligomeren tritt keine nennenswerte Änderung der Form der Strom-Spannungskennlinie auf. Die kleinere Stromdichte bei der Morphologie mit endlich langen Oligomeren kann durch die niedrigere Rate für Sprünge zwischen Oligomeren im Vergleich zu Sprüngen innerhalb von Oligomeren begründet werden. Die Stromdichte reduziert sich nochmals weiter im Fall einer Verteilung der Energien der Ketten, was die Rate für Sprünge zwischen Oligomeren weiter verringert. Eine mögliche Erklärung dafür, warum das Maximum der Stromdichte bei Oligomeren mit einer Verteilung der Energien der Ketten weiter zu höherer Gatespannung hin verschoben ist als bei den übrigen Morphologien, kann sein, dass der Ladungstransport durch die energetische Unordnung beeinflusst wird. Dadurch fällt die Coulombwechselwirkung zwischen den Ladungsträgern weniger stark ins Gewicht und wird erst bei höherer Gatespannung sichtbar. Diese Hypothese bedarf einer genaueren Untersuchung und wird im nächsten Abschnitt behandelt.

8.5. Einfluss der energetischen Unordnung auf die Transferkurven

Wie sich im Abschnitt zuvor gezeigt hat, bewirkt eine zusätzliche Verteilung der Energien der Oligomere eine Reduzierung der Stromdichte und der Mobilität im Vergleich zu isoenergetischen Oligomeren. Außerdem ist das Maximum der Stromdichte weiter zu höherer Gatespannung hin verschoben. Um diese Beobachtungen weiter zu untersuchen, habe ich Transferkurven und die Temperaturabhängigkeit der Ladungsträgermobilität eines Halbleiters ohne energetischer Unordnung, sowie von Halbleitern mit einer energetischen Unordnung von 25 und 50 meV untersucht. Für diese Untersuchung habe ich nun ausschließlich meine eigene Vielteilchensimulation Teil 2 verwendet (Kapitel 5.2.2). Dabei habe ich einen Teil der Daten aus dem Kapitel 5.2.2 übernommen, da ich dort bereits den Ladungstransport eines Halbleiters mit einer energetischen Unordnung von 50 meV simuliert hatte. Die Größe der Simulationsbox unterscheidet sich dort zu den Untersuchungen in den vorangegangenen Abschnitten, weswegen ich auch hier eine Größe der Simulationsbox von 100 x 50 x 3 verwendet habe. Dies hat jedoch keinen Einfluss auf die in diesem Abschnitt getroffenen Aussagen wie in Kapitel 8.2 gezeigt. Als Source-Drain-Spannung verwende ich die Werte 0.1 und 1 V. Da sich der Vorfaktor der Hüpfraten ν_0, den ich in Kapitel 5.2.2 verwendet habe, von dem in den vorhergegangenen Kapiteln 8.3 und 8.4 unterscheidet, habe ich aus Anschaulichkeitsgründen in diesem Abschnitt die Ergebnisse der Stromdichte und der Ladungsträgermobilität mit dem Faktor $2.203 \cdot 10^6$ multipliziert, was dem Verhältnis der Vorfaktoren in den vorangegangenen Kapiteln und dem in Kapitel 5.2.2 entspricht. Da es sich dabei lediglich um einen Vorfaktor handelt, hat dies keinerlei Auswirkung auf die in diesem Abschnitt getroffenen Aussagen.

8.5.1. Ergebnisse der Transferkurven bei verschiedener energetischer Unordnung und Temperatur

In Abbildung 8.15 sind die Stromdichte von Halbleitern mit einer energetischen Unordnung von 0, 25 und 50 meV als Funktion der Gatespannung einmal parametrisch in der Temperatur und einmal parametrisch in der energetischen Unordnung dargestellt. Die Daten für den Halbleiter mit einer energetischer Unordnung von 50 meV stammen dabei größtenteils aus Kapitel 5.2.2 und wurden um weitere Gatespannungen ergänzt.

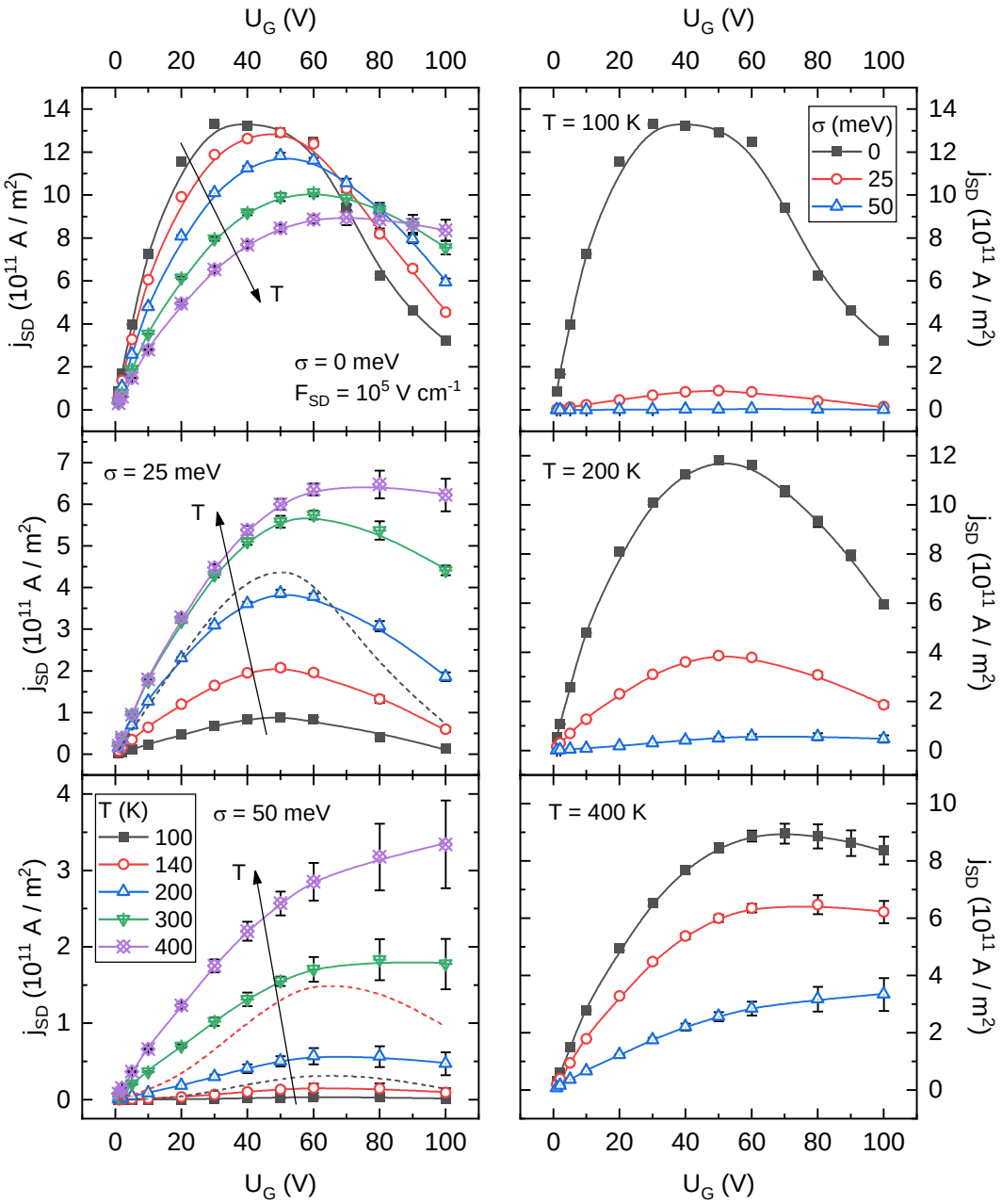

Abbildung 8.15 Linke Seite: Stromdichte als Funktion der Gatespannung bei verschiedenen Temperaturen und für verschiedene energetische Unordnung im Halbleiter wie angegeben. Die energetische Unordnung des Halbleiters beträgt von oben nach unten 0, 25 und 50 meV. Die gestrichelten Linien bei σ = 25 und 50 meV zeigen zur besseren Sichtbarkeit die Kurven von T = 100 skaliert mit einem Faktor 5 (σ = 25 meV) und von T = 100 und 140 K skaliert mit einem Faktor von 10 (σ = 50 meV). Rechte Seite: Dieselben Daten parametrisch in der energetischen Unordnung wie angegeben. Von oben nach unten beträgt die Temperatur 100, 200 und 400 K. Die Linien dienen lediglich der Veranschaulichung.

Beim Halbleiter ohne energetischer Unordnung (σ = 0) nimmt die Stromdichte bei Gatespannungen kleiner 60 V mit der Temperatur ab. Oberhalb von 60 V kehrt sich dieser Trend mit steigender Gatespannung zunehmend um, sodass bei U_G = 100 V die Stromdichte mit der Temperatur streng monoton zunimmt. Bei Halbleitern mit energetischer Unordnung nimmt die Stromdichte durchgehend bei allen Gatespannungen mit steigender Temperatur zu. Innerhalb des untersuchten Bereichs der Gatespannung nimmt die Stromdichte bei fast jeder Transferkurve ein lokales Maximum an. Die Position des Maximums hängt von der Temperatur ab. Bei σ = 0 liegt das Maximum bei T = 100 K bei U_G = 30 V und verschiebt sich zu U_G = 70 V bei T = 400 K, bei σ = 25 meV verschiebt es sich von U_G = 50 V nach U_G = 80 V mit steigender Temperatur und bei σ = 50 meV liegt es im Bereich von U_G = 60 V bis U_G = 80 V und nimmt bei T = 400 K im untersuchten Bereich kein lokales Maximum an. Insgesamt verschiebt sich somit das Maximum der Stromdichte mit steigender Temperatur und energetischer Unordnung zu höherer Gatespannung hin.

Um auf den Einfluss der Coulombwechselwirkung auf die Transferkurven bei unterschiedlicher energetischer Unordnung schließen zu können, habe ich in Abbildung 8.16 die Stromdichte aus Abbildung 8.15 in eine Ladungsträgermobilität gemäß Gleichung (3.5) umgerechnet. Als Darstellung habe ich eine doppeltlogarithmische Skalierung gewählt, um einen Abfall der Mobilität bei einer bestimmten Gatespannung besser darzustellen.

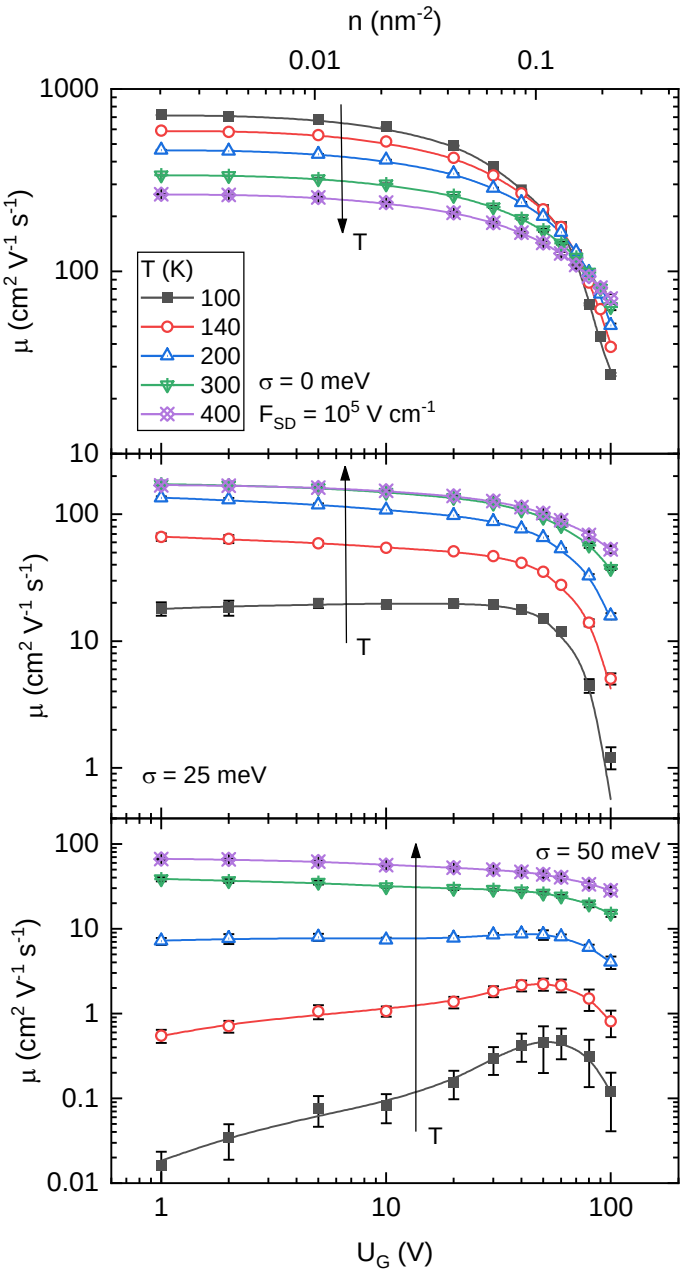

Abbildung 8.16 Ladungsträgermobilität als Funktion der Gatespannung in doppellogarithmischer Darstellung für verschiedene Temperaturen wie angegeben. Von oben nach unten ändert sich die energetische Unordnung σ von 0 nach 25 nach 50 meV. Die obere x-Achse zeigt die Flächenladungsdichte n (Gitterabstand 1 nm) unter der Annahme, dass sich alle Ladungsträger in der untersten Halbleiterschicht befinden. Dies kann insbesondere bei niedrigen Gatespannungen zu einer Abweichung von der tatsächlichen Ladungsträgerdichte führen aufgrund dessen, dass sich nicht alle Ladungsträger in der untersten Halbleiterschicht befinden. Die Linien dienen lediglich der Veranschaulichung.

Erkennbar ist, dass alle Kurven ab einer gewissen Gatespannung abzuknicken beginnen. Bei einer energetischen Unordnung von σ = 50 meV nimmt die Mobilität zudem ein Maximum bei

Temperaturen bis zu 200 K bei etwa U_G = 60 V an. Außerdem nimmt die Mobilität ohne energetischer Unordnung bei Gatespannung unterhalb von 60 V mit steigender Temperatur ab, bei den beiden anderen Fällen mit energetischer Unordnung nimmt sie zu.

Da die Stromdichte und Mobilität ohne energetische Unordnung mit steigender Temperatur bei Gatespannungen unterhalb von U_G = 60 V abnimmt und in den Fällen mit energetischer Unordnung zunimmt, lohnt es sich die Temperaturabhängigkeit des Ladungstransports genauer zu untersuchen. Dafür habe ich die Ladungsträgermobilität wie in Kapitel 5.2.2 beschrieben aus dem zurückgelegten Weg der Ladungsträger in Feldrichtung innerhalb der Simulationszeit berechnet. In Abbildung 8.17 ist sie als Funktion der Temperatur für Halbleiter mit verschiedener energetischer Unordnung aufgetragen. Um den Verlauf der Mobilität besser darzustellen, habe ich den Ladungstransport bei einigen Gatespannungen für zusätzliche Temperaturen simuliert. Die Daten für σ = 50 meV entstammen größtenteils denen aus Kapitel 5.2.2.

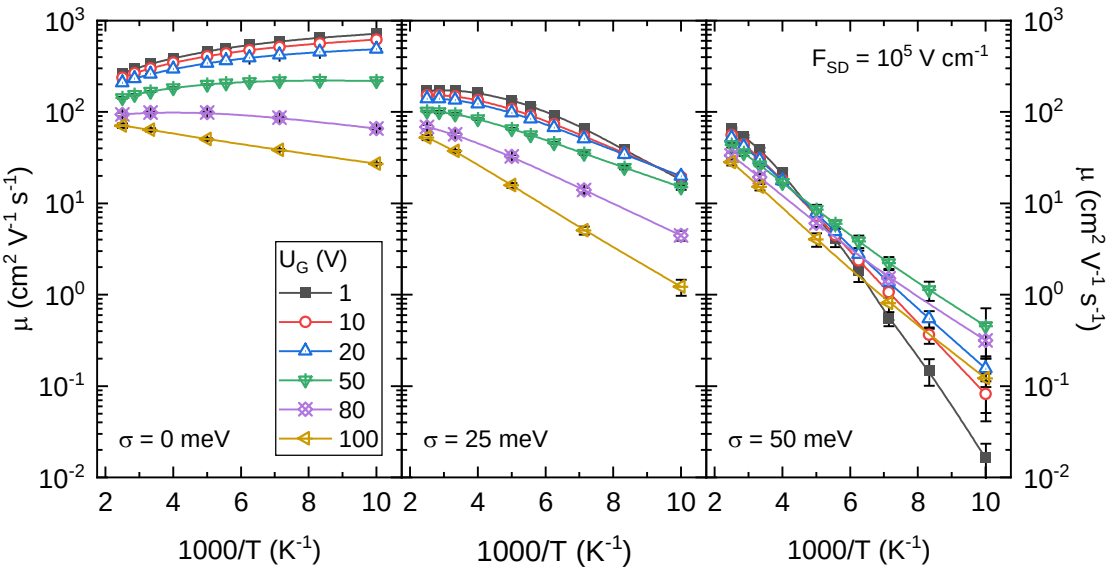

Abbildung 8.17 Ladungsträgermobilität als Funktion der Temperatur eines Halbleiters mit verschiedener energetischer Unordnung von σ = 0 bis σ = 50 meV für verschiedene Gatespannungen wie angegeben. Die Linien dienen lediglich der Veranschaulichung.

Erkennbar ist, dass bei einem Halbleiter ohne energetischer Unordnung die Ladungsträgermobilität bei Gatespannungen kleiner 50 V mit sinkender Temperatur zunimmt. Für größere Gatespannungen nimmt sie mit der Temperatur ab. Bei σ = 25 und 50 meV nimmt die Mobilität bei allen Gatespannungen mit sinkender Temperatur ab. Der Verlauf der Mobilität als Funktion der

Temperatur ist bei einer energetischen Unordnung von σ = 50 meV nahezu linear. Dabei nimmt die Steigung der Temperaturabhängigkeit mit steigender Gatespannung ab.

Da bei σ = 0 bei Gatespannungen unterhalb von 50 V die Mobilität mit sinkender Temperatur ansteigt, ist der Ladungstransport dort nicht thermisch aktiviert gemäß der Arrheniusgleichung:[42]

$$\mu \propto \mu' \exp\left(-\frac{\Delta}{k_B T}\right) \tag{8.5}$$

Im thermischen Gleichgewicht gilt für den funktionalen Zusammenhang zwischen Mobilität und Temperatur die Einstein-Beziehung:[42,127]

$$\mu = \frac{eD}{k_B T} \tag{8.6}$$

e ist die Elementarladung und D der Diffusionskoeffizient. Um die Temperaturabhängigkeit der Mobilität mit der Einstein-Beziehung zu vergleichen, habe ich den Diffusionskoeffizient als Funktion der Temperatur in Abbildung 8.18 aufgetragen. Um den Vergleich der Einstein-Beziehung mit der beobachteten Temperaturabhängigkeit der Mobilität bewerten zu können, habe ich zudem die Temperaturabhängigkeit der Mobilität bei einer niedrigeren Source-Drain-Feldstärke von F_{SD} = 10^4 V cm^{-1} simuliert und ebenfalls in Abbildung 8.18 dargestellt. Die Daten für die Source-Drain-Feldstärke F_{SD} = 10^5 V cm^{-1} entstammen aus Abbildung 8.17.

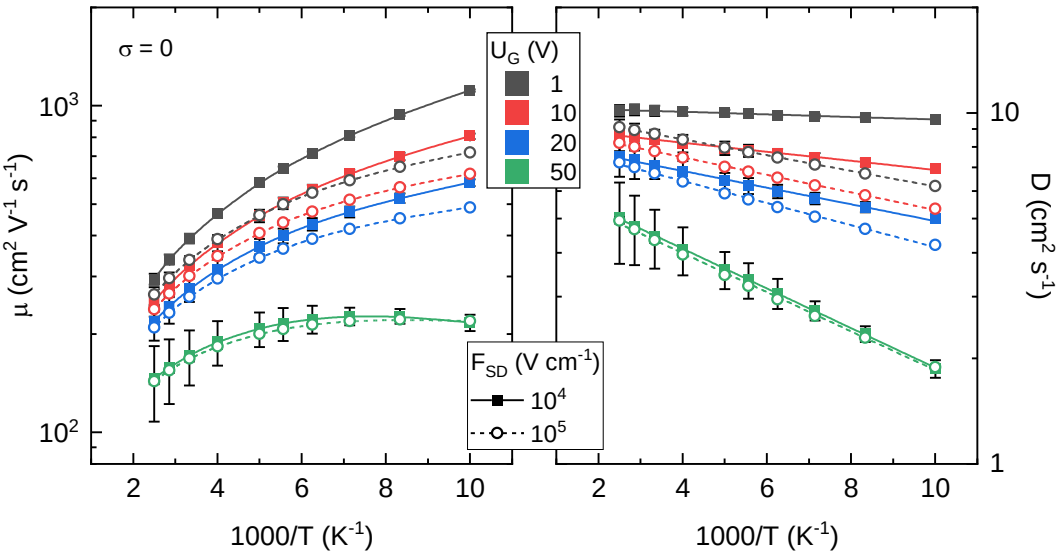

Abbildung 8.18 Links: Ladungsträgermobilität als Funktion der Temperatur eines einkristallinen Halbleiters für verschiedene Gatespannungen und Source-Drain-Feldstärken wie angegeben. Rechts: Diffusionskoeffizient als Funktion der Temperatur eines einkristallinen Halbleiters für dieselben Gatespannungen und Source-Drain-Feldstärken wie angegeben. Die Linien in der linken Abbildung dienen lediglich der Veranschaulichung, auf der rechten Seite stellen sie lineare Kurvenanpassungen dar.

Es zeigt sich, dass sich der Verlauf der Mobilität als Funktion der Temperatur zwischen den beiden Source-Drain-Feldstärken vor allem bei niedriger Gatespannung unterscheidet. Bei U_G = 50 V verlaufen die Kurven nahezu gleich. Der Diffusionskoeffizient verläuft annähernd konstant für eine Source-Drain-Feldstärke von F_{SD} = 10^4 V cm^{-1} bei einer Gatespannung von 1 V. Bei höherer Gatespannung und bei der Source-Drain-Feldstärke von F_{SD} = 10^5 V cm^{-1} bei allen Gatespannungen nimmt der Diffusionskoeffizient mit sinkender Temperatur ab. Insbesondere reduziert sich der Absolutwert des Diffusionskoeffizienten mit zunehmender Gatespannung. Das ist eine deutliche Manifestation des Effektes der Coulombwechselwirkung.

Für eine genauere Quantifizierung der Temperaturabhängigkeit habe ich in halblogarithmischer Darstellung eine lineare Kurvenanpassung für die Diffusionskoeffizienten durchgeführt, um die Steigung und daraus die Aktivierungsenergie E_A bestimmen zu können gemäß:

$$D = D_\infty \exp\left(-\frac{E_A}{k_B T}\right) \tag{8.7}$$

wobei D_∞ der Diffusionskoeffizient bei unendlich hoher Temperatur ist. Die sich daraus ergebende Aktivierungsenergie ist in Abbildung 8.19 dargestellt.

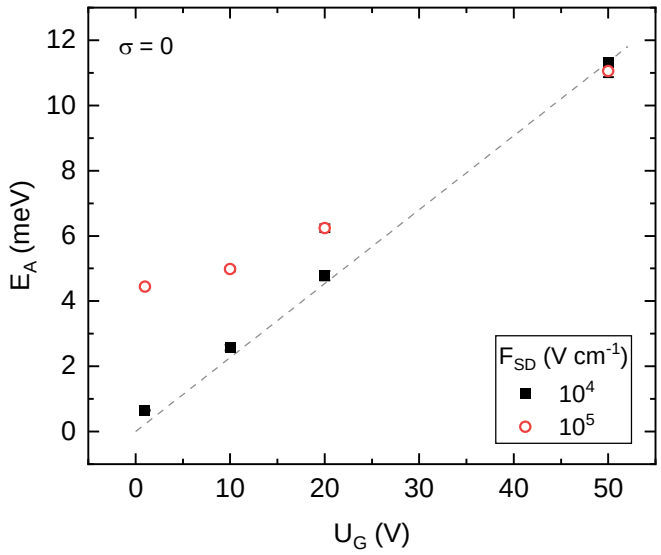

Abbildung 8.19 Aktivierungsenergie des Diffusionskoeffizienten als Funktion der Gatespannung bei verschiedenen Source-Drain-Feldstärken wie angegeben. Die gestrichelte Linie stellt eine Ursprungsgerade dar.

Die gestrichelte Linie in Abbildung 8.19 stellt eine Ursprungsgerade dar, die durch den Datenpunkt bei U_G = 50 V geht bei einer Source-Drain-Feldstärke von F_{SD} = 10^4 V cm^{-1}. Es ist erkennbar, dass für die Source-Drain-Feldstärke F_{SD} = 10^4 V cm^{-1} die Datenpunkte annähernd auf dieser Ursprungsgeraden liegen. Bei der Source-Drain-Feldstärke F_{SD} = 10^5 V cm^{-1} weicht die Aktivierungsenergie bei niedrigen Gatespannungen von der Ursprungsgeraden ab.

8.5.2. Diskussion der Ergebnisse der Transferkurven bei verschiedener energetischer Unordnung

In organischen Halbleitern mit energetischer Unordnung (σ > 0) ist der Ladungstransport ein temperaturaktivierter Prozess.[17,47] Der Stromfluss setzt sich dabei aus einem diffusiven Anteil, und einem Driftanteil zusammen.[17] Beide Anteile überlagern sich, wobei der diffusive Anteil isotrop und der Driftanteil entlang des Potentialgradienten gerichtet ist. Bei einem einkristallinen Halbleiter ohne energetischer Unordnung habe ich in Abbildung 8.15 beobachtet, dass bei Gatespannungen kleiner 60 V die Stromdichte mit steigender Temperatur abnimmt. In Systemen ohne energetischer Unordnung gibt es zunächst keine Sprünge energetisch aufwärts. Da jedoch eine Spannung in Source-Drain-Richtung anliegt, entsteht durch das elektrische Feld in

dieser Richtung ein Gradient in den Energien der Gitterplätze, entlang dessen sich die Ladungsträger bevorzugt bewegen. Sprünge der Ladungsträger in Feldrichtung sind somit immer energetisch abwärts, wodurch in der Miller-Abrahams-Hüpfrate (Gleichung (5.7)) ein Faktor von 1 verwendet wird. Bei den Sprüngen entgegen der Feldrichtung muss immer die Potentialdifferenz $\Delta\varphi_x$ zwischen zwei benachbarten Gitterpunkten in x-Richtung aufgrund des Source-Drain-Feldes überwunden werden, weswegen Sprünge entgegen der Feldrichtung immer mit dem Faktor $\exp(-\Delta\varphi_x/k_BT)$ in der Miller-Abrahams-Hüpfrate multipliziert werden. Bei höherer Temperatur ist dieser Faktor näher an eins, sodass Sprünge entgegen der Source-Drain-Feldrichtung wahrscheinlicher sind. Dies steht jedoch dem Stromfluss in Source-Drain-Richtung entgegen, wodurch sich eine niedrigere Stromdichte ergibt. Für die untersuchten Fälle mit energetischer Unordnung nimmt die Stromdichte mit steigender Temperatur zu, da Sprünge in Source-Drain-Richtung temperaturaktiviert sind und die Rate somit mit steigender Temperatur steigt. Selbstverständlich treten auch Sprüngen entgegen der Feldrichtung auf, die mit steigender Temperatur wahrscheinlicher werden und somit die Stromdichte verringern würden. Dieser Effekt verschwindet jedoch gegenüber der thermisch aktivierten Sprüngen bei einer energetischen Unordnung von σ = 25 und 50 meV.

Ein weiteres Phänomen ist das Auftreten eines Maximums in der Stromdichte als Funktion der Gatespannung (siehe Abbildung 8.15). Dies lässt sich damit begründen, dass mit zunehmender Gatespannung die Coulombwechselwirkung zwischen den Ladungsträgern zunimmt, wie in Kapitel 8.3.2 erläutert. Dass sich das Maximum zu höherer Gatespannung mit zunehmender energetischer Unordnung verschiebt, ist plausibel, da der abschwächende Effekt der energetische Unordnung auf den Ladungstransport mit zunehmender Anzahl an Ladungsträgern geringer wird aufgrund des Auffüllens der Zustandsdichte. Dies steht der Coulombwechselwirkung entgegen, weswegen eine höhere Gatespannung für das Maximum der Stromdichte notwendig ist. Eine höhere Temperatur bewirkt ebenfalls, dass der Effekt durch die Coulombwechselwirkung abgeschwächt wird und eine höhere Gatespannung für das Maximum notwendig ist. Der Effekt durch die Coulombwechselwirkung zeigt sich ebenfalls im Auftreten des Maximums in der Mobilität. Ebenso gilt auch das gleiche Argument wie bei der Stromdichte für die Abnahme der Mobilität mit der Temperatur ohne energetischer Unordnung, dass Sprünge entgegen der Feldrichtung durch die Temperatur begünstigt werden, was den Ladungstransport verschlechtert. Bei σ = 50 meV kann bei tiefer Temperatur in der Mobilität der konkurrierende Effekt zwischen dem Auffüllen der Zustandsdichte und der Coulombwechselwirkung mit steigender Gatespannung beobachtet werden (siehe Abbildung 8.16). Dies zeigt sich daran, dass bei Temperaturen

bis 200 K ein Maximum in der Mobilität bei etwa U_G = 60 V beobachtet wird. Bei höherer Temperatur oder kleinerer energetischer Unordnung wird dies nicht beobachtet, da dort das Auffüllen der Zustandsdichte keine ausreichend große Verbesserung des Ladungstransports hervorruft. Als Argument kann folgendes einfaches Rechenbeispiel dienen: Der Füllgrad der Zustandsdichte beträgt gemäß Gleichung (3.6) bei einer Gatespannung von 10 V 2.2 % und bei U_G = 60 V 13.26 % unter der Annahme, dass sich alle Ladungsträger in der untersten Halbleiterschicht befinden. Um das Ferminiveau zu berechnen, nehme ich eine einfache Gaußverteilung für die Zustandsdichte unter der Vernachlässigung der Aufweichung der Fermikante durch die Temperatur an. Das Ferminiveau berechnet sich dann über eine numerische Nullstellensuche der Gleichung:

$$\eta = \int_{-\infty}^{\varepsilon_F} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) \mathrm{d}x \tag{8.8}$$

wobei η der Füllgrad der Zustandsdichte und ε_F das Ferminiveau ist. Daraus ergibt sich ein Ferminiveau von etwa -2σ bei einer Gatespannung von 10 V und -1.1σ bei einer Gatespannung von 60 V. Das Ferminiveau ist somit abhängig von der Breite der Verteilung der Energien. Dadurch bewirkt eine höhere energetische Unordnung eine größere absolute Änderung des Ferminiveaus bei gleicher Änderung der Gatespannung, weswegen der Effekt auf den Ladungstransport dort stärker sichtbar ist. Das Auffüllen der Zustandsdichte geschieht gemäß der Fermi-Dirac-Statistik, wobei die Fermi-Kante mit steigender Temperatur aufweicht.[48] Dies schwächt den Effekt des Auffüllens der Zustandsdichte ab, wodurch bei höherer Temperatur kein Maximum in der Mobilität beobachtet wird.

Die Darstellung der Mobilität als Funktion der Temperatur in Abbildung 8.17 hat ebenso wie in Abbildung 8.16 gezeigt, dass die Mobilität ohne energetischer Unordnung mit sinkender Temperatur bei kleiner Gatespannung zunimmt und bei Gatespannung größer als 50 V abnimmt. Bei einer Gatespannung größer 50 V ist der Ladungstransport temperaturaktiviert, da sich ein Verlauf gemäß der Arrheniusgleichung ergibt, wobei die thermische Energie der Coulombwechselwirkung entgegen steht. Bei höherer energetischer Unordnung, insbesondere bei σ = 50 meV, ist der Temperaturverlauf der Mobilität bei allen Gatespannungen annähernd gemäß der Arrheniusgleichung, was dem erwarteten Ladungstransport eines Hüpfprozesses in organischen Halbleitern mit hoher Ladungsträgerdichte entspricht.[42] Die Abnahme der Steigung der Temperaturabhängigkeit mit steigender Gatespannung bei σ = 50 meV ist plausibel, da die Zustandsdichte weiter aufgefüllt wird und somit die Aktivierungsenergie sinkt.[42]

Um die Zunahme der Mobilität mit sinkender Temperatur ohne energetischer Unordnung bei niedriger Gatespannung auf den funktionalen Zusammenhang zu überprüfen, eignet sich eine doppellogarithmische Darstellung. In Abbildung 8.20 habe ich den Temperaturverlauf der Mobilität bei einer Gatespannung von 1 V für die untersuchten Source-Drain-Feldstärken F_{SD} = 10^4 und 10^5 V cm^{-1} dargestellt. Mittels einer linearen Kurvenanpassung habe ich die Steigung der Verläufe der Mobilität ermittelt, die somit den Exponenten einer Potenzfunktion angeben.

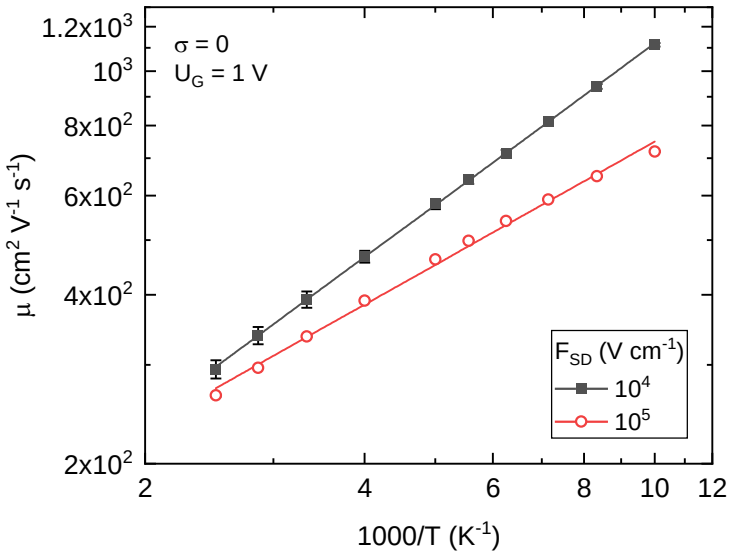

Abbildung 8.20 Ladungsträgermobilität als Funktion der Temperatur in doppeltlogarithmischer Darstellung eines einkristallinen Halbleiters bei einer Gatespannung von U_G = 1 V bei zwei verschiedenen Source-Drain-Feldstärken wie angegeben. Die Linien stellen lineare Kurvenanpassungen dar.

Für die Source-Drain-Feldstärke 10^4 V cm^{-1} erhalte ich als Steigung 0.960 ± 0.003 und für F_{SD} = 10^5 V cm^{-1} als Steigung 0.730 ± 0.017. Der sich ergebende funktionale Zusammenhang zwischen Mobilität und Temperatur entspricht somit

$$\mu \propto T^{-\alpha} \tag{8.9}$$

wobei α die Steigung der Geraden ist. Für die Source-Drain-Feldstärke 10^4 V cm^{-1} entspricht dies annähernd der Einstein-Beziehung nach Gleichung (8.6). Dies stimmt überein mit dem annähernd konstanten Verlauf des Diffusionskoeffizienten bei dieser Source-Drain-Feldstärke bei U_G = 1 V. Die Abweichung von der Einstein-Beziehung bei höherer Source-Drain-Feldstärke kann dadurch zustande kommen, dass bei höherem Source-Drain-Feld der Ladungstransport stärker

durch die Drift der Ladungsträger in Feldrichtung bestimmt wird als durch Diffusion. Die Einstein-Beziehung gilt jedoch nur für den Fall des diffusionsgetriebenen Ladungstransports im thermischen Gleichgewicht.[151] Bei höherer Gatespannung weicht der Diffusionskoeffizient stärker von einem konstanten Verlauf ab, da Coulombeffekte stärker werden, die die Mobilität reduzieren. Somit ist der konstante Verlauf des Diffusionskoeffizienten nur bei niedriger Gatespannung und niedriger Source-Drain-Feldstärke gegeben.

Der Ladungstransport setzt sich aus einem diffusiven und einem Driftanteil zusammen.[17] Bei niedriger Source-Drain-Feldstärke tritt der diffusive Anteil des Ladungstransports stärker in den Vordergrund. In Abbildung 8.19 kann deswegen bei der Source-Drain-Feldstärke F_{SD} = 10^4 V cm^{-1} der Verlauf der Aktivierungsenergie vor allem dem diffusiven Anteil zugeschrieben werden. Für einen Halbleiter ohne energetische Unordnung wird eine Aktivierungsenergie von null erwartet, da es keine thermische Aktivierung des Ladungstransports gibt. Jedoch bewegen sich die Ladungsträger im gegenseitigen Coulombpotential, gegen das die Ladungsträger thermisch aktiviert werden müssen. Die Aktivierungsenergie gibt somit die Stärke der Coulombabstoßung bei einer bestimmten Gatespannung wieder. Diese Art der Bestimmung der Coulombwechselwirkung zwischen den Ladungsträgern wurde nach meinem besten Wissen noch nicht angewandt. Die Aktivierungsenergie des Diffusionskoeffizienten bei U_G = 50 V liegt bei etwa 11 meV, was deutlich kleiner ist als die mittlere Abstoßung zwischen zwei Ladungsträgern von etwa 173 meV, wenn sich die Ladungsträger gleichmäßig in der untersten Halbleiterschicht verteilen (siehe Abbildung 8.12). Dies liegt daran, dass sich die Ladungsträger als ein Ensemble bewegen und sich nicht gegen das starre Coulombpotential eines einzelnen Ladungsträgers bewegen müssen.

Bei σ = 0 kehrt sich das Vorzeichen der Steigung der Temperaturabhängigkeit mit zunehmender Gatespannung um. Um den Einfluss der Coulombwechselwirkung dabei und bei Halbleitern mit energetischer Unordnung besser quantifizieren zu können, habe ich in Abbildung 8.21 die Mobilität auf den Wert der Mobilität bei der Gatespannung 1 V für jede energetische Unordnung normiert.

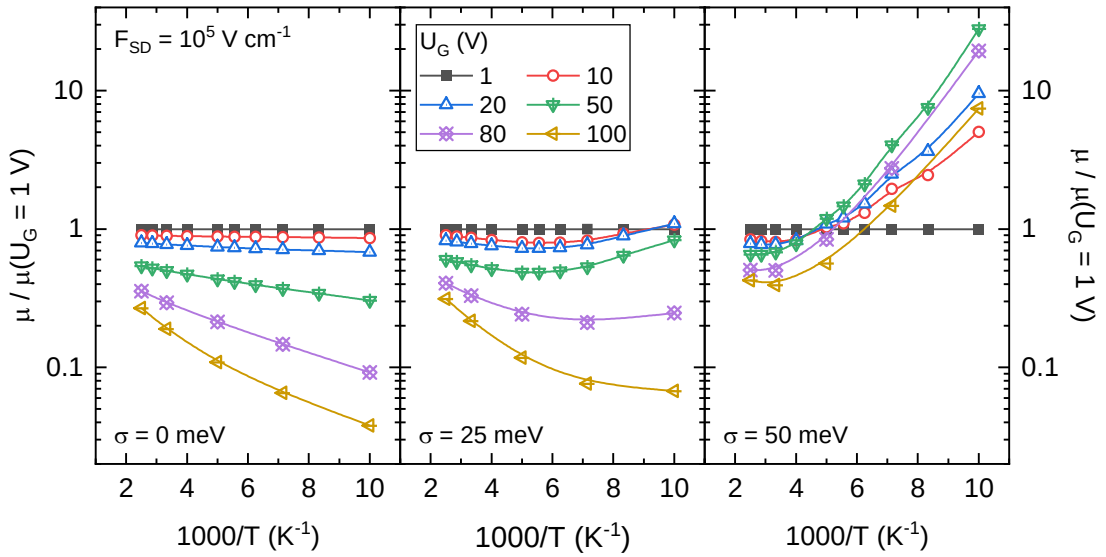

Abbildung 8.21 Normierte Ladungsträgermobilität als Funktion der Temperatur für verschiedene Gatespannungen wie angegeben. Von links nach rechts: Die energetische Unordnung des Halbleiters ändert sich von σ = 0 auf 50 meV wie angegeben. Die Mobilität wurde für jede energetische Unordnung auf den Wert der Gatespannung 1 V bei jeder Temperatur normiert. Die Linien dienen lediglich der Veranschaulichung.

Es zeigt sich, dass bei σ = 0 die Mobilität bei einer Gatespannung von 10 V um etwa 10 % abnimmt im Vergleich zur Gatespannung 1 V und für höhere Gatespannungen noch kleiner wird. Da bei einer Gatespannung von 1 V die Mobilität durch die Einstein-Beziehung gegeben ist, zeigt diese Darstellung damit den reinen Coulombeffekt auf die Mobilität. Wie in Kapitel 8.3.2 bereits erläutert, ist ab Gatespannungen größer 10 V mit einem deutlicher sichtbaren Coulombeffekt zu rechnen aufgrund der Größenverhältnisse zwischen Coulombwechselwirkung zwischen zwei Ladungsträgern und der Potentialdifferenz zwischen zwei Halbleiterschichten. Die Beobachtung hier stimmt mit dieser Aussage überein. Analog gilt bei Halbleitern mit energetischer Unordnung σ > 0, dass der Verlauf der normierten Mobilität bei höherer Gatespannung den Einfluss der Coulombwechselwirkung zeigt. Dabei zeigt sich, dass insbesondere bei tiefen Temperaturen die erhöhte Ladungsträgerdichte einen positiven Effekt auf die Ladungsträgermobilität hat, indem bei σ = 50 meV beispielsweise bei einer Gatespannung von 50 V für Temperaturen unterhalb von 250 K die normierte Mobilität einen Wert größer als 1 annimmt. Dies kann damit erklärt werden, dass hier ein Auffüllen der Zustandsdichte durch die Ladungsträger mit steigender Gatespannung vorliegt, deren positiver Effekt auf die Mobilität in diesem Fall stärker ist als die

abschwächende Wirkung durch die Coulombwechselwirkung. Ein Grund dafür, warum die normierte Mobilität der Gatespannung 100 V bei σ = 50 meV den Wert 1 bei niedrigerer Temperatur übersteigt als bei den übrigen Gatespannungen kann darin liegen, dass hierbei der Effekt der Coulombwechselwirkung stärker ist im Vergleich zum Effekt durch das Auffüllen der Zustandsdichte. Insgesamt kann somit geschlussfolgert werden, dass Coulombeffekte bei niedriger energetischer Unordnung am deutlichsten sichtbar sind, da sie nicht durch andere Effekte überlagert werden.

8.6. Zusammenfassung

In diesem Kapitel konnte ich zeigen, dass die Coulombwechselwirkung zwischen den Ladungsträgern die Transfercharakteristik von OFETs beeinflussen kann. Dabei entsteht eine wie in Abbildung 8.1c gezeigte gekrümmte Transfercharakteristik, die sogar ein Maximum aufweisen kann. Um die Coulombwechselwirkung dabei korrekt zu erfassen, muss das Coulombpotential weiträumig mit einem Cut-off-Radius von mindestens 40 nm berücksichtigt werden, was im Fachgebiet meines Wissens nach so noch nicht publiziert wurde. Der Effekt der Coulombwechselwirkung auf die Transfercharakteristik wird durch eine zusätzliche energetische Unordnung im Halbleiter und eine hohe Temperatur abgemildert. Dabei kann das Auffüllen der Zustandsdichte bei großer energetischer Unordnung und niedriger Temperatur den Effekt durch die Coulombwechselwirkung teilweise kompensieren. Der Effekt durch die Coulombwechselwirkung macht sich bei Raumtemperatur im von mir untersuchten Bereich der energetischen Unordnung immer bemerkbar. Weiterhin konnte ich zeigen, dass es in Bezug auf die Form der Transfercharakteristik keinen Unterschied macht, ob die ausgerichteten Oligomere unendlich oder endlich lang sind.

9. Schlussfolgerung

In dieser Arbeit habe ich die Frage untersucht, welchen Einfluss die Morphologie des Halbleiters und die Coulombwechselwirkung auf den Ladungstransport, insbesondere die Mobilität, in OFETs hat. Dafür habe ich mittels kinetischer Monte-Carlo-Simulation den Einfluss von Korngrenzen, der Anordnung von Oligomeren zueinander, sowie der Coulombwechselwirkung auf den Ladungstransport und die Transfercharakteristiken untersucht. Die von mir entwickelten Methoden, sowie die von mir gewonnenen Erkenntnisse zum zugrundeliegenden Ladungstransportmechanismus fasse ich im Folgenden kurz zusammen.

Zur Untersuchung des Ladungstransports in OFETs habe ich eine Methode zur Berücksichtigung der Coulombwechselwirkung in der Simulation entwickelt, welche in Kapitel 5 beschrieben ist. Diese liefert akkurate Ergebnisse, benötigt jedoch mehr Rechenzeit als die kommerzielle Software Bumblebee. Um jedoch mit der Software Bumblebee mit meiner Methode übereinstimmende Ergebnisse zu erhalten, ist ein Cut-off-Radius von mindestens 40 nm erforderlich.

In polykristallinen OFETs treten Korngrenzen auf. Ich habe in meinen Untersuchungen in Kapitel 6 zum Ladungstransport in polykristallinen OFETs drei verschiedene Bereiche des Ladungstransports identifiziert, je nach der energetischen Lage der Korngrenzen im Vergleich zu den Kristalliten. Für energetisch tiefe Korngrenzen findet der Ladungstransport ausschließlich in den Korngrenzen statt. Bei energetisch flachen Korngrenzen findet der Ladungstransport zunehmend in den Kristalliten statt und die Korngrenzen wirken als Fallen für die Ladungsträger. Dabei spielt die Position des Ferminiveaus eine wichtige Rolle, da die Position des Ferminiveaus mit abnehmender Breite der Korngrenzen zunimmt, was das Entkommen der Ladungsträger aus den Fallenzuständen begünstigt. Handelt es sich bei den Korngrenzen um energetische Barrieren, so werden diese von den Ladungsträgern mittels Tunneln oder thermischer Aktivierung überwunden. Die wesentliche neue Erkenntnis aus diesem Teil der Arbeit ist, dass für eine Beschreibung des Ladungstransports in organischen Halbleitern mit Korngrenzen das Ferminiveau berücksichtigt werden muss, was in die Veröffentlichung Meier et al. in Advanced Optical Materials eingegangen ist.[108]

Als weiteren morphologischen Aspekt habe ich den Einfluss der Kettenlänge und des Versatzes benachbarter Oligomere in OFETs auf den Ladungstransport untersucht (Kapitel 7). Dabei habe ich gezeigt, dass das Anwachsen der Ladungsträgermobilität mit steigender Kettenlänge mit dem funktionalen Zusammenhang für einen eindimensionalen Oligomerstrang übereinstimmt.

Liegt zwischen den Oligomeren jedoch eine energetische Unordnung vor, so nimmt die Abhängigkeit der Ladungsträgermobilität von der Kettenlänge einen deutlich linearen Verlauf an. Außerdem hat sich gezeigt, dass sich ein Versatz benachbarter Oligomere grundsätzlich positiv auf den Ladungstransport auswirkt. Die Erhöhung der Mobilität ist dabei größer, wenn der Versatz benachbarter Oligomere in der Substratebene des OFETs vorliegt, als wenn er nur in der Richtung senkrecht zur Substratebene vorliegt. Das beobachtete Maximum der Stromdichte und Mobilität bei einem Versatz benachbarter Oligomere, der genau der Hälfte der Kettenlänge entspricht, stimmt mit der aufgestellten theoretischen Vorhersage überein. Wird der Versatz benachbarter Oligomere zufällig gewählt, so ergeben sich nahezu keine Unterschiede im Vergleich zu einem Versatz benachbarter Oligomere um genau die Hälfte der Kettenlänge. Dies ist eine bedeutende Aussage für die notwendige Sorgfalt, die bei der Herstellung von oligomeren Halbleiterschichten erforderlich ist.

Schließlich habe ich den expliziten Effekt der Coulombwechselwirkung auf den Ladungstransport in OFETs untersucht (Kapitel 8). Dazu habe ich verschiedene Methoden zur Berücksichtigung der Coulombwechselwirkung miteinander verglichen und konnte daraus zeigen, dass die Coulombwechselwirkung zu einem Maximum in der Transfercharakteristik führt. Wichtig für die Berechnung der Coulombwechselwirkung ist dabei eine weiträumige Berücksichtigung mit einem Cut-off-Radius von mindestens 40 nm. Liegt zusätzlich eine energetische Unordnung oder eine hohe Temperatur im Halbleiter vor, so kann dies den Effekt durch die Coulombwechselwirkung abschwächen. Bei niedriger Temperatur und hoher energetischer Unordnung, z.B. bei σ = 50 meV und T = 100 K, kann das Auffüllen der Zustandsdichte den Effekt der Coulombwechselwirkung teilweise kompensieren. Außerdem habe ich gezeigt, dass eine Morphologie mit endlich oder unendlich langen Oligomeren keinen Einfluss auf die Transfercharakteristik hat.

Zusammenfassend lässt sich sagen, dass Coulombeffekte den Ladungstransport in OFETs in erheblichem Maße beeinflussen können, was sich insbesondere in den Transferkurven zeigt. Die Größe und energetische Lage von Korngrenzen in polykristallinen OFETs, sowie die Kettenlänge und der Versatz benachbarter Oligomere und deren energetische Unordnung in oligomeren OFETs bestimmen ebenfalls den Ladungstransport. Dies hat Einfluss darauf, welche Morphologie des Halbleiterfilms für bestmögliche Ladungstransporteigenschaften anzustreben ist.

10. Literaturverzeichnis

1 R. G. Kepler, Charge Carrier Production and Mobility in Anthracene Crystals, *Phys. Rev.*, 1960, **119**, 1226–1229.

2 O. H. LeBlanc, Hole and Electron Drift Mobilities in Anthracene, *J. Chem. Phys.*, 1960, **33**, 626–626.

3 Z. A. Lamport, H. F. Haneef, S. Anand, M. Waldrip and O. D. Jurchescu, Tutorial: Organic field-effect transistors: Materials, structure and operation, *J. Appl. Phys.*, 2018, **124**, 071101.

4 H. Sirringhaus, T. Sakanoue and J.-F. Chang, Charge-transport physics of high-mobility molecular semiconductors, *Phys. Status Solidi B*, 2012, **249**, 1655–1676.

5 S. Fratini, M. Nikolka, A. Salleo, G. Schweicher and H. Sirringhaus, Charge transport in high-mobility conjugated polymers and molecular semiconductors, *Nat. Mater.*, 2020, **19**, 491–502.

6 A. C. Arias, J. D. MacKenzie, I. McCulloch, J. Rivnay and A. Salleo, Materials and Applications for Large Area Electronics: Solution-Based Approaches, *Chem. Rev.*, 2010, **110**, 3–24.

7 J. Rivnay, R. M. Owens and G. G. Malliaras, The Rise of Organic Bioelectronics, *Chem. Mater.*, 2014, **26**, 679–685.

8 A. Tsumura, H. Koezuka and T. Ando, Macromolecular electronic device: Field-effect transistor with a polythiophene thin film, *Appl. Phys. Lett.*, 1986, **49**, 1210–1212.

9 H. Un, J. Wang and J. Pei, Recent Efforts in Understanding and Improving the Nonideal Behaviors of Organic Field-Effect Transistors, *Adv. Sci.*, 2019, **6**, 1900375.

10 L. Luo and Z. Liu, Recent progress in organic field-effect transistor-based chem/bio-sensors, *VIEW*, 2022, **3**, 20200115.

11 Y. Wang, Q. Gong and Q. Miao, Structured and functionalized organic semiconductors for chemical and biological sensors based on organic field effect transistors, *Mater. Chem. Front.*, 2020, **4**, 3505–3520.

12 J. T. Friedlein, R. R. McLeod and J. Rivnay, Device physics of organic electrochemical transistors, *Org. Electron.*, 2018, **63**, 398–414.

13 E. Macchia, K. Manoli, B. Holzer, C. Di Franco, M. Ghittorelli, F. Torricelli, D. Alberga, G. F. Mangiatordi, G. Palazzo, G. Scamarcio and L. Torsi, Single-molecule detection with a millimetre-sized transistor, *Nat. Commun.*, 2018, **9**, 3223.

14 G. Gelinck, P. Heremans, K. Nomoto and T. D. Anthopoulos, Organic Transistors in Optical Displays and Microelectronic Applications, *Adv. Mater.*, 2010, **22**, 3778–3798.

15 Samsung Galaxy Fold 5G | Samsung Deutschland, https://www.samsung.com/de/smartphones/galaxy-fold/, (accessed November 22, 2022).

16 HUAWEI Mate Xs 2 - HUAWEI UK, https://consumer.huawei.com/uk/phones/mate-xs2/, (accessed November 22, 2022).

17 A. Köhler and H. Bässler, *Electronic Processes in Organic Semiconductors: An Introduction*, Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany, 2015.

18 S. K. Park, T. N. Jackson, J. E. Anthony and D. A. Mourey, High mobility solution processed 6,13-bis(triisopropyl-silylethynyl) pentacene organic thin film transistors, *Appl. Phys. Lett.*, 2007, **91**, 063514.

19 O. D. Jurchescu, S. Subramanian, R. J. Kline, S. D. Hudson, J. E. Anthony, T. N. Jackson and D. J. Gundlach, Organic Single-Crystal Field-Effect Transistors of a Soluble Anthradithiophene, *Chem. Mater.*, 2008, **20**, 6733–6737.

20 A. Y. Amin, A. Khassanov, K. Reuter, T. Meyer-Friedrichsen and M. Halik, Low-Voltage Organic Field Effect Transistors with a 2-Tridecyl[1]benzothieno[3,2- *b*][1]benzothiophene Semiconductor Layer, *J. Am. Chem. Soc.*, 2012, **134**, 16548–16550.

21 H. Minemawari, T. Yamada, H. Matsui, J. Tsutsumi, S. Haas, R. Chiba, R. Kumai and T. Hasegawa, Inkjet printing of single-crystal films, *Nature*, 2011, **475**, 364–367.

22 K. Nakayama, Y. Hirose, J. Soeda, M. Yoshizumi, T. Uemura, M. Uno, W. Li, M. J. Kang, M. Yamagishi, Y. Okada, E. Miyazaki, Y. Nakazawa, A. Nakao, K. Takimiya and J. Takeya, Patternable Solution-Crystallized Organic Transistors with High Charge Carrier Mobility, *Adv. Mater.*, 2011, **23**, 1626–1629.

23 I. N. Hulea, S. Fratini, H. Xie, C. L. Mulder, N. N. Iossad, G. Rastelli, S. Ciuchi and A. F. Morpurgo, Tunable Fröhlich polarons in organic single-crystal transistors, *Nature Mater.*, 2006, **5**, 982–986.

24 O. D. Jurchescu, M. Popinciuc, B. J. van Wees and T. T. M. Palstra, Interface-Controlled, High-Mobility Organic Transistors, *Adv. Mater.*, 2007, **19**, 688–692.

25 C. Luo, A. K. K. Kyaw, L. A. Perez, S. Patel, M. Wang, B. Grimm, G. C. Bazan, E. J. Kramer and A. J. Heeger, General Strategy for Self-Assembly of Highly Oriented Nanocrystalline Semiconducting Polymers with High Mobility, *Nano Lett.*, 2014, **14**, 2764–2771.

26 N. D. Arora, J. R. Hauser and D. J. Roulston, Electron and hole mobilities in silicon as a function of concentration and temperature, *IEEE Trans. Electron Devices*, 1982, **29**, 292–295.

27 J. Mei, Y. Diao, A. L. Appleton, L. Fang and Z. Bao, Integrated Materials Design of Organic Semiconductors for Field-Effect Transistors, *J. Am. Chem. Soc.*, 2013, **135**, 6724–6746.

28 O. D. Jurchescu, D. A. Mourey, Y. Li, D. J. Gundlach and T. N. Jackson, in *Organic Electronics II*, ed. H. Klauk, Wiley, 1st edn., 2012, pp. 327–352.

29 H. H. Choi, A. F. Paterson, M. A. Fusella, J. Panidi, O. Solomeshch, N. Tessler, M. Heeney, K. Cho, T. D. Anthopoulos, B. P. Rand and V. Podzorov, Hall Effect in Polycrystalline Organic Semiconductors: The Effect of Grain Boundaries, *Adv. Funct. Mater.*, 2020, **30**, 1903617.

30 W. L. Kalb, S. Haas, C. Krellner, T. Mathis and B. Batlogg, Trap density of states in small-molecule organic semiconductors: A quantitative comparison of thin-film transistors with single crystals, *Phys. Rev. B*, 2010, **81**, 155315.

31 G. Horowitz and M. E. Hajlaoui, Mobility in Polycrystalline Oligothiophene Field-Effect Transistors Dependent on Grain Size, *Adv. Mater.*, 2000, **12**, 5.

32 A. Di Carlo, F. Piacenza, A. Bolognesi, B. Stadlober and H. Maresch, Influence of grain sizes on the mobility of organic thin-film transistors, *Appl. Phys. Lett.*, 2005, **86**, 263501.

33 A. B. Chwang and C. D. Frisbie, Temperature and gate voltage dependent transport across a single organic semiconductor grain boundary, *J. Appl. Phys.*, 2001, **90**, 1342–1349.

34 J. Rivnay, L. H. Jimison, J. E. Northrup, M. F. Toney, R. Noriega, S. Lu, T. J. Marks, A. Facchetti and A. Salleo, Large modulation of carrier transport by grain-boundary molecular packing and microstructure in organic thin films, *Nature Mater.*, 2009, **8**, 952–958.

35 G. Schweicher, G. Garbay, R. Jouclas, F. Vibert, F. Devaux and Y. H. Geerts, Molecular Semiconductors for Logic Operations: Dead-End or Bright Future?, *Adv. Mater.*, 2020, **32**, 1905909.

36 T. Afzal, M. J. Iqbal, M. Z. Iqbal, A. Sajjad, M. A. Raza, S. Riaz, M. A. Kamran, A. Numan and S. Naseem, Effect of post-deposition annealing temperature on the charge carrier mobility and morphology of DPPDTT based organic field effect transistors, *Chem. Phys. Lett.*, 2020, **750**, 137507.

37 S. R. Mohan, M. P. Singh and M. P. Joshi, A model for charge transport in semicrystalline polymer thin films, *J. Polym. Sci. B Polym. Phys.*, 2019, **57**, 137–141.

38 I. Vladimirov, M. Kühn, T. Geßner, F. May and R. T. Weitz, Energy barriers at grain boundaries dominate charge carrier transport in an electron-conductive organic semiconductor, *Sci. Rep.*, 2018, **8**, 14868.

39 I. Vladimirov, M. Kellermeier, T. Geßner, Z. Molla, S. Grigorian, U. Pietsch, L. S. Schaffroth, M. Kühn, F. May and R. T. Weitz, High-Mobility, Ultrathin Organic Semiconducting Films Realized by Surface-Mediated Crystallization, *Nano Lett.*, 2018, **18**, 9–14.

40 R. Noriega, J. Rivnay, K. Vandewal, F. P. V. Koch, N. Stingelin, P. Smith, M. F. Toney and A. Salleo, A general relationship between disorder, aggregation and charge transport in conjugated polymers, *Nature Mater.*, 2013, **12**, 1038–1044.

41 R. Noriega and A. Salleo, in *Organic Electronics II*, ed. H. Klauk, Wiley, 1st edn., 2012, pp. 67–104.

42 S. D. Baranovskii, Theoretical description of charge transport in disordered organic semiconductors, *Phys. Status Solidi B*, 2014, **251**, 487–525.

43 H. H. Choi, K. Cho, C. D. Frisbie, H. Sirringhaus and V. Podzorov, Critical assessment of charge mobility extraction in FETs, *Nature Mater.*, 2017, **17**, 2–7.

44 S. Fratini, H. Xie, I. N. Hulea, S. Ciuchi and A. F. Morpurgo, Current saturation and Coulomb interactions in organic single-crystal transistors, *New J. Phys.*, 2008, **10**, 033031.

45 S. Verlaak, V. Arkhipov and P. Heremans, Modeling of transport in polycrystalline organic semiconductor films, *Appl. Phys. Lett.*, 2003, **82**, 745–747.

46 G. Horowitz, Tunneling Current in Polycrystalline Organic Thin-Film Transistors, *Adv. Funct. Mater.*, 2003, **13**, 53–60.

47 H. Bässler, Charge Transport in Disordered Organic Photoconductors a Monte Carlo Simulation Study, *Phys. Status Solidi B*, 1993, **175**, 15–56.

48 S. Hunklinger, *Festkörperphysik*, De Gruyter, München, 4. Aufl., 2014.

49 H. Bässler and A. Köhler, in *Unimolecular and Supramolecular Electronics I*, ed. R. M. Metzger, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, vol. 312, pp. 1–65.

50 V. Coropceanu, J. Cornil, D. A. da Silva Filho, Y. Olivier, R. Silbey and J.-L. Brédas, Charge Transport in Organic Semiconductors, *Chem. Rev.*, 2007, **107**, 926–952.

51 O. Kallenberg, *Foundations of Modern Probability*, Springer New York, New York, NY, 2002.

52 C. Tanase, E. J. Meijer, P. W. M. Blom and D. M. Leeuw, Unification of the hole transport in polymeric field-effect transistors and light-emitting diodes, *Phys. Rev. Lett.*, 2003, **91**, 216601.

53 S. Baranovski, Ed., *Charge transport in disordered solids with applications in electronics*, Wiley, Chichester, England ; Hoboken, NJ, 2006.

54 S. D. Baranovskii, I. P. Zvyagin, H. Cordes, S. Yamasaki and P. Thomas, Percolation Approach to Hopping Transport in Organic Disordered Solids, *Phys. Status Solidi B*, 2002, **230**, 281–288.

55 L. Demeyu, S. Stafström and M. Bekele, Monte Carlo simulations of charge carrier mobility in semiconducting polymer field-effect transistors, *Phys. Rev. B*, 2007, **76**, 47.

56 B. El-Kareh, *Fundamentals of semiconductor processing technologies*, Kluwer Academic Publishers, Boston, 1995.

57 H.-R. Tseng, H. Phan, C. Luo, M. Wang, L. A. Perez, S. N. Patel, L. Ying, E. J. Kramer, T.-Q. Nguyen, G. C. Bazan and A. J. Heeger, High-mobility field-effect transistors fabricated with macroscopic aligned semiconducting polymers, *Adv. Mater.*, 2014, **26**, 2993–2998.

58 S. Schott, E. Gann, L. Thomsen, S.-H. Jung, J.-K. Lee, C. R. McNeill and H. Sirringhaus, Charge-Transport Anisotropy in a Uniaxially Aligned Diketopyrrolopyrrole-Based Copolymer, *Adv. Mater.*, 2015, **27**, 7356–7364.

59 D. P. Landau and K. Binder, *A guide to Monte Carlo simulations in statistical physics*, Cambridge University Press, Cambridge ; New York, 2nd ed., 2005.

60 N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, Equation of State Calculations by Fast Computing Machines, *J. Chem. Phys.*, 1953, **21**, 1087–1092.

61 H. Li and J.-L. Brédas, Modeling of Actual-Size Organic Electronic Devices from Efficient Molecular-Scale Simulations, *Adv. Funct. Mater.*, 2018, **28**, 1801460.

62 H. Li, L. Duan, D. Zhang, G. Dong, J. Qiao, L. Wang and Y. Qiu, Relationship between Mobilities from Time-of-Flight and Dark-Injection Space-Charge-Limited Current Measurements for Organic Semiconductors, *J. Phys. Chem. C*, 2014, **118**, 6052–6058.

63 H. Li, Y. Li, H. Li and J.-L. Brédas, Organic Field-Effect Transistors, *Adv. Funct. Mater.*, 2017, **27**, 1605715.

64 H. Bässler, Charge transport in molecularly doped polymers, *Philos. mag. B*, 1984, **50**, 347–362.

65 P. M. Borsenberger, L. Pautmeier and H. Bässler, Charge transport in disordered molecular solids, *J. Chem. Phys.*, 1991, **94**, 5447.

66 P. K. Watkins, A. B. Walker and G. L. B. Verschoor, Dynamical Monte Carlo Modelling of Organic Solar Cells: The Dependence of Internal Quantum Efficiency on Morphology, *Nano Lett.*, 2005, **5**, 1814–1818.

67 M. Mesta, M. Carvelli, R. J. de Vries, H. van Eersel, J. J. M. van der Holst, M. Schober, M. Furno, B. Lüssem, K. Leo, P. Loebl, R. Coehoorn and P. A. Bobbert, Molecular-scale simulation of electroluminescence in a multilayer white organic light-emitting diode, *Nature Mater.*, 2013, **12**, 652–658.

68 M. Pippig and F. Mercuri, Efficient evaluation of Coulomb interactions in kinetic Monte Carlo simulations of charge transport, *J. Chem. Phys.*, 2020, **152**, 164102.

69 V. Ambegaokar, B. I. Halperin and J. S. Langer, Hopping Conductivity in Disordered Systems, *Phys. Rev. B*, 1971, **4**, 2612–2620.

70 J. J. M. van der Holst, M. A. Uijttewaal, B. Ramachandhran, R. Coehoorn, P. A. Bobbert, G. A. de Wijs and R. A. de Groot, Modeling and analysis of the three-dimensional current density in sandwich-type single-carrier devices of disordered organic semiconductors, *Phys. Rev. B*, 2009, **79**, 085203.

71 J. Cottaar and P. A. Bobbert, Calculating charge-carrier mobilities in disordered semiconducting polymers: Mean field and beyond, *Phys. Rev. B*, 2006, **74**, 115204.

72 F. Liu, H. van Eersel, B. Xu, J. G. E. Wilbers, M. P. Jong, W. G. van der Wiel, P. A. Bobbert and R. Coehoorn, Effect of Coulomb correlation on charge transport in disordered organic semiconductors, *Phys. Rev. B*, , DOI:10.1103/PhysRevB.96.205203.

73 J. Zhou, Y. C. Zhou, J. M. Zhao, C. Q. Wu, X. M. Ding and X. Y. Hou, Carrier density dependence of mobility in organic solids, *Phys. Rev. B*, 2007, **75**, 416.

74 L. J. A. Koster, E. C. P. Smits, V. D. Mihailetchi and P. W. M. Blom, Device model for the operation of polymer/fullerene bulk heterojunction solar cells, *Phys. Rev. B*, 2005, **72**, 085205.

75 L. Meng, D. Wang, Q. Li, Y. Yi, J.-L. Brédas and Z. Shuai, An improved dynamic Monte Carlo model coupled with Poisson equation to simulate the performance of organic photovoltaic devices, *J. Chem. Phys.*, 2011, **134**, 124102.

76 H. Li and J.-L. Brédas, Developing molecular-level models for organic field-effect transistors, *Natl. Sci. Rev.*, 2020, **0**, 1–14.

77 J. J. M. van der Holst, F. W. A. van Oost, R. Coehoorn and P. A. Bobbert, Monte Carlo study of charge transport in organic sandwich-type single-carrier devices, *Phys. Rev. B*, 2011, **83**, 482.

78 H. Li and J.-L. Brédas, Kinetic Monte Carlo Modeling of Charge Carriers in Organic Electronic Devices: Suppression of the Self-Interaction Error, *J. Phys. Chem. Lett.*, 2017, **8**, 2507–2512.

79 P. Carbone and A. Troisi, Charge Diffusion in Semiconducting Polymers: Analytical Relation between Polymer Rigidity and Time Scales for Intrachain and Interchain Hopping, *J. Phys. Chem. Lett.*, 2014, **5**, 2637–2641.

80 R. Noriega, A. Salleo and A. J. Spakowitz, Chain conformations dictate multiscale charge transport phenomena in disordered semiconducting polymers, *Proc. Natl. Acad. Sci. U.S.A.*, 2013, **110**, 16315–16320.

81 H.-R. Tseng, L. Ying, B. B. Y. Hsu, L. A. Perez, C. J. Takacs, G. C. Bazan and A. J. Heeger, High Mobility Field Effect Transistors Based on Macroscopically Oriented Regioregular Copolymers, *Nano Lett.*, 2012, **12**, 6353–6357.

82 N. Yadav, N. Kumari, Y. Ando, S. S. Pandey and V. Singh, PCPDTBT copolymer based high performance organic phototransistors utilizing improved chain alignment, *Opt. Mater.*, 2021, **113**, 110886.

83 G. H. Roche, G. Bruckner, D. G. Dumitrescu, J. J. E. Moreau, A. Lee, G. Wantz and O. J. Dautel, Structural Odd–Even Effect Impacting the Dimensionality of Transport in BTBT-C *n* OH Organic Field Effect Transistors, *Adv. Electron. Mater.*, 2021, 2100265.

84 P. E. Rudnicki, Q. MacPherson, L. Balhorn, B. Feng, J. Qin, A. Salleo and A. J. Spakowitz, Impact of Liquid-Crystalline Chain Alignment on Charge Transport in Conducting Polymers, *Macromolecules*, 2019, **52**, 8932–8939.

85 H. Li and J.-L. Brédas, Quasi-One-Dimensional Charge Transport Can Lead to Nonlinear Current Characteristics in Organic Field-Effect Transistors, *J. Phys. Chem. Lett.*, 2018, **9**, 6550–6555.

86 D. Mendels and N. Tessler, A Comprehensive study of the Effects of Chain Morphology on the Transport Properties of Amorphous Polymer Films, *Sci. Rep.*, 2016, **6**, 29092.

87 A. K. Kron, The Monte Carlo method in statistical calculations of macromolecules, *Polym. Sci. U.S.S.R.*, 1965, **7**, 1361–1367.

88 F. T. Wall and F. Mandel, Macromolecular dimensions obtained by an efficient Monte Carlo method without sample attrition, *J. Chem. Phys.*, 1975, **63**, 4592–4595.

89 J. M. Frost, F. Cheynis, S. M. Tuladhar and J. Nelson, Influence of polymer-blend morphology on charge transport and photocurrent generation in donor-acceptor polymer blends, *Nano Lett.*, 2006, **6**, 1674–1681.

90 S. R. Mohan, M. P. Singh, M. P. Joshi and L. M. Kukreja, Monte Carlo Simulation of Carrier Diffusion in Organic Thin Films with Morphological Inhomogeneity, *J. Phys. Chem. C*, 2013, **117**, 24663–24672.

91 S. R. Mohan, M. P. Joshi and M. P. Singh, Charge transport in disordered organic solids: A Monte Carlo simulation study on the effects of film morphology, *Org. Electron.*, 2008, **9**, 355–368.

92 F. Steiner, C. Poelking, D. Niedzialek, D. Andrienko and J. Nelson, Influence of orientation mismatch on charge transport across grain boundaries in tri-isopropylsilylethynyl (TIPS) pentacene thin films, *Phys. Chem. Chem. Phys.*, 2017, **19**, 10854–10862.

93 C. Li, L. Duan, Y. Sun, H. Li and Y. Qiu, Charge Transport in Mixed Organic Disorder Semiconductors: Trapping, Scattering, and Effective Energetic Disorder, *J. Phys. Chem. C*, 2012, **116**, 19748–19754.

94 C. Li, L. Duan, H. Li and Y. Qiu, Universal Trap Effect in Carrier Transport of Disordered Organic Semiconductors, *J. Phys. Chem. C*, 2014, **118**, 10651–10660.

95 T. A. Madison, A. G. Gagorik and G. R. Hutchison, Charge Transport in Imperfect Organic Field Effect Transistors: Effects of Charge Traps, *J. Phys. Chem. C*, 2012, **116**, 11852–11858.

96 P. M. Borsenberger, L. T. Pautmeier and H. Bässler, Nondispersive-to-dispersive charge-transport transition in disordered molecular solids, *Phys. Rev. B*, 1992, **46**, 12145–12153.

97 G. Schönherr, H. Bässler and M. Silver, Dispersive hopping transport via sites having a Gaussian distribution of energies, *Philos. mag. B*, 1981, **44**, 47–61.

98 W. Nolting, *Grundkurs Theoretische Physik 3*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

99 A. Miller and E. Abrahams, Impurity Conduction at Low Concentrations, *Phys. Rev.*, 1960, **120**, 745–755.

100 M. S. Tamboli, P. K. Palei, S. S. Patil, M. V. Kulkarni, N. N. Maldar and B. B. Kale, Polymethyl methacrylate (PMMA)–bismuth ferrite (BFO) nanocomposite: low loss and high dielectric constant materials with perceptible magnetic properties, *Dalton Trans.*, 2014, **43**, 13232–13241.

101 V. I. Arkhipov, P. Heremans, E. V. Emelianova and H. Bässler, Effect of doping on the density-of-states distribution and carrier hopping in disordered organic semiconductors, *Phys. Rev. B*, 2005, **71**, 045214.

102 A. Sharma, F. W. A. van Oost, M. Kemerink and P. A. Bobbert, Dimensionality of charge transport in organic field-effect transistors, *Phys. Rev. B*, 2012, **85**, 235302.

103 M. Koopmans, M. A. T. Leiviskä, J. Liu, J. Dong, L. Qiu, J. C. Hummelen, G. Portale, M. C. Heiber and L. J. A. Koster, Electrical Conductivity of Doped Organic Semiconductors Limited by Carrier–Carrier Interactions, *ACS Appl. Mater. Interfaces*, 2020, **12**, 56222–56230.

104 H. van Eersel, P. A. Bobbert, R. A. J. Janssen and R. Coehoorn, Monte Carlo study of efficiency roll-off of phosphorescent organic light-emitting diodes: Evidence for dominant role of triplet-polaron quenching, *Appl. Phys. Lett.*, 2014, **105**, 143303.

105 H. van Eersel, P. A. Bobbert and R. Coehoorn, Kinetic Monte Carlo study of triplet-triplet annihilation in organic phosphorescent emitters, *J. Appl. Phys.*, 2015, **117**, 115502.

106 F. Liu, H. van Eersel, P. A. Bobbert and R. Coehoorn, Three-Dimensional Modeling of Bipolar Charge-Carrier Transport and Recombination in Disordered Organic Semiconductor Devices at Low Voltages, *Phys. Rev. Applied*, , DOI:10.1103/PhysRevApplied.10.054007.

107 R. Coehoorn, X. Lin, C. H. L. Weijtens, S. Gottardi and H. van Eersel, Three-Dimensional Modeling of Organic Light-Emitting Diodes Containing Molecules with Large Electric Dipole Moments, *Phys. Rev. Applied*, 2021, **16**, 034048.

108 T. Meier, H. Bässler and A. Köhler, The Impact of Grain Boundaries on Charge Transport in Polycrystalline Organic Field-Effect Transistors, *Adv. Optical Mater.*, 2021, **9**, 2100115.

109 J.-I. Park, J. W. Chung, J.-Y. Kim, J. Lee, J. Y. Jung, B. Koo, B.-L. Lee, S. W. Lee, Y. W. Jin and S. Y. Lee, Dibenzothiopheno[6,5-b:6′,5′-f]thieno[3,2-b]thiophene (DBTTT): High-Performance Small-Molecule Organic Semiconductor for Field-Effect Transistors, *J. Am. Chem. Soc.*, 2015, **137**, 12175–12178.

110 N. Kurihara, A. Yao, M. Sunagawa, Y. Ikeda, K. Terai, H. Kondo, M. Saito, H. Ikeda and H. Nakamura, High-Mobility Organic Thin-Film Transistors Over 10 $cm^{2} V^{-1} s^{-1}$ Fabricated Using Bis(benzothieno)naphthalene Polycrystalline Films, *Jpn. J. Appl. Phys.*, 2013, **52**, 05DC11.

111 S. Fratini, D. Mayou and S. Ciuchi, The Transient Localization Scenario for Charge Transport in Crystalline Organic Materials, *Adv. Funct. Mater.*, 2016, **26**, 2292–2315.

112 D. Venkateshvaran, M. Nikolka, A. Sadhanala, V. Lemaur, M. Zelazny, M. Kepa, M. Hurhangee, A. J. Kronemeijer, V. Pecunia, I. Nasrallah, I. Romanov, K. Broch, I. McCulloch, D. Emin, Y. Olivier, J. Cornil, D. Beljonne and H. Sirringhaus, Approaching disorder-free transport in high-mobility conjugated polymers, *Nature*, 2014, **515**, 384–388.

113 K. Takimiya, S. Shinamura, I. Osaka and E. Miyazaki, Thienoacene-Based Organic Semiconductors, *Adv. Mater.*, 2011, **23**, 4347–4370.

114 K. Zhang, Z. Wang, T. Marszalek, M. Borkowski, G. Fytas, P. W. M. Blom and W. Pisula, Key role of the meniscus shape in crystallization of organic semiconductors during meniscus-guided coating, *Mater. Horiz.*, 2020, **7**, 1631–1640.

115 M. Kim, S. U. Ryu, S. A. Park, K. Choi, T. Kim, D. Chung and T. Park, Donor–Acceptor-Conjugated Polymer for High-Performance Organic Field-Effect Transistors: A Progress Report, *Adv. Funct. Mater.*, 2020, **30**, 1904545.

116 H. Kleemann, K. Krechan, A. Fischer and K. Leo, A Review of Vertical Organic Transistors, *Adv. Funct. Mater.*, 2020, **30**, 1907113.

117 K. Puntambekar, J. Dong, G. Haugstad and C. D. Frisbie, Structural and Electrostatic Complexity at a Pentacene/Insulator Interface, *Adv. Funct. Mater.*, 2006, **16**, 879–884.

118 M. Tello, M. Chiesa, C. M. Duffy and H. Sirringhaus, Charge Trapping in Intergrain Regions of Pentacene Thin Film Transistors, *Adv. Funct. Mater.*, 2008, **18**, 3907–3913.

119 L. C. Teague, B. H. Hamadani, O. D. Jurchescu, S. Subramanian, J. E. Anthony, T. N. Jackson, C. A. Richter, D. J. Gundlach and J. G. Kushmerick, Surface Potential Imaging of Solution Processable Acene-Based Thin Film Transistors, *Adv. Mater.*, 2008, **20**, 4513–4516.

120 C. Qian, J. Sun, L. Zhang, H. Huang, J. Yang and Y. Gao, Crystal-Domain Orientation and Boundary in Highly Ordered Organic Semiconductor Thin Film, *J. Phys. Chem. C*, 2015, **119**, 14965–14971.

121 M. J. Jaquith, J. E. Anthony and J. A. Marohn, Long-lived charge traps in functionalized pentacene and anthradithiophene studied by time-resolved electric force microscopy, *J. Mater. Chem.*, 2009, **19**, 6116–6123.

122 T. W. Kelley and C. D. Frisbie, Gate Voltage Dependent Resistance of a Single Organic Semiconductor Grain Boundary, *J. Phys. Chem. B*, 2001, **105**, 4538–4540.

123 M. Mladenović, N. Vukmirović and I. Stanković, Electronic States at Low-Angle Grain Boundaries in Polycrystalline Naphthalene, *J. Phys. Chem. C*, 2013, **117**, 15741–15748.

124 M. Mladenović and N. Vukmirović, Electronic States at the Interface between Crystalline and Amorphous Domains in Conjugated Polymers, *J. Phys. Chem. C*, 2015, **119**, 23329–23333.

125 L. G. Kaake, P. F. Barbara and X.-Y. Zhu, Intrinsic Charge Trapping in Organic and Polymeric Semiconductors: A Physical Chemistry Perspective, *J. Phys. Chem. Lett.*, 2010, **1**, 628–635.

126 S. Verlaak and P. Heremans, Molecular microelectrostatic view on electronic states near pentacene grain boundaries, *Phys. Rev. B*, 2007, **75**, 115127.

127 A. Einstein, Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen, *Ann. Phys.*, 1905, **322**, 549–560.

128 T. Pang, *An introduction to computational physics*, Cambridge University Press, Cambridge, New York, 2nd ed., 2008.

129 W. Nolting, *Grundkurs Theoretische Physik 5/1: Quantenmechanik - Grundlagen*, Springer, Berlin, Heidelberg, 7th edn., 2009, vol. 5/1.

130 Q. Li, Z. Yao, Y. Lu, S. Zhang, Z. Ahmad, J. Wang, X. Gu and J. Pei, Achieving High Alignment of Conjugated Polymers by Controlled Dip-Coating, *Adv. Electron. Mater.*, 2020, **6**, 2000080.

131 M. J. Han, D.-W. Lee, E. K. Lee, J.-Y. Kim, J. Y. Jung, H. Kang, H. Ahn, T. J. Shin, D. K. Yoon and J.-I. Park, Molecular Orientation Control of Liquid Crystal Organic Semiconductor for High-Performance Organic Field-Effect Transistors, *ACS Appl. Mater. Interfaces*, 2021, **13**, 11125–11133.

132 M. Watanabe, Y. J. Chang, S.-W. Liu, T.-H. Chao, K. Goto, Md. M. Islam, C.-H. Yuan, Y.-T. Tao, T. Shinmyozu and T. J. Chow, The synthesis, crystal structure and charge-transport properties of hexacene, *Nature Chem.*, 2012, **4**, 574–578.

133 Y.-F. Huang, S.-T. Chang, K.-Y. Wu, S.-L. Wu, G.-T. Ciou, C.-Y. Chen, C.-L. Liu and C.-L. Wang, Influences of Conjugation Length on Organic Field-Effect Transistor Performances and Thin Film Structures of Diketopyrrolopyrrole-Oligomers, *ACS Appl. Mater. Interfaces*, 2018, **10**, 8869–8876.

134 C. R. Singh, G. Gupta, R. Lohwasser, S. Engmann, J. Balko, M. Thelakkat, T. Thurn-Albrecht and H. Hoppe, Correlation of charge transport with structural order in highly ordered melt-crystallized poly(3-hexylthiophene) thin films, *J. Polym. Sci. B Polym. Phys.*, 2013, **51**, 943–951.

135 S. A. Mollinger, A. Salleo and A. J. Spakowitz, Anomalous Charge Transport in Conjugated Polymers Reveals Underlying Mechanisms of Trapping and Percolation, *ACS Cent. Sci.*, 2016, **2**, 910–915.

136 R. S. Fedorenko, A. V. Kuevda, V. A. Trukhanov, V. G. Konstantinov, A. Yu. Sosorev, A. A. Sonina, M. S. Kazantsev, N. M. Surin, S. Grigorian, O. V. Borshchev, S. A. Ponomarenko and D. Yu. Paraschuk, Luminescent High-Mobility 2D Organic Semiconductor Single Crystals, *Adv. Electron. Mater.*, 2022, **8**, 2101281.

137 H. Iino, T. Usui and J. Hanna, Liquid crystals for organic thin-film transistors, *Nat. Commun.*, 2015, **6**, 6828.

138 S. Athanasopoulos, S. T. Hoffmann, H. Bässler, A. Kohler and D. Beljonne, To Hop or Not to Hop?, *J. Phys. Chem. Lett.*, 2013, **4**, 1694–1700.

139 W. Shockley, A Unipolar “Field-Effect” Transistor, *Proc. IRE*, 1952, **40**, 1365–1376.

140 A. F. Paterson, S. Singh, K. J. Fallon, T. Hodsden, Y. Han, B. C. Schroeder, H. Bronstein, M. Heeney, I. McCulloch and T. D. Anthopoulos, Recent Progress in High-Mobility Organic Transistors: A Reality Check, *Adv. Mater.*, 2018, **30**, 1801079.

141 H. Phan, M. J. Ford, A. T. Lill, M. Wang, G. C. Bazan and T.-Q. Nguyen, Electrical Double-Slope Nonideality in Organic Field-Effect Transistors, *Adv. Funct. Mater.*, 2018, **28**, 1707221.

142 A. Yamamura, S. Watanabe, M. Uno, M. Mitani, C. Mitsui, J. Tsurumi, N. Isahaya, Y. Kanaoka, T. Okamoto and J. Takeya, Wafer-scale, layer-controlled organic single crystals for high-speed circuit operation, *Sci. Adv.*, 2018, **4**, eaao5758.

143 R. W. I. de Boer, N. N. Iosad, A. F. Stassen, T. M. Klapwijk and A. F. Morpurgo, Influence of the gate leakage current on the stability of organic single-crystal field-effect transistors, *Appl. Phys. Lett.*, 2005, **86**, 032103.

144 J. Zaumseil and H. Sirringhaus, Electron and Ambipolar Transport in Organic Field-Effect Transistors, *Chem. Rev.*, 2007, **107**, 1296–1323.

145 M. E. Gershenson, V. Podzorov and A. F. Morpurgo, Colloquium : Electronic transport in single-crystal organic transistors, *Rev. Mod. Phys.*, 2006, **78**, 973–989.

146 S. Jiang, J. Qian, Q. Wang, Y. Duan, J. Guo, B. Zhang, H. Sun, X. Wang, C. Liu, Y. Shi and Y. Li, Probing Coulomb Interactions on Charge Transport in Few-Layer Organic Crystalline Semiconductors by the Gated van der Pauw Method, *Adv. Electron. Mater.*, 2020, **6**, 2000136.

147 M. J. Panzer and C. D. Frisbie, Polymer Electrolyte Gate Dielectric Reveals Finite Windows of High Conductivity in Organic Thin Film Transistors at High Charge Carrier Densities, *J. Am. Chem. Soc.*, 2005, **127**, 6960–6961.

148 M. J. Panzer and C. D. Frisbie, High charge carrier densities and conductance maxima in single-crystal organic field-effect transistors with a polymer electrolyte gate dielectric, *Appl. Phys. Lett.*, 2006, **88**, 203504.

149 M. J. Panzer and C. D. Frisbie, Polymer Electrolyte-Gated Organic Field-Effect Transistors: Low-Voltage, High-Current Switches for Organic Electronics and Testbeds for Probing Electrical Transport at High Charge Carrier Density, *J. Am. Chem. Soc.*, 2007, **129**, 6599–6607.

150 Y. Xia, W. Xie, P. P. Ruden and C. D. Frisbie, Carrier localization on surfaces of organic semiconductors gated with electrolytes, *Phys. Rev. Lett.*, 2010, **105**, 036802.

151 G. A. H. Wetzelaer, L. J. A. Koster and P. W. M. Blom, Validity of the Einstein Relation in Disordered Organic Semiconductors, *Phys. Rev. Lett.*, 2011, **107**, 066605.

11. Publikationsliste

Im Lauf meiner Promotion sind verschiedene Publikationen entstanden. Davon habe ich in dieser Arbeit folgende Publikation verwendet:

T. Meier, H. Bässler und A. Köhler, The Impact of Grain Boundaries on Charge Transport in Polycrystalline Organic Field-Effect Transistors, *Adv. Optical Mater.*, 2021, **9**, 2100115.

Folgende weitere Publikationen sind entstanden, die nicht Teil dieser Arbeit sind:

F. Panzer, C. Li, T. Meier, A. Köhler und S. Hüttner, Impact of Structural Dynamics on the Optical Properties of Methylammonium Lead Iodide Perovskites, *Adv. Energy Mater.*, 2017, **7**, 1700286.

T. Meier, T. Gujar, A. Schönleber, S. Olthof, K. Meerholz, S. van Smaalen, F. Panzer, M. Thelakkat und A. Köhler, Impact of excess PbI_2 on the structure and the temperature dependent optical properties of methylammonium lead iodide perovskites, *J. Mater. Chem. C*, 2018, **6**, 7512–7519.

F. Rodella, S. Bagnich, E. Duda, T. Meier, J. Kahle, S. Athanasopoulos, A. Köhler und P. Strohriegel, High Triplet Energy Host Materials for Blue TADF OLEDs—A Tool Box Approach, *Front. Chem.*, 2020, **8**, 657.

R. Saxena, T. Meier, S. Athanasopoulos, H. Bässler und A. Köhler, Kinetic Monte Carlo Study of Triplet-Triplet Annihilation in Conjugated Luminescent Materials, *Phys. Rev. Applied*, 2020, **14**, 034050.

12. Danksagung

An dieser Stelle möchte ich einigen Personen besonders danken, die auf direkte oder indirekte Weise zum Gelingen dieser Arbeit beigetragen haben.

An erster Stelle gilt mein Dank Prof. Dr. Anna Köhler für die Betreuung meiner Doktorarbeit. Danke, dass du mir diese Promotion ermöglicht hast und mir während meiner Arbeit auch so viel Freiraum in Bezug auf meine zeitliche Einteilung und den thematischen Schwerpunkt gelassen hast. Besonders danke ich dir für die letzten zwei Jahre, in denen du mich weiterhin so gut betreut hast. Danke für die vielen Besprechungen, oft auch spät am Abend! Ohne diese regelmäßigen Treffen wäre diese Arbeit zum jetzigen Zeitpunkt nicht fertig geworden.

Als nächstes möchte ich meinen Kollegen am Lehrstuhl danken für so manche Kickerrunde nach der Mittagspause, die den Arbeitsalltag bestens aufgelockert hat. Danke auch an Frank, Irene und Thomas für so manche technische Unterstützung, nicht nur im Zusammenhang mit meiner Promotion.

Mein Dank gilt auch Prof. Dr. Heinz Bässler für die geistreichen Diskussionen, die mir geholfen haben mich tiefer in die Physik hineinzudenken.

Besonders danke ich auch allen Freunden, die nicht nur während meiner Promotionszeit an meiner Seite standen, für verschiedenerlei Arten von Unternehmungen, ob zahlloses Klettern oder auch einfach nur Kaffeetrinken. Ihr bereichert mein Leben sehr und habt mir geholfen diese Promotionszeit durchzustehen.

Nicht zuletzt danke ich dem, ohne den diese Arbeit nicht zustande gekommen wäre, meinem Gott.

13. Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass ich die Hilfe von gewerblichen Promotionsberatern bzw. -vermittlern oder ähnlichen Dienstleistern weder bisher in Anspruch genommen habe, noch künftig in Anspruch nehmen werde.

Zusätzlich erkläre ich hiermit, dass ich keinerlei frühere Promotionsversuche unternommen habe.

Bayreuth, den

Tobias Meier

A. Anhang

Die im Rahmen dieser Dissertation von mir entwickelten Simulationsprogramme und Hilfsskripte habe ich im Folgenden aufgeführt. Alle Programme sind in Python geschrieben und können mit der Version Python 3.8 und höher verwendet werden.

A.1. Skripte zum Erstellen der Morphologie

In Kapitel 5.4 habe ich zwei Algorithmen beschrieben, mit denen die Morphologie des organischen Halbleiters dargestellt werden kann. Die beiden aufgeführten Skripte können in der dargestellten Form in die kommerzielle Software Bumblebee eingegeben werden und mit den gewünschten Parametern verwendet werden.

A.1.1. Skript zur Erzeugung einer polykristallinen Struktur aus Kapitel 5.4.1

```
# -*- coding: utf-8 -*-
"""
Tobias Meier.

Create a morphology of crystalline domains.

In the composition section the background material has to have the index 0.
The material index for the chains has to be 1.

The argument passed to the script has to be a list of one dictionary. One argument has
to be "kind": "polymer" to activate the chains.in file. Also double quotes have to be
used.
"""
from __future__ import division
import argparse
import yaml
import numpy as np
import itertools

# assume a cubic grid
gridspacing = np.array([1, 1, 1])

def chain_generator(layer):
    """
    Create the chain morphology.
    """
    box = np.array([layer['nx'], layer['ny'], layer['nz']], dtype=np.int32)

    # user-defined parameters in the composition section
    params = layer['parameters'][0]

    crystal_a, crystal_b = 1.2, 1
    crystal_gamma = 60
    if 'crystal_a' in params:
        crystal_a = params['crystal_a']
    if 'crystal_b' in params:
        crystal_b = params['crystal_b']
    if 'crystal_gamma' in params:
        crystal_gamma = params['crystal_gamma']

    # create the chain morphology
    chains = create_domain(box, crystal_a, crystal_b, crystal_gamma, \
                           params['number_seeds'], params['boundary_gap'])

    return chains

def create_domain(box, crystal_a, crystal_b, crystal_gamma, number_seeds,
boundary_gap=0):
    """
    box : 1D array
        size of the simulation box
    crystal_a, crystal_b : float, float
        length of the edges of the parallelograms for the crystallites (nm)
    crystal_gamma : float
        angle of the crystallites (parallelograms) (degree)
    number_seeds : int
        number of seeds for crystallite growth, placed on a quadratic mesh
    boundary_gap : float
```

```
    gap between two crystallites (in nm)

returns
-------
morphology, eta_boundaries : 3D array, float
    morphology
"""
# x und z Achse vertauschen
box = box[::-1]
# Winkel in Bogenmass umrechnen
crystal_gamma = crystal_gamma / 180 * np.pi

# Abstand zwischen den Startpunkten fuer Kristallitwachstum
dist = box[:2] / np.sqrt(number_seeds)
# Startpunkte fuer Kristallitwachstum
seeds = np.array(list(itertools.product(
        range(int(np.sqrt(number_seeds))), repeat=2))) * dist + dist/2
# Zahlenwerte ganzzahlig machen
seeds = np.array(seeds, dtype=np.int)

# eine Schicht der Morhpologie (y,z layer)
layer = np.full(box[:2], -1, dtype=np.int)
# Indizes mit maximalem Abstand zu einem Punkt
sphere = SphereIndices(boundary_gap, gridspacing[1:])
# Ausrichtung der Kristallite
seed_directions = np.random.random(size=number_seeds) * 2 * np.pi

# pro Schleifendurchlauf ein Kristallwachstumsschritt
# Schleifenvariable
i = 0
while True:
    # alte Morphologieschicht speichern
    old_layer = layer.copy()
    # jeder Kristallit waechst nacheinander
    for s,seed in enumerate(seeds):
        # Index des Kristallits
        crystallite = s
        # Position des Wachstumspunkts
        sx, sy = seed
        # Winkel der Richtung des Kristallits
        phi = seed_directions[s]

        # meshgrid fuer alle Gitterpunkte
        x, y = np.meshgrid(np.arange(-box[0], 2 * box[0]), \
                           np.arange(-box[1], 2 * box[1]), indexing='ij')
        x = x.flatten()
        y = y.flatten()

        # Spezialfall 90 grad wird nicht benötigt
        # erstes Geradenpaar
        m = np.tan(phi)
        t = abs(i * crystal_b * np.sin(crystal_gamma) / np.sin(np.pi / 2 + phi))
        x, y = BetweenLines(x, y, sx, sy, m, t)

        # zweites Geradenpaar
        m = np.tan(phi + crystal_gamma)
        t = abs(i * crystal_a * (np.cos(phi)*np.tan(phi + crystal_gamma) -
np.sin(phi)))
        x, y = BetweenLines(x, y, sx, sy, m, t)

        # alle noch unbesetzten Plaetze
        mask = np.where(layer[tuple((x % box[0], y % box[1]))] == -1)
        x = x[mask]
        y = y[mask]

        # Dicke der Grain boundaries
        if boundary_gap > 0:
            # durchlaufe alle moeglichen neuen Gitterplaetze
            for new in zip(x, y):
                # alle Punkte innerhalb eines Abstands um den neuen Punkt
                p = np.transpose((new + sphere) % box[:2])
                # teste, ob alle Plaetze innerhalb einer Kugel mit Radius der
Dicke
                # des Abstands zwischen den Kristalliten entweder unbesetzt oder
                # mit dem aktuellen Kristallit belegt sind
                if np.all(np.any(np.stack((layer[tuple(p)] == -1, \
                                layer[tuple(p)] == crystallite)), axis=0)):
                    layer[tuple(new % box[:2])] = crystallite
        else:
            layer[tuple((x % box[0], y % box[1]))] = crystallite
    # Wenn alle Plaetze besetzt sind, bricht das Wachstum ab
    if np.all(old_layer == layer):
        break
    # Schleifenvariable erhoehen
    i += 1

# Grain boundaries in Fallenzustaende umwandeln
# Bereich der Grain boundaries
mask = np.where(layer == -1)
layer[mask] = -1
# Morphologie in die dritte Dimension ausdehnen
morphology = np.empty(box, dtype=np.int)
morphology[:] = layer.reshape((box[0], box[1], 1))
morphology = np.transpose(morphology, axes=(2, 1, 0))
return morphology

def BetweenLines(x, y, sx, sy, m, t):
    """
    Gebe die Punkte zurueck, die sich zwischen den Geraden
    y = m*x + t
    und
    y = m*x - t
    befinden.

    x, y : 1D arrays
        x- und y-Koordinaten der Punkte
    sx, sy :
        Punkt, um den die Geraden verschoben sind.
    m : float
        Steigung der Geraden
```

```
    t : float
        Achsenabschnitt
    """
    mask = np.where(y - sy <= m * (x - sx) + t)
    x = x[mask]
    y = y[mask]
    mask = np.where(y - sy >= m * (x - sx) - t)
    x = x[mask]
    y = y[mask]
    return x, y

def SphereIndices(R, gridspacing, include_origin=False):
    """
    Ermittle die Koordinaten, die innerhalb einer Kugel mit festem Radius
    liegen. Der Kugelmittelpunkt liegt dabei auf der Null.

    R : float
        Radius der Kugel
    gridspacing : 1D array
        Gitterabstaende, [dx, dy, dz]
    include_origin : bool
        Soll der Ursprungspunkt [0,0,0] mit in der Liste enthalten sein?

    returns
    -------
    indices : 2D array
        Liste der Koordinaten, die innerhalb der Kugel liegen. Index 0: Punkte,
        Index 1: [x,y,z]
    """
    # Anzahl der Dimensionen
    dim = len(gridspacing)
    # Anzahl der maximal moeglichen Gitterpunkte in einer Richtung
    n = int(np.max(R / gridspacing))
    indices = np.array(list(itertools.product(range(-n, n+1), repeat=dim)))
    length = np.sqrt(np.sum((indices * gridspacing)**2, axis=1))
    indices = indices[np.where(length <= R)]
    if not include_origin:
        indices = indices[np.any(np.not_equal(indices, [0] * dim), axis=1)]
    return indices

def composition_generator(chains):
    """
    Generate the composition from the chain morphology.
    In the composition section the background material has to have the index 0.
    The material index for the chains has to be 1.

    Returns
    -------
    composition (3D array): The array with the material indices corresponding to
    the chain morphology.
    """
    composition = np.zeros_like(chains)

    # find the sites corresponding to chains
    chain_indices = np.where(chains >= 0)

    # create the composition
    composition[chain_indices] = 1

    return composition

def write_composition(mat_dist, layer_index, disorder):
    """
    Write the 3D composition array of a layer to file.
    The output filename should be:
        composition_{layer_index}_{boxid}.in

    Inputs:
        mat_dist (np 3Darray, int): array containing the materials indexes
        layer_index (int): index of the layer
        disorder (int): box index
    """
    filename = "composition_{0}_{1:03d}.in".format(layer_index, disorder)

    with open(filename, "w") as f:
        for k in range(mat_dist.shape[2]):
            for j in range(mat_dist.shape[1]):
                for i in range(mat_dist.shape[0]):
                    f.write("{} ".format(mat_dist[i, j, k]))
                f.write("\n")
            f.write("\n")

def write_chains(chains, layer_index, disorder):
    """
    Write the chain file.

    Input:
        chains (np 3D array, int): array with indices of the chains
        layer_index (int): index of the layer
        disorder (int): box index

    output file:
        chains_{layer_index}_{boxid}.in
    """
    fn = 'chains_{0}_{1:03d}.in'.format(layer_index, disorder)

    with open(fn, 'w') as f:
        for k in range(chains.shape[2]):
            for j in range(chains.shape[1]):
                for i in range(chains.shape[0]):
                    f.write("{} ".format(chains[i, j, k]))
                f.write("\n")
            f.write("\n")

def morphology_generator(filename, layer_index, disorder):
    """
    Load the morphology from the yaml input file generated by Bumblebee
    and generate the composition for the layer

    Inputs:
```

```
        filename (string): name of the yaml file with input parameters
        layer_index (int): index of the layer
        disorder (int): box index
    """
    if disorder < 1:
        raise ValueError("disorder number must be > 0")
    if layer_index < 0:
        raise ValueError("layer index must be >= 0")

    # load layer parameters
    layer_params = yaml.load(open(filename, "r"),
Loader=yaml.FullLoader)["morphology"][layer_index]

    # seed the random generator
    np.random.seed((layer_index + 1)*disorder)

    # create the morphology of chains
    chains = chain_generator(layer_params)

    # generate the 3D distribution of materials in the layer
    lay_compo = composition_generator(chains)

    # write the required composition files
    write_composition(lay_compo, layer_index, disorder)

    # write the chains file
    write_chains(chains, layer_index, disorder)

################################################################################
### DO NOT CHANGE HERE ###

# get input arguments
parser = argparse.ArgumentParser()
parser.add_argument('layer', type=int, help='input layer number')
parser.add_argument('disorder', type=int, help='input disorder number')
parser.add_argument('--input-file', type=str, default="morphology.yml", help='input
file name')
arg = parser.parse_args()

# generate composition
morphology_generator(arg.input_file, arg.layer, arg.disorder)
```

A.1.2. Skript zur Erzeugung einer oligomeren Struktur aus Kapitel 5.4.2

```
# -*- coding: utf-8 -*-
"""
Tobias Meier.

Create a morphology of conjugated segments.

The oligomer chains can be displaced in x and y direction separately (z is
source-drain direction).

In the composition section the background material has to have the index 0. This
can also be used for trap states.
The material for the chains can be of different types, e.g. the chains can be
assigned with different energies in order to mimic a gaussian distribution of
site energies among the chains.
For this, the occurence of every material has to be given as a list. The number
of materials used will be calculated from its length. The occurence corresponds
to the number of chains with a specific material, so the sum of all occurences
times the chain length equals the total number of sites.
The parameters will look like this:
    "occurence": [10, 20, 35, 100, 35, 20, 10]
The number of materials in the composition section has to coincide with the length
of the list plus the background material.

The argument passed to the script has to be a list of one dictionary. One argument
has to be "kind": "polymer" to activate the chains.in file. Also double quotes
have to be used.

Updates:
    - Loader=yaml.FullLoader in yaml.load() has been added.
"""
from __future__ import division
import argparse
import yaml
import numpy as np

def chain_generator(layer):
    """
    Create the chain morphology.
    """
    box = np.array([layer['nx'], layer['ny'], layer['nz']], dtype=np.int32)

    # user-defined parameters in the composition section
    params = layer['parameters'][0]

    # some default parameters
    trapdensity = 0
    chain_sigma = 0
    orientation = 2
    if 'trapdensity' in params:
        trapdensity = params['trapdensity']
    if 'chain_sigma' in params:
        chain_sigma = params['chain_sigma']
    if 'orientation' in params:
        orientation = params['orientation']

    # create the chain morphology
```

```
    chains = create_domain(box, params['chain_length'], params['chain_offset'],
chain_sigma=chain_sigma, \
              orientation=orientation, trapdensity=trapdensity)

    return chains

def create_domain(box, chain_length, chain_offset, chain_sigma=0, orientation=2,
trapdensity=0., periodic=True):
    """
    Erstelle einen Bereich fuer die Morphologie, in dem alle Ketten in einer
    Richtung verlaufen.
    - Es werden Gitterpunkte zu Ketten verbunden. Dabei werden alle Gitterpunkte
      durchnummeriert, wobei Gitterpunkte einer Kette die gleiche Nummer erhalten.
    - Es muss nicht jede Zahl als Nummer einer Kette vorkommen.
    - Kann verwendet werden, um mehrere solcher Kaesten aneinanderzureihen.
    - Optionalen Parameter, wenn Ketten periodisch sein sollen.
    - Fallenzustaende haben negative Nummern, jeder seinen eigenen Wert.

    box : 1D array
        Groesse der Morphologie
    chain_length : int
        Laenge der Ketten
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufaelligen, keinen oder einen bestimmten
        Versatz haben.
    chain_sigma : float
        width of the distribution of chain lengths
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)
    trapdensity : float
        Wahrscheinlichkeit, dass ein Gitterplatz ein Fallenzustand ist
    periodic : bool
        Sollen die Ketten ueber die Raender periodisch fortgesetzt werden?

    returns
    -------
    morphology : 3D array
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplaetze zur gleichen Polymerkette gehoeren. Ein Wert von -1 bedeutet
        einen Fallenzustand.
    """
    box = np.asarray(box)
    # die Richtung, in der die Ketten durchnummeriert werden, ist die Achse
    # der Orientierung. Vertausche die Achsen so, dass die Achse der Kettenausrichtung
    # die letzte Achse ist.
    shape = box[np.roll(np.arange(3), 2 - orientation)]
    # Achsen, die wieder zurueck vertauscht werden muessen
    transpose = np.roll(np.arange(3), orientation - 2)

    # wenn keine periodische Randbedingung fuer die Kette gelten soll, dann eine
    # Kette mehr mitnehmen
    if not periodic:
        shape[2] += chain_length

    # ------------------------------------------------------------------------
    # Polymerketten mit bestimmter Laenge erstellen
    #
    # Array fuer die Morphologie
    morphology = np.zeros(shape, dtype=np.int)
    # Laufindex der Kettennumer
    k = 1
    # die zwei Dimensionen durchlaufen, die nicht der Kettenausrichtung entsprechen
    for i in range(shape[0]):
        for j in range(shape[1]):
            # ein Kettenstrang (1D)
            line = []
            # Fuege solange Segmente hinzu, bis der Strang voll ist
            while True:
                # Laenge der neuen Ketten aus Verteilung bestimmen
                l = int(np.random.normal(loc=chain_length, scale=chain_sigma) + 0.5)
                # kann diese Kettenlaenge noch hinzugefuegt werden?
                if len(line) + l <= shape[2]:
                    line.extend([k]*l)
                    # Kettenindex erhoehen
                    k += 1
                else:
                    line.extend([k]*(shape[2] - len(line)))
                    # Strang zur Morphologie hinzufuegen
                    morphology[i,j] = line
                    # Kettenindex erhoehen
                    k += 1
                    break

    # ------------------------------------------------------------------------
    # Fallenzustaende einfuegen
    #
    if trapdensity > 0:
        # Laufindex fuer die Fallenzustaende
        trap = -1
        # Alle Gitterplaetze werden durchlaufen
        for i in range(shape[0]):
            for j in range(shape[1]):
                for k in range(box[orientation]):
                    # zufaellig eine Falle platzieren
                    if np.random.random() < trapdensity:
                        # Wenn eine Kette auseinandergerissen werden wuerde,
                        # wird die Falle nach hinten verschoben
                        x = k
                        while morphology[i, j, x-1] == morphology[i, j, x]:
                            x += 1
                            if x >= box[orientation]:
                                break
                        # Falle einbauen
                        if x < box[orientation]:
                            morphology[i, j] = np.append(morphology[i, j, :x], \
                                       np.append(trap, morphology[i, j, x:-1]))
    # Versatz der Ketten nebeneinander
    offset = ChainOffset(box, chain_length, chain_offset, orientation)
    # Verschiebe die Kettennummern entsprechend dem Versatz
```

```
    for i in range(shape[0]):
        for j in range(shape[1]):
            morphology[i,j,:] = np.roll(morphology[i,j,:], -offset[i,j])
    # wenn keine periodischen Ketten, dann auf die urspruengliche Laenge kuerzen
    if not periodic:
        morphology = morphology[:,:,:box[orientation]]
    # Achsen wieder zuruecktauschen
    return np.transpose(morphology, axes=transpose)

def ChainOffset(box, chain_length, chain_offset, orientation):
    """
    Erstelle zufaellige Startpunkte der Ketten entlang der Richtung, in der sie
    ausgerichtet sind.

    box : 1D array
        Groesse der Simulationsbox.
    chain_length : int
        Laenge der Polymerketten.
    chain_offset : 1D list
        List of 2 elements each representing the displacement of the chains in
        each dimension. Allowed values 'random' or an integer number.
        E.g. ['random', 2] means random displacement in the first dimension (x)
        and a constant displacement of 2 between adjacent chains in the second
        dimension (y).
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)

    returns
    -------
    chain_offset : 2D array
        Array der Offsets fuer die Polymerketten. Index 0, 1: Koordinaten ohne
        der, in der die Ketten ausgerichtet sind, z.B. y und z fuer Ketten in
        x-Richtung
    """
    # for backwards compatibility
    if isinstance(chain_offset, (str, int)):
        chain_offset = [chain_offset] * 2
    # cyclic shifting of axes so that the last axis is the orientation of the chains
    size = box[np.roll(np.arange(3), 2 - orientation)]
    # random displacement in first dimension
    if chain_offset[0] == 'random':
        offset_0 = np.random.randint(chain_length, size=size[0])
    # fixed displacement
    else:
        offset_0 = np.arange(size[0]) * chain_offset[0]
    # random displacement in second dimension
    if chain_offset[1] == 'random':
        offset_1 = np.random.randint(chain_length, size=size[1])
    # fixed displacement
    else:
        offset_1 = np.arange(size[1]) * chain_offset[1]
    # entire offset
    offset = np.zeros(size[:2], dtype=np.int)
    # add up the offset in both dimensions
    offset += offset_0.reshape(-1, 1)
    offset += offset_1.reshape(1, -1)

    return offset

def composition_generator(chains, layer_params):
    """
    Generate the composition from the chain morphology.
    In the composition section the background material has to have the index 0.
    The material index for the chains has to be 1, additional materials to be
    used have to have indices 2 .. n.

    Returns
    -------
    composition (3D array): The array with the material indices corresponding to
    the chain morphology.
    """
    # extract the parameters
    params = layer_params['parameters'][0]
    if 'occurence' in params:
        occurence = params['occurence']
    else:
        occurence = None

    # fill the grid with the background material per default
    composition = np.zeros_like(chains)

    # simple case, only one material for all chains
    if occurence is None:
        # find the sites corresponding to chains
        chain_indices = np.where(chains >= 0)
        # create the composition
        composition[chain_indices] = 1
    else:
        # find the unique indices of the chains
        numbers = np.unique(chains)
        # matching list between chain index and material index
        mat_index = np.zeros(np.sum(occurence), dtype=np.int)
        # create the corresponding material indices list
        i = 0
        for n,j in zip(np.arange(1, len(occurence) + 1), np.cumsum(occurence)):
            mat_index[i:j] = n
            i = j
        # shuffle
        np.random.shuffle(mat_index)

        # assign the chain index to the material index
        for n,m in zip(numbers, mat_index):
            composition[np.where(chains == n)] = m

    return composition

def write_composition(mat_dist, layer_index, disorder):
    """
    Write the 3D composition array of a layer to file.
```

```
    The output filename should be:
        composition_{layer_index}_{boxid}.in

    Inputs:
        mat_dist (np 3Darray, int): array containing the materials indexes
        layer_index (int): index of the layer
        disorder (int): box index
    """
    filename = "composition_{0}_{1:03d}.in".format(layer_index, disorder)

    with open(filename, "w") as f:
        for k in range(mat_dist.shape[2]):
            for j in range(mat_dist.shape[1]):
                for i in range(mat_dist.shape[0]):
                    f.write("{} ".format(mat_dist[i, j, k]))
                f.write("\n")
            f.write("\n")

def write_chains(chains, layer_index, disorder):
    """
    Write the chain file.

    Input:
        chains (np 3D array, int): array with indices of the chains
        layer_index (int): index of the layer
        disorder (int): box inde

    output file:
        chains_{layer_index}_{boxid}.in
    """
    fn = 'chains_{0}_{1:03d}.in'.format(layer_index, disorder)

    with open(fn, 'w') as f:
        for k in range(chains.shape[2]):
            for j in range(chains.shape[1]):
                for i in range(chains.shape[0]):
                    f.write("{} ".format(chains[i, j, k]))
                f.write("\n")
            f.write("\n")

def morphology_generator(filename, layer_index, disorder):
    """
    Load the morphology from the yaml input file generated by Bumblebee
    and generate the composition for the layer

    Inputs:
        filename (string): name of the yaml file with input parameters
        layer_index (int): index of the layer
        disorder (int): box index
    """
    if disorder < 1:
        raise ValueError("disorder number must be > 0")
    if layer_index < 0:
        raise ValueError("layer index must be >= 0")

    # load layer parameters
    layer_params = yaml.load(open(filename, "r"),
Loader=yaml.FullLoader)["morphology"][layer_index]

    # seed the random generator
    np.random.seed((layer_index + 1) * disorder)

    # create the morphology of chains
    chains = chain_generator(layer_params)

    # generate the 3D distribution of materials in the layer
    lay_compo = composition_generator(chains, layer_params)

    # write the required composition files
    write_composition(lay_compo, layer_index, disorder)

    # write the chains file
    write_chains(chains, layer_index, disorder)

################################################################################
### DO NOT CHANGE HERE ###

# get input arguments
parser = argparse.ArgumentParser()
parser.add_argument('layer', type=int, help='input layer number')
parser.add_argument('disorder', type=int, help='input disorder number')
parser.add_argument('--input-file', type=str, default="morphology.yml", help='input
file name')
arg = parser.parse_args()

# generate composition
morphology_generator(arg.input_file, arg.layer, arg.disorder)
```

A.1.3. Skript zur Ereugung der diskreten Verteilung der Energien der Oligomere

Das Skript kann zusätzlich für die Erzeugung einer oligomeren Struktur verwendet werden. Dabei müssen die erhaltenen Häufigkeiten per Hand in die Simulationsparameter eingetragen werden, wobei darauf geachtet werden muss, dass die Gesamtzahl der Oligomere der Summe der Häufigkeiten der Energiewerte entspricht.

```
# -*- coding: utf-8 -*-
"""
Anzahl der Bins für eine quasi-gaußverteilte Energielandschaft berechnen.
Die gewünschten Simulationsparameter müssen eingetragen werden.
"""
import numpy as np
import matplotlib.pyplot as plt

# Boxgröße
Nx, Ny, Nz = 10, 180, 180
# Kettenlänge
chain_length = 180
# Zentrum der DOS
x0 = 0
# sigma in eV
sigma = 0.05
# Anzahl der Bins
N_bins = 9
# Bereich, den die Bins abdecken, +- um das Zentrum, Vielfache von sigma
bin_range = 2.576 # entspricht 1 % der Werte außerhalb

################################################################################

def gauss(x, x0, sigma):
    return np.exp(-(x - x0)**2 / 2 / sigma**2)

################################################################################

# Breite eines Bins
bin_width = sigma * bin_range * 2 / N_bins
# Mitten der Bins
bin_centers = np.linspace(-bin_range * sigma + x0 + bin_width / 2, \
                          bin_range * sigma + x0 - bin_width / 2, num=N_bins)
# Höhe der Bins
bin_heights = gauss(bin_centers, x0, sigma)

# Anzahl der Ketten
N_chains = np.prod((Nx, Ny, Nz)) / chain_length

# normieren auf die Anzahl der Ketten
frequency = np.array(bin_heights / np.sum(bin_heights) * N_chains, dtype=np.int)

# Energien mit den absoluten Häufigkeiten
print('index\tenergy\toccurence')
for i, center, height in zip(np.arange(1, N_bins + 1), bin_centers, frequency):
    print('{0}\t{1: .4f}\t{2:g}'.format(i, center, height))

if np.sum(frequency) == N_chains:
    print('distribution matches number of chains')
else:
    print('difference between distribution and number of chains', np.sum(frequency) -
N_chains)

print('list of occurence')
print('[{}]'.format(', '.join(frequency.astype(np.str))))
```

```
fig, ax = plt.subplots(figsize=(4.33, 3.31))

X = np.linspace(x0 / sigma - 5, x0 / sigma + 5, num=101)
ax.plot(X, gauss(X * sigma, x0 ,sigma), linewidth=1)
ax.bar(bin_centers / sigma, bin_heights, align='center', width=bin_width / sigma,
color='none', \
       linewidth=1, edgecolor='darkgrey')
ax.set_xlabel('Energie ($\sigma$)')
ax.set_ylabel('Häufigkeit (normiert)')
ax.tick_params(axis='both', direction='in', right=True, top=True)
```

A.2. Eigenentwickelte Simulationsprogramme

In den Kapiteln 5.1 und 5.2 habe ich eigene Programme zur Einteilchen- und Vielteilchensimulation von Ladungstransport in OFETs entwickelt. Diese habe ich in diesem Abschnitt aufgelistet. Neben den Simulationsprogrammen selbst sind auch Hilfsskripte zur Erzeugung der Parametersätze aufgeführt. Diese erstellen die nötigen Unterordner mit den gewünschten Parametern. Im Fall der Einteilchensimulation mit effektiver Energielandschaft (Kapitel 5.1) wird in den Parameter festgelegt, ob es sich um eine Simulation mit oder ohne Berücksichtigung der Coulombwechselwirkung handelt. Für die beiden Vielteilchensimulationen Teil 1 und 2 (Kapitel 5.2) existieren zwei separate Programme.

A.2.1. Simulationsprogramm zur Einteilchensimulation mit effektiver Energielandschaft

```
# -*- coding: utf-8 -*-
"""
Simulation von Ladungstransport durch OFET.

Charakteristische Merkmale:

    - Einteilchensimulation
    - Coulombwechselwirkung zwischen den festen Ladungsträgern ist möglich
    - Es kann eine Morphologie erstellt werden. Dabei werden Gitterpunkte zu Ketten
      zusammengefasst, die die gleiche Energie haben.
    - Sprünge innerhalb einer Kette haben eine eigene Kopplungsstärke.
    - Fallenzustände haben einen negativen Wert in der Morphologie.
    - Einzelne Gitterplätze können Fallenzuständen zugewiesen werden. Diese
      haben eine Energie aus einer Gaußverteilung, die unterhalb der der anderen
      Plätze liegt.
    - Orthorhombisches Gitter mit periodischen Randbedingungen in x- und y-Richtung.
    - Parallelisierung: bezüglich der Trials
    - Dateien für Energielandschaft und Morphologie können optional gespeichert
      und nach jedem Trial wieder gelöscht werden.
    - Anzahl der Prozesse wird als Parameter übergeben, damit ist jedes Cluster
      möglich.
    - Konsistenzcheck der Ladungsträgerdatei bei Fortsetzung einer vorherigen
      Simulation.
    - Option ob sich Fallenzustände nur an der Oberfläche oder überall
      im Halbleiter befinden sollen.
    - Bei der Besetzung der Plätze können auch nur die Fallenzustände besetzt
      werden.
    - Der Typ der Morphologie wird in der Parameterdatei bestimmt. Nur die benötigten
      Parameter werden geladen.
    - Positive und negative Gatespannung wird gleich behandelt, beide Male wie
      Elektronen.
    - Parallelisierung über mehrere Nodes mit mpi4py.
    - Erste Schicht mit ausschließlich Fallen möglich. Die Fallenschicht befindet
      sich nur im halben Abstand zur nächsten Schicht.
    - Trials sind reproduzierbar in den Zufallszahlen.
    - Der Zufallszahlengenerator für die Energielandschaft und die Ladungsträger
      können entkoppelt werden in der Parameterdatei. Bei Entkopplung wird immer
      die Energielandschaft von Trial 0 erzeugt.
    - Eine Morphologie kann geladen werden für alle Trials. Die Datei muss
      morphology.npy heißen und sich im Hauptordner befinden.
    - Anzahl der Simulationsschritte wurde als Abbruchkriterium festgelegt.
    - Vorherige Version übernimmt die Anzahl bisheriger Simulationsschritte.

Dateiausgabe:

    - Position und Zeitpunkt des Ladungsträgers nach bestimmter Anzahl an Sprüngen.
"""
import numpy as np
import numpy.random as rnd
import scipy.interpolate as interpolate
import random
import os
import itertools
import multiprocessing as mp
from mpi4py import MPI
import sys
import time
import pickle

###############################################################################
# spezielle Parameter im Programm
# -----------------------------------------------------------------------------

# Modus für Fehlersuche
debugmode = False
# Dateien für Morphologie und Energielandschaft speichern?
# Kann sinnvoll sein bei großen Dateien diese nicht zu speichern
save_data = True
# Morphologie und Energielandschaft nach Beendigung eines Trials löschen?
delete_data = True
# Sollen nur die Fallenzustände besetzt werden?
occupy_traps = False

###############################################################################
# Funktionendefinition
# -----------------------------------------------------------------------------

# -----------------------------------------------------------------------------
# Energielandschaft
#
def Energylandscape(box, sigma, morphology, model='GDM', gatevoltage=True):
    """
    Energielandschaft erstellen.

    box : 1D array
        Größe der Simulationsbox
    sigma : float
        gewünschte Standardabweichung der Energielandschaft (eV)
    morphology : 3D array
        Morphologie
    model : str
        Gewünschtes Modell. Mögliche Modelle: gaussian disorder model ('GDM'),
        correlated disorder model ('CDM').
    gatevoltage : bool
        Soll das Gatepotential zur Energielandschaft dazuaddiert werden?

    returns
    -------
    E : 3D array
        Energiewerte für jeden Gitterplatz nach gewünschtem Modell.
    """
    if model == 'GDM':
```

```
        E = GDM(box, sigma, morphology)
    elif model == 'CDM':
        E = CDM(box, sigma)
    else:
        raise ValueError('model for energylandscape unknown:', model)

    if gatevoltage:
        # Potential durch Gateelektrode dazuaddieren
        gate_potential = np.empty(box)
        gate_potential[:] = np.linspace(0, gate_potential_step*(box[2]-1), \
                                        num=box[2]).reshape(1,1,-1)
        # in der Fallenschicht einen halben Potentialsprung dazuaddieren
        if trap_layer:
            gate_potential[:,:,0] += gate_potential_step / 2
        E += gate_potential
    return E

def GDM(box, sigma, morphology):
    """
    Gaussian disorder model.

    Erstelle eine Energielandschaft mit statischen Unordnung. Die Energien der
    Gitterplätze sind einer Gaußverteilung entnommen. Morphologie wird
    berücksichtigt.

    box : 1D array
        Größe der Simulationsbox.
    sigma : float
        Standardabweichung der Verteilung der Energien.
    morphology : 3D array
        Morphologie

    returns
    -------
    E : 3D array
        Energielandschaft der Größe `box` nach dem GDM.
    """
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_inverse=True)
    # Energiewerte für jede Kettennummer
    E_reduced = rnd.normal(loc=0, scale=sigma, size=len(numbers))
    # rekonstruiere die Morphologie
    E = E_reduced[indices].reshape(box)
    # Positionen der Fallenzustände
    traps = np.where(morphology <= -1)
    # ------------------------------
    # Energiewerte für Fallenzustände
    #
    # Gaußverteilung
    if trap_distribution == 'gauss':
        E_traps = rnd.normal(loc=E_trap, scale=sigma_trap, size=len(traps[0]))
    elif trap_distribution == 'exp':
        E_traps = E_trap - rnd.exponential(scale=sigma_trap, size=len(traps[0]))
    elif trap_distribution == 'gaussexp':
        size_gauss = int(len(traps[0]) * np.sqrt(np.pi) / (stretch_exp * np.sqrt(2) +
np.sqrt(np.pi)))
        size_exp = len(traps[0]) - size_gauss
        gauss_traps = []
        while len(gauss_traps) < size_gauss:
            x = rnd.normal(loc=E_trap, scale=sigma_trap)
            if x >= E_trap:
                gauss_traps.append(x)
        E_traps = np.append(gauss_traps, E_trap -
rnd.exponential(scale=stretch_exp*sigma_trap, size=size_exp))
    else:
        raise ValueError('trap distribution unknown', trap_distribution)
    # Energiewerte den Plätzen zuweisen
    E[traps] = E_traps
    return E

def CDM(box, sigma, gridspacing):
    #TODO Morphologie für CDM berücksichtigen
    """
    Correlated disorder model.

    Create energy landscape with correlated disorder.
    Nach Phys Rev B 83, 085206 (2011), Bumblebee.

    box : 1D array
        size of simulation box
    sigma : float
        desired energetic width of the energy landscape (eV)
    gridspacing : 1D array
        lattice spacing (nm)

    returns
    -------
    E : 3D array
        Energielandschaft mit räumlicher Korrelation.
    """
    # array of the energy landscape
    E = np.empty(box, dtype=np.float64)
    # strength of the dipoles in order to achieve an energetic width of sigma
    dipole_strength = sigma * 8.854e-12 * epsilon_rel * np.min(gridspacing)**2 * 1e-9
/ (2.35 * 1.602e-19)
    # maybe take a unit dipole strength and rescale it at the end to the desired
    # sigma

    # does not work, morphology should be considered at some times
#    numbers, indices = np.unique(morphology, return_inverse=True)

    # randomly place dipoles on the grid
    dipoles = RandomDipoles(box, dipole_strength)

    # indices of a sphere with the cutoff radius for the dipole correlation
    indices = SphereIndices(cutoff_dipoles, gridspacing)

    # list of all sites
    x, y, z = np.mgrid[0:box[0], 0:box[1], 0:box[2]]
```

```
    pos_list = np.stack((x, y, z), axis=3).reshape(-1, 3)

    # go through all sites and calculate the dipole interaction energy
    for pos in pos_list:
        E[tuple(pos)] = np.sum( np.sum( dipoles[tuple(np.transpose((pos + indices) %
box))] \
             * indices * gridspacing, axis=1 ) / \
                np.sum((indices * gridspacing)**2, axis=1)**1.5 )

    # scale energy to eV
    E *= -1.602e-19 / (8.854e-12 * epsilon_rel * 1e-9)
    if debugmode:
        print('std of E', np.std(E))
    return E

def RandomDipoles(box, dipol_strength):
    """
    Fill the grid randomly with dipoles. All dipoles have the same strength, but
    the orientation is random.

    box : 1D array
        size of simulation box
    dipol_strength : float
        strength of the dipoles (in units of the elementary charge, nm)

    returns
    -------
    dipoles : 4D array
        Index 0: x-cooordinate, Index 1: y-coordinate, Index 2: z-cooordinate,
        Index 3: list, dipole orientation [x, y, z]
    """
    dipoles = np.empty((box[0], box[1], box[2], 3))
    phi = rnd.random(size=np.prod(box)) * 2 * np.pi
    theta = rnd.random(size=np.prod(box)) * np.pi
    dipoles[:,:,:,0] = np.reshape(dipol_strength * np.sin(theta) * np.cos(phi), box)
    dipoles[:,:,:,1] = np.reshape(dipol_strength * np.sin(theta) * np.sin(phi), box)
    dipoles[:,:,:,2] = np.reshape(dipol_strength * np.cos(theta), box)
    return dipoles

def SphereIndices(R, gridspacing, include_origin=False):
    """
    Determine the coordinates that are within a sphere with radius R. The center
    of the sphere is at [0,0,0] and can be included in the result or not. Also a
    non-cubic grid is possible.

    R : float
        radius of the sphere
    gridspacing : 1D array
        lattice spacing (nm) [dx, dy, dz]
    include_origin : bool
        include the origin [0,0,0] in the output?

    returns
    -------
    indices : 2D array
        List of coordinates that are within the sphere. Index 0: goes through the
        sites, Index 1: [x,y,z] coordinates
    """
    # Anzahl der Dimensionen
    dim = len(gridspacing)
    # Anzahl der maximal möglichen Gitterpunkte in einer Richtung
    n = int(np.max(R / gridspacing))
    indices = np.array(list(itertools.product(range(-n, n+1), repeat=dim)))
    length = np.sqrt(np.sum((indices * gridspacing)**2, axis=1))
    indices = indices[np.where(length <= R)]
    if not include_origin:
        indices = indices[np.any(np.not_equal(indices, [0] * dim), axis=1)]
    return indices

# ----------------------------------------------------------------------------
# Morphologie erstellen
#
def CreateMorphology(morphology_type):
    """
    Erstelle die Morphologie
    - Es können verschiedene Schichten aneinandergefügt werden.
    - Die Ausrichtung der Ketten kann in jeder Schicht individuell eingestellt
      werden.
    - Fallen können in verschiedenen Schichten hinzugefügt werden.

    box : 1D array
        Größe der Simulationsbox.
    morphology_type : str
        Art der zu erstellenden Morphologie
    returns
    -------
    morphology, eta_traps : 3D array, float
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Kette gehören.
        eta_traps ist die tatsächliche Fallendichte in der Schicht, in der
        Fallen vorhanden sind.
    """
    if morphology_type == 'crystallite':
        morphology, eta = CreateCrystallite(box, crystal_a, crystal_b, \
                                            crystal_gamma, number_seeds, boundary_gap)
    elif morphology_type == 'pointgrid':
        if interface_traps:
            # Trapstates nur in der ersten Schicht
            interface, eta = CreatePointgrid(np.array([box[0], box[1], 1]),
chain_length, chain_sigma, chain_offset, 0, trapdensity)
            # Restliche Morphologie ohne Fallen
            morphology, e = CreatePointgrid(np.array([box[0], box[1], box[2] - 1]),
chain_length, chain_sigma, chain_offset, 0)
            # Kettennummern eindeutig machen
            morphology += np.max(interface) + 1
            morphology = np.concatenate((interface, morphology), axis=2)
        else:
            morphology, eta = CreatePointgrid(box, chain_length, chain_sigma,
chain_offset, 0, trapdensity)
    # Enthält die unterste Schicht nur Fallen?
```

```
    if trap_layer:
        morphology[:,:,0][np.where(morphology[:,:,0] > 0)] = 0
    return morphology, eta

# ------------------------------------------------------------------------------

def CreateCrystallite(box, crystal_a, crystal_b, crystal_gamma, number_seeds,
boundary_gap=0):
    """
    box : 1D array
        Boxgröße
    crystal_a, crystal_b : float, float
        Kantenlängen der Kristallite (Parallelogramme) (nm)
    crystal_gamma : float
        Winkel der Kristallite (Parallelogramme) (°)
    number_seeds : int
        Anzahl der Startpunkte für Kristallitwachstum, werden auf einem
        quadratischen Gitter angeordnet.
    boundary_gap : float
        Dicke der Korngrenzen zwischen den Kristalliten

    returns
    -------
    morphology, eta_boundaries : 3D array, float
        Morphologie und Bruchteil der Korngrenzen in der Morphologie
    """
    # Winkel in Bogenmaß umrechnen
    crystal_gamma = crystal_gamma/180*np.pi

    # Abstand zwischen den Startpunkten für Kristallitwachstum
    dist = box[:2]/np.sqrt(number_seeds)
    # Startpunkte für Kristallitwachstum
    seeds = np.array(list(itertools.product(
            range(int(np.sqrt(number_seeds))), repeat=2))) * dist + dist/2
    # Zahlenwerte ganzzahlig machen
    seeds = np.array(seeds, dtype=np.int)

    # eine Schicht der Morhpologie
    layer = np.zeros(box[:2], dtype=np.int)
    # Indizes mit maximalem Abstand zu einem Punkt
    sphere = SphereIndices(boundary_gap, gridspacing[:2])
    # Ausrichtung der Kristallite
    seed_directions = np.random.random(size=number_seeds)*2*np.pi

    # pro Schleifendurchlauf ein Kristallwachstumsschritt
    # Schleifenvariable
    i = 0
    while True:
        # alte Morphologieschicht speichern
        old_layer = layer.copy()
        # jeder Kristallit wächst nacheinander
        for s,seed in enumerate(seeds):
            # Index des Kristallits
            crystallite = s + 1
            # Position des Wachstumspunkts
            sx, sy = seed
            # Winkel der Richtung des Kristallits
            phi = seed_directions[s]

            # meshgrid für alle Gitterpunkte
            x, y = np.meshgrid(np.arange(-box[0], 2*box[0]), \
                               np.arange(-box[1], 2*box[1]), indexing='ij')
            x = x.flatten()
            y = y.flatten()

            #TODO Spezialfall 90 grad
            # erstes Geradenpaar
            m = np.tan(phi)
            t = abs(i * crystal_b * np.sin(crystal_gamma) / np.sin(np.pi/2 + phi))
            x, y = BetweenLines(x, y, sx, sy, m, t)

            # zweites Geradenpaar
            m = np.tan(phi + crystal_gamma)
            t = abs(i * crystal_a * (np.cos(phi)*np.tan(phi + crystal_gamma) -
np.sin(phi)))
            x, y = BetweenLines(x, y, sx, sy, m, t)

            # alle noch unbesetzten Plätze
            mask = np.where(layer[tuple((x % box[0], y % box[1]))] == 0)
            x = x[mask]
            y = y[mask]

            # Dicke der Grain boundaries
            if boundary_gap > 0:
                # durchlaufe alle möglichen neuen Gitterplätze
                for new in zip(x,y):
                    # alle Punkte innerhalb eines Abstands um den neuen Punkt
                    p = np.transpose((new + sphere) % box[:2])
                    # teste, ob alle Plätze innerhalb einer Kugel mit Radius der Dicke
                    # des Abstands zwischen den Kristalliten entweder unbesetzt oder
                    # mit dem aktuellen Kristallit belegt sind
                    if np.all(np.any(np.stack((layer[tuple(p)] == 0, \
                                    layer[tuple(p)] == crystallite)), axis=0)):
                        layer[tuple(new % box[:2])] = crystallite
            else:
                layer[tuple((x % box[0], y % box[1]))] = crystallite
        # Wenn alle Plätze besetzt sind, bricht das Wachstum ab
        if np.all(old_layer == layer):
            break
        # Schleifenvariable erhöhen
        i += 1

    # Grain boundaries in Fallenzustände umwandeln
    # Bereich der Grain boundaries
    mask = np.where(layer == 0)
    eta_boundaries = len(mask[0]) / layer.size
    layer[mask] = np.arange(-1, -len(mask[0]) - 1, -1)
    # Morphologie in die dritte Dimension ausdehnen
    morphology = np.empty(box, dtype=np.int)
    morphology[:] = layer.reshape((*box[:2], 1))
```

```
    return morphology, eta_boundaries

def BetweenLines(x, y, sx, sy, m, t):
    """
    Gebe die Punkte zurück, die sich zwischen den Geraden
    y = m*x + t
    und
    y = m*x - t
    befinden.

    x, y : 1D arrays
        x- und y-Koordinaten der Punkte
    sx, sy :
        Punkt, um den die Geraden verschoben sind.
    m : float
        Steigung der Geraden
    t : float
        Achsenabschnitt
    """
    mask = np.where(y - sy <= m * (x - sx) + t)
    x = x[mask]
    y = y[mask]
    mask = np.where(y - sy >= m * (x - sx) - t)
    x = x[mask]
    y = y[mask]
    return x, y

# ----------------------------------------------------------------------------

def CreatePointgrid(box, chain_length, chain_sigma, chain_offset, orientation,
trapdensity=0., periodic=True):
    """
    Erstelle einen Bereich für die Morphologie, in dem alle Ketten in einer
    Richtung verlaufen.
    - Es werden Gitterpunkte zu Ketten verbunden. Dabei werden alle Gitterpunkte
      durchnummeriert, wobei Gitterpunkte einer Kette die gleiche Nummer erhalten.
    - Es muss nicht jede Zahl als Nummer einer Kette vorkommen.
    - Kann verwendet werden, um mehrere solcher Kästen aneinanderzureihen.
    - Optionalen Parameter, wenn Ketten periodisch sein sollen.
    - Fallenzustände haben negative Nummern, jeder seinen eigenen Wert.

    box : 1D array
        Größe der Morphologie
    chain_length : int
        Länge der Ketten
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)
    trapdensity : float
        Wahrscheinlichkeit, dass ein Gitterplatz ein Fallenzustand ist
    periodic : bool
        Sollen die Ketten über die Ränder periodisch fortgesetzt werden?

    returns
    -------
    morphology, eta_traps : 3D array, float
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Polymerkette gehören. Ein Wert von -1 bedeutet
        ein Fallenzustand.
        eta_traps ist der Bruchteil der Gitterplätze mit Fallen. Wenn keine
        vorhanden sind, wird None zurückgegeben.
    """
    box = np.asarray(box)
    # die Richtung, in der die Ketten durchnummeriert werden, ist die Achse
    # der Orientierung. Vertausche die Achsen so, dass die Achse der Kettenausrichtung
    # die letzte Achse ist.
    shape = box[np.roll(np.arange(3), 2 - orientation)]
    # Achsen, die wieder zurück vertauscht werden müssen
    transpose = np.roll(np.arange(3), orientation - 2)

    # wenn keine periodische Randbedingung für die Kette gelten soll, dann eine
    # Kette mehr mitnehmen
    if not periodic:
        shape[2] += chain_length

    # ------------------------------------------------------------------------
    # Polymerketten mit bestimmter Länge erstellen
    #
    # Array für die Morphologie
    morphology = np.zeros(shape, dtype=np.int)
    # Laufindex der Kettennumer
    k = 1
    # die zwei Dimensionen durchlaufen, die nicht der Kettenausrichtung entsprechen
    for i in range(shape[0]):
        for j in range(shape[1]):
            # ein Kettenstrang (1D)
            line = []
            # Füge solange Segmente hinzu, bis der Strang voll ist
            while True:
                # Länge der neuen Ketten aus Verteilung bestimmen
                l = int(rnd.normal(loc=chain_length, scale=chain_sigma) + 0.5)
                # kann diese Kettenlänge noch hinzugefügt werden?
                if len(line) + l <= shape[2]:
                    line.extend([k]*l)
                    # Kettenindex erhöhen
                    k += 1
                else:
                    line.extend([k]*(shape[2] - len(line)))
                    # Strang zur Morphologie hinzufügen
                    morphology[i,j] = line
                    # Kettenindex erhöhen
                    k += 1
                    break

    # ------------------------------------------------------------------------
    # Fallenzustände einfügen
    #
```

```
    # Laufindex für die Fallenzustände
    trap = -1
    # Alle Gitterplätze werden durchlaufen
    for i in range(shape[0]):
        for j in range(shape[1]):
            for k in range(box[orientation]):
                # zufällig eine Falle platzieren
                if rnd.random() < trapdensity:
                    # Wenn eine Kette auseinandergerissen werden würde,
                    # wird die Falle nach hinten verschoben
                    x = k
                    while morphology[i, j, x-1] == morphology[i, j, x]:
                        x += 1
                        if x >= box[orientation]:
                            break
                    # Falle einbauen
                    if x < box[orientation]:
                        morphology[i, j] = np.append(morphology[i, j, :x], \
                                  np.append(trap, morphology[i, j, x:-1]))
                        # Laufindex für Fallen erniedrigen
                        trap -= 1
    # Versatz der Ketten nebeneinander
    offset = ChainOffset(box, chain_length, chain_offset, orientation)
    # Verschiebe die Kettennummern entsprechend dem Versatz
    for i in range(shape[0]):
        for j in range(shape[1]):
            morphology[i,j,:] = np.roll(morphology[i,j,:], -offset[i,j])
    # wenn keine periodischen Ketten, dann auf die ursprüngliche Länge kürzen
    if not periodic:
        morphology = morphology[:,:,:box[orientation]]
    # mittlere Kettenlänge bestimmen
    numbers, counts = np.unique(morphology, return_counts=True)
    chainlength_mean = np.average(counts[np.where(numbers > 0)])
    print('average chain length', '{:.3g}'.format(chainlength_mean), '+-', \
          '{:.3g}'.format(np.std(counts[np.where(numbers > 0)])))
    # Anzahl der Fallen
    counts = np.where(morphology < 0)
    # Fallendichte im Bereich
    if len(counts[0]) > 0:
        eta_traps = len(counts[0])/np.prod(box)
    else:
        eta_traps = None
    # Achsen wieder zurücktauschen
    return np.transpose(morphology, axes=transpose), (chainlength_mean, eta_traps)

def ChainOffset(box, chain_length, chain_offset, orientation):
    """
    Erstelle zufällige Startpunkte der Ketten entlang der Richtung, in der sie
    ausgerichtet sind.

    box : 1D array
        Größe der Simulationsbox.
    chain_length : int
        Länge der Polymerketten.
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)

    returns
    -------
    chain_offset : 2D array
        Array der Offsets für die Polymerketten. Index 0, 1: Koordinaten ohne
        der, in der die Ketten ausgerichtet sind, z.B. y und z für Ketten in
        x-Richtung
    """
    # zyklisches Vertauschen der Dimensionen, sodass orientation die letzte Achse ist
    size = box[np.roll(np.arange(3), 2 - orientation)]
    if chain_offset == 'random':
        return rnd.randint(chain_length, size=size[:-1])
    # Ketten werden in die beiden übrigen Richtungen um festen Wert jeweils zueinander
verschoben
    else:
        offset0 = np.repeat(np.arange(size[1]).reshape((1,-1)), size[0], axis=0) *
chain_offset
        offset1 = np.repeat(np.arange(size[0]).reshape((-1,1)), size[1], axis=1) *
chain_offset
        return offset0 + offset1

# ----------------------------------------------------------------------------
# Sprungraten berechnen
#
def CalculateRates(E, morphology, occupied):
    """
    Berechnet die Sprungraten an jeder Position für jede Sprungrichtung.
    Die Morphologie wird berücksichtigt.

    E : 3D array
        Energielandschaft
    morphology : 3D array
        Morphologie
    occupied : 2D array
        Array mit den besetzten Gitterplätzen

    returns
    -------
    k : 4D array
        Array mit den Sprungraten
        Index 0: Sprungrichtung, Index 1: x-Koordinate, Index 2: y-Koordinate,
        Index 3: z-Koordinate
    """
    k = np.empty((number_jumps, *box))
    for j,jum in enumerate(jumps):
        # alle Sprungraten für eine Sprungrichtung berechnen
        k[j] = RatesPerJump(E, jum, morphology)
        # Sprünge auf besetzte Gitterplätze sind nicht möglich
        if not len(occupied) == 0:
            oldpos = (occupied - jum) % box
```

```
            k[j, oldpos[:,0], oldpos[:,1], oldpos[:,2]] = 0
        # Sprünge wegen Beschränkung in z-Richtung ausschließen
        if jum[2] > 0:
            k[j,:,:,-jum[2]:] = 0
        elif jum[2] < 0:
            k[j,:,:,:-jum[2]] = 0
    return k

def RatesPerJump(E, jump, morphology):
    """
    Sprungraten für eine Sprungrichtung berechnen. Berücksichtigt Sprünge mit
    unterschiedlichen Gitterabständen und inversen Lokalisierungskonstanten.
    Sprung in x-Richtung mit gamma_para, in y-,z-Richtung mit gamma_perp.
    Sprünge innerhalb einer Kette mit gamma_intra.

    E : 3D array
        Energielandschaft
    jump : 1D array
        Sprungrichtung
    morphology : 3D array
        Morphologie

    returns
    -------
    k : 3D array
        Sprungraten an jeder Position in eine Sprungrichtung.
    """
    # Zielarray der Energiewerte
    E_ziel = np.array(E)
    # Zielarray der Morphologie
    morphology_final = np.array(morphology)
    # Verschiebe die Zielarrays entgegen der Sprungrichtung
    for axis,i in enumerate(jump):
        E_ziel = np.roll(E_ziel, -i, axis=axis)
        morphology_final = np.roll(morphology_final, -i, axis=axis)
    # Potential bei Sprüngen in x-Richtung dazuaddieren
    E_ziel += voltage_difference*jump[0]
    # ------------------------------------------------------------------------
    # Normale Sprünge
    #
    r_gamma = np.sqrt(np.sum((jump*gridspacing*[gamma_x, gamma_y, gamma_z])**2))
    k = Hoppingrate(r_gamma, E, E_ziel)
    # ------------------------------------------------------------------------
    # Sprünge innerhalb der Kette
    #
    # welcher Sprung findet innerhalb einer Kette statt
    same_chain = np.where(morphology == morphology_final)
    r_gamma = np.sqrt(np.sum((jump*gridspacing*gamma_intra)**2))
    k[same_chain] = Hoppingrate(r_gamma, E[same_chain], E_ziel[same_chain])
    # ------------------------------------------------------------------------
    # Sprünge zur Fallenschicht, wenn vorhanden
    #
    if trap_layer:
        # Annahme: Sprünge von und zu Fallen finden mit der Schicht darüber statt
        if jump[2] == 1:
            r_gamma = np.sqrt(np.sum((jump*gridspacing*[gamma_x, gamma_y,
gamma_z/2])**2))
            k[:,:,0] = Hoppingrate(r_gamma, E[:,:,0], E_ziel[:,:,0])
        elif jump[2] == -1:
            r_gamma = np.sqrt(np.sum((jump*gridspacing*[gamma_x, gamma_y,
gamma_z/2])**2))
            k[:,:,1] = Hoppingrate(r_gamma, E[:,:,1], E_ziel[:,:,1])
    # ------------------------------------------------------------------------
    # leere Plätze in der Morphologie
    #
    empty_sites = np.where(morphology_final == 0)
    k[empty_sites] = 0
    return k

def Hoppingrate(r_gamma, E_i, E_f):
    """
    Berechne Hoppingrate nach Miller-Abrahams (Tunnelprozess + thermische
    Aktivierung). Funktioniert für einzelne Energie als auch Array von Energien.

    r_gamma : float
        Produkt aus Abstand zwischen den Gitterplätzen und der inversen Lokalisierung
    E_i : 3D array
        Energie des Startgitterplatz
    E_f : 3D array
        Energie des Zielgitterplatz
    """
    if hopping == 'MA':
        thermal = np.full_like(E_i, 1.)
        mask = np.where(E_i < E_f)
        thermal[mask] = np.exp((E_i[mask] - E_f[mask])/kT)
        return nu0 * np.exp(-2 * r_gamma) * thermal
    else:
        return C1_marcus * np.exp(-2 * r_gamma) * np.exp(-(E_f - E_i +
reorg_energy)**2 / C2_marcus)

# ----------------------------------------------------------------------------

def FillSites(E, morphology, N):
    """
    Fülle die Gitterplätze mit Ladungsträger einen nach dem anderen ohne
    Coulombwechselwirkung! Jede Kette wird nur einmal besetzt.

    E : 3D array
        Array der Energien der Gitterplätze
    morphology : 3D array
        Morphologie
    N : int
        Anzahl der zu besetzenden Plätze mit Ladungsträgern. Entspricht der Anzahl
        der zu besetzenden Polymerketten.

    returns
    -------
    occupied_sites : 2D array
        Array der besetzten Gitterplätze, sortiert nach aufsteigender Energie.
        Alle Plätze einer Kette werden besetzt.
```

```
    """
    # Array für die besetzten Gitterplätze
    occupied_sites = np.empty((0, 3), dtype=np.int)
    # überprüfe, ob N > 0
    if not N > 0:
        return occupied_sites
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_index=True)
    # reduziere das Energiearray
    E_reduced = E[np.unravel_index(indices, box)]
    # finde die Positionen der Kettennummern mit den niedrigsten Energien in der
Morphologie
    for n in numbers[np.argsort(E_reduced)[:N]]:
        occupied_sites = np.append(occupied_sites, np.transpose(np.where(n ==
morphology)), axis=0)

    return occupied_sites

def FillSitesWFermi(E, morphology, e_Fermi):
    """
    Fülle die Gitterplätze mit Ladungsträgern bis zur Fermienergie auf.
    Keine Coulombwechselwirkung zwischen den Ladungsträgern wird berücksichtigt.
    Jede Kette wird nur einmal besetzt.
    Fallenzustände werden behandelt, dadurch dass jede Falle eine eindeutige
    Nummer hat.

    E : 3D array
        Array der Energien der Gitterplätze
    morphology : 3D array
        Morphologie
    e_Fermi : float
        Fermienergie

    returns
    -------
    occupied_sites, number_carriers : 2D array, int
        Array der besetzten Gitterplätze, sortiert nach aufsteigender Energie.
        Alle Plätze einer Kette werden besetzt.
    """
    # prüfe, ob die Fermienergie nicht nan ist
    if e_Fermi is np.nan:
        return np.empty((0, 3), dtype=np.int), 0
    # Array für die besetzten Gitterplätze
    occupied_sites = np.empty((0, 3), dtype=np.int)
    # Boxgröße
    box = E.shape
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_index=True)
    # Gitterplätze, die besetzt werden sollen
    if occupy_traps:
        # nur Fallenzustände
        sites = np.where(numbers < 0)
    else:
        # alle Zustände möglich (außer null)
        sites = np.where(numbers != 0)
    # besetzbare Ketten
    numbers = numbers[sites]
    indices = indices[sites]
    # reduziere das Energiearray
    E_reduced = E[np.unravel_index(indices, box)]
    # Besetzungswahrscheinlichkeit für jeden Gitterplatz
    probability = 1 / (1 + np.exp((E_reduced - e_Fermi)/kT))
    # Zufallszahlen für die Besetzung der Gitterplätze
    occupation = rnd.random(size=E_reduced.size)
    # Nummern der besetzten Plätze
    occupied_numbers = numbers[np.where(probability > occupation)]
    if debugmode:
        print ('number of chains/sites', len(numbers))
    # tatsächliche Anzahl der Ladungsträger
    number_carriers = len(occupied_numbers)
    if debugmode:
        print('actual number of carriers', number_carriers)
    # finde die Positionen der Kettennummern mit den niedrigsten Energien in der
Morphologie
    for n in occupied_numbers:
        # alle Plätze einer Kette besetzen
        occupied_sites = np.append(occupied_sites, np.transpose(np.where(n ==
morphology)), axis=0)
    return occupied_sites, number_carriers

def FillSitesWCoulomb(E, N):
    #!!! funktioniert nicht mit Morphologie
    """
    Fülle die Gitterplätze mit Ladungsträger einen nach dem anderen.
    Nach jedem Platzieren wird die Coulombenergie neu berechnet. Diese wird
    bei der Ermittlung des energetisch niedrigsten Platzes mit berücksichtigt.

    E : 3D array
        Array der Energien der Gitterplätze
    N : int
        Anzahl der zu füllenden Plätze mit Ladungsträgern

    returns
    -------
    (potential, occupied_sites) : (2D array, 2D array)
        Tuple aus Array des Potentials und Array der besetzten Gitterplätze,
        sortiert nach aufsteigender Energie
    """
    # single carrier potential berechnen
    V_single = SingleCarrierPotential()
    # erzeuge ein Array mit Maskierungsfunktion für bereits besetzte Gitterplätze
    E_m = np.ma.array(E)
    # Array der besetzten Plätze
    occupied = np.empty((0, 3), dtype=np.int)
    # Array für Potential
    potential = np.zeros(box)
    for i in range(N):
        # Platz mit niedrigster Energie
        carrier = np.unravel_index(np.argmin(E_m + potential), box)
        # Gitterplatz maskieren
```

```
        E_m[carrier] = np.ma.masked
        # Position zum Array dazufügen
        occupied = np.vstack((occupied, carrier))
        # Potential durch den neuen Ladungsträger dazuaddieren
        potential += V_single[box[0] - carrier[0] : 2*box[0] - carrier[0], \
                              box[1] - carrier[1] : 2*box[1] - carrier[1], \
                              box[2] - 1 - carrier[2] : 2*box[2] - 1 - carrier[2]]
    # setze das Potential an den Stellen der Ladungsträger null
    potential[tuple(occupied.transpose())] = 0

    return occupied, potential

def SingleCarrierPotential():
    """
    Berechne das Potential eines Ladungsträgers mit periodischer Fortsetzung.
    Der Ladungsträger wird einmal in beide x-Richtungen und einmal in beide
    y-Richtungen kopiert, sowie in die vier diagonalen Ecken. Das Potential
    wird an jedem Punkt berechnet, wobei immer der Abstand zur nächstliegenden
    Ladung genommen wird. Der Ladungsträger erzeugt somit an keinem Punkt
    mehrmals ein Potential durch periodische Kopien.

    returns
    -------
    : 3D array
        Potential an jeder Stelle, shape = (2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1)
    """
    # Doppelte Boxgröße
    x2, y2, z2 = np.meshgrid(np.arange(2*box[0] + 1), np.arange(2*box[1] + 1),
np.arange(2*box[2] - 1), \
                         indexing='ij')
    # Potential
    V_single = np.zeros((2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1))

    # Liste mit den Positionen der Ladungsträger (periodisch)
    carrier_list = np.array([[0, 0, box[2] - 1], [box[0], 0, box[2] - 1], [box[0] * 2,
0, box[2] - 1],
                     [0, box[1], box[2] - 1], [box[0], box[1], box[2] - 1], [box[0] *
2, box[1], box[2] - 1],
                     [0, box[1] * 2, box[2] - 1], [box[0], box[1] * 2, box[2] - 1],
[box[0] * 2, box[1] * 2, box[2] - 1]])
    # berechne das Potential eines jeden Ladungsträgers an der Stelle, an der es
größer als der bisherige Wert ist
    for carrier in carrier_list:
        # Potential eines Ladungsträgers in der gesamten Box
        with np.errstate(divide='ignore'):
            V = faktor_Coulomb / np.sqrt(
                ((x2 - carrier[0]) * gridspacing[0])**2 + ((y2 - carrier[1]) *
gridspacing[1])**2 +
                ((z2 - carrier[2]) * gridspacing[2])**2)
        # Bereich, in dem das neue Potential größer ist als das bisherige
        rng = V > V_single
        V_single[rng] = V[rng]
    # setze das Potential an den Stellen der Ladungsträger null
    V_single[tuple(carrier_list.transpose())] = 0
    return V_single

def CalculatePotential(V_single, carriers):
    """
    Berechne das Coulomb-Potential an jeder Position, das durch die Ladungsträger
    verursacht wird.

    V_single : 3D array
        Potential eines Ladungsträgers, inklusive optionaler periodischer Fortsetzung
        shape = (2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1)
    carriers : 2D array
        Array mit den Positionen der Ladungsträger.

    returns
    -------
    : float
        Coulomb-Potential in eV.
    """
    V = np.zeros(box)
    for c in carriers:
        V += V_single[box[0] - c[0] : 2*box[0] - c[0], \
                      box[1] - c[1] : 2*box[1] - c[1], \
                      box[2] - 1 - c[2] : 2*box[2] - 1 - c[2]]
    return V

def Fermienergie(E, morphology, N):
    #TODO funktioniert nicht bei gleichzeitig Ketten und Fallen
    """
    Fermienergie berechnen. Als Zustandsdichte wird die Energielandschaft
    verwendet. Die thermische Aufweichung der Fermikante wird berücksichtigt.

    E : 3D array
        Array der Energien der Gitterplätze
    morphology : 3D array
        Morphologie
    N : int
        Anzahl der zu besetzenden Gitterplätze bzw. Ketten mit Ladungsträgern.

    returns
    -------
    e_Fermi : float
        Fermienergie
    """
    box = E.shape
    # prüfe, ob N positiv ist
    if N < 0:
        N = 0
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_index=True)
    if occupy_traps:
        # nur Fallenzustände
        sites = np.where(numbers < 0)
    else:
        # alle Zustände möglich (außer null)
        sites = np.where(numbers != 0)
```

```
    # wenn keine Zustände besetzt werden können, dann Abbruch
    if len(sites[0]) == 0:
        return np.nan
    # reduziere das Energiearray
    E_reduced = E[np.unravel_index(indices[sites], box)]
    # Bruchteil der besetzten Gitterplätze/Ketten
    eta = N / E_reduced.size
    # ----------------------------------------------------------------------------
    # Startwert für die Nullstellensuche, grobe Schätzung wo die Fermienergie liegt
    start, step = Startvalue(temperature, eta)
    # Schleifenvariable
    i = 0
    # sonst wird das Programm nicht fertig
    while True:
        # Fermienergie bestimmen
        e_Fermi = Secant(start, step, Nullstellenfunktion, args=(E_reduced, eta))
        if np.isnan(e_Fermi):
            # Schleifenvariable erhöhen
            i += 1
            # Startbedingungen für die Nullstellensuche ändern
            if i >= 100:
                return np.nan
            else:
                start -= 0.01
        else:
            break
    return e_Fermi

def Startvalue(T, eta):
    """
    Startwert für die Nullstellensuche, grobe Schätzung wo die Fermienergie liegt.

    T : float
        Temperatur
    eta : float
        Anteil der besetzten Plätze

    returns
    -------
    start, step
        Startwert und Schrittweite
    """
    # Stützpunkte für die Temperatur
    temps = [150, 500]
    if trapdensity == 0:
        c = [1e-3, 1e-2]
        fp = [ [-3*sigma, -3*sigma],
               [-sigma, -sigma] ]
    elif interface_traps:
        # Stützpunkte für die Bestimmung der Fermienergie
        den = trapdensity / z_length
        c = [0.3*den, den, 1.1*den]
        fp = [ [E_trap - sigma_trap, E_trap - sigma_trap],
               [E_trap / 2, E_trap / 2],
               [-2*sigma, -2*sigma] ]
    # Fallen überall im Halbleiter
    else:
        # Stützpunkte für die Bestimmung der Fermienergie
        den = trapdensity
        c = [0.3*den, den, 1.1*den]
        fp = [ [E_trap - sigma_trap, E_trap - sigma_trap],
               [E_trap / 2, E_trap / 2],
               [-2*sigma, -2*sigma] ]
    # Startwert durch Interpolation der besetzten Plätze auf die Stützpunkte
    start = interpolate.interp2d(temps, c, fp)(T, eta)[0]
    if debugmode:
        print('start value for fermi energy', start)
    # Schrittweite für die Nullstellensuche
    step = 0.1
    return start, step

def Secant(x0, dx0, func, tol=1e-8, stop=100, args=()):
    """
    Sekanten-Methode zur Nullstellensuche. Entspricht Newton-Methode, nur dass
    die erste Ableitung der Funktion numerisch berechnet wird.

    x_k+1 = x_k - (x_k - x_k-1)*f(x_k)/(f(x_k) - f(x_k-1))

    x0 : float
        Startwert für die Nullstelle
    dx0 : float
        Abstand zum Startwert, der für die Berechnung der Ableitung am Anfang \
        verwendet werden soll.
    func : func
        Funktion, deren Nullstelle gesucht werden soll, erstes Argument dieser
        Funktion ist die Größe, die bestimmt werden soll
    tol : float
        Toleranzwert für eine Nullstelle, wenn |f(x)| < tol
    stop : int
        Abbruch der Suche nach dieser Anzahl an Schritten
    args : tuple
        optimale Argumente, die an die Funktion func übergeben werden sollen

    returns
    -------
    x : float
        Wert der Nullstelle
    """
    def f(x):
        """
        Funktion mit übergebenen Argumenten
        """
        return func(x, *args)

    x = x0
    x1 = x0 + dx0
    dx = f(x)
    k = 0
    while abs(dx) > tol and k < stop:
        fx = f(x)
```

```
        x_new = x - (x - x1)*fx/(fx - f(x1))
        x1 = x
        x = x_new
        dx = fx
        k += 1
    if k == stop:
        return np.nan
    return x

def Nullstellenfunktion(eF, E_reduced, eta):
    """
    Funktion der Nullstellensuche für die Berechnung der Fermienergie.

    eF : float
        Fermienergie, soll bestimmt werden durch Nullstellensuche.
    E_reduced : 3D array
        Energielandschaft, reduziert um gleiche Energiewerte innerhalb der
        Ketten.
    eta : float
        Bruchteil der besetzten Gitterplätze.

    returns
    -------
    np.sum( f(E_reduced, e_Fermi) ) / E_reduced.size - eta
    """
    # Anzahl der Energiewerte
    N = E_reduced.size
    return np.sum(1 / (1 + np.exp((E_reduced - eF)/kT))) / N - eta

# ----------------------------------------------------------------------------

def PlaceCarrier(box, morphology, occupied, x=None, y=None, z=None):
    """
    Ein Gitterplatz wird zufällig mit einem Ladungsträger gefüllt.

    occupied : 2D array
        Array der besetzten Gitterplätze
    box : 1D array
        Array der Boxgröße
    morphology : 3D array
        Array der Morphologie
    x,y,z = None : int
        Wenn angegeben ist, dann wird diese Koordinate festgelegt

    returns
    -------
    p : 1D array
        Array mit neu besetztem Gitterplatz
    """
    # Dimensionen
    dim = len(box)
    # zufällige Wahl eines Startpunktes des Ladungsträgers unter Berücksichtigung
    # leerer Plätze in der Morphologie
    while True:
        # zufällige Position
        p = np.array(rnd.random(dim)*box, dtype=np.int)
        # setze eine Koordinate fest, wenn gewünscht
        if x != None:
            p[0] = x
        if y != None:
            p[1] = y
        if z != None:
            p[2] = z
        # Morphologie darf an dieser Stelle nicht 0 sein und nicht bereits
        # von einem anderen Ladungsträger besetzt sein
        if morphology[tuple(p)] != 0 and \
            not np.any(np.all(p == occupied, axis=1)):
            break
    return p

def PossibleJumps(NN, dim):
    """
    Ermittle mögliche Sprungrichtungen, berücksichtigt unterschiedliche
    Gitterabstände.

    NN : int
        Anzahl nächster Nachbarn, bei ungleichen Gitterabständen sollte diese
        höher eingestellt werden.
    dim : int
        Anzahl der Dimensionen

    returns
    -------
    2D array
        Array mit den möglichen Sprungrichtungen, sortiert nach steigendem
Sprungabstand
    """
    all_jumps = np.array(list(itertools.product(range(-NN,NN+1), repeat=dim)))
    jump_length = np.sqrt(np.sum(np.square(all_jumps*gridspacing), axis=1))
    jump_length_unique, counts = np.unique(jump_length, return_counts=True)
    return all_jumps[np.argsort(jump_length)[1:np.sum(counts[1:NN+1])+1]]

# ----------------------------------------------------------------------------

def LoadParameters(file):
    """
    Lade den Parametersatz aus einer Datei. Wandelt die Werte wenn möglich in
    float um.

    file : str
        Name der Parameterdatei

    returns
    -------
    parameter : dict
        Ein dictionary der Parameter
    """
    key = []
    value = []
    # Datei einlesen
```

```
    with open(file, 'r') as f:
        lines = f.read().split('\n')
    # Spalten den Namen und Werten zuordnen
    for l in lines:
        if not l == '':
            k, v = l.split('\t')
            key.append(k)
            if v == 'True':
                value.append(True)
            elif v == 'False':
                value.append(False)
            else:
                # Werte wenn möglich in Zahlen umwandeln
                try:
                    value.append(float(v))
                except:
                    value.append(v)
    return dict(zip(key, value))

def ConvertBool(value):
    #!!! sollte nicht mehr benötigt werden
    """
    Konvertiere einen True/False Parameterwert nach bool in der Parameterdatei.

    value : str
        String, der nach True/False konvertiert werden soll.

    returns
    -------
    value : bool
        True oder False
    """
    if value == 'True':
        return True
    elif value == 'False':
        return False
    else:
        raise ValueError('Could not convert', value, 'to bool')

def Initialize(trial):
    """
    Zuerst wird gesucht, ob eine frühere Simulation fortgesetzt werden kann.
    Das ist der Fall, wenn alle Dateien für Ladungsträger, Energielandschaft,
    Morphologie und besetzte Zustände gleichzeitig existieren und lesbar sind.
    Ansonsten wird alles neu erstellt.

    trial : int
        Nummer des Trials

    returns
    -------
    global_time, a, carrier_final, morphology, E, occupied_sites

    Wenn die Simulation bereits fertig ist, sind alle Werte `None`.
    """
    # ----------------------------------------------------------------------
    # alle Dateinamen
    #
    # Dateiname für die Position des Ladungsträgers
    fn_car = os.path.join(folder_out, 'carrier_trial' + str(trial) + '.asc')
    # Dateinamen, unter dem die Morphologie gespeichert wird
    fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) + '.npy')
    # Dateiname der Morphologie, falls diese aus dem Hauptordner geladen werden
    # soll
    fn_morph_all = 'morphology.npy'
    # Dateiname, unter dem die Energielandschaft gespeichert wird
    fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
    # Dateiname für die besetzten Zustände
    fn_occu = os.path.join(folder_out, 'occupation_trial' + str(trial) + '.npy')
    # Dateiname für den Zustand des Zufallszahlengenerators
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')
    # prinzipiell erstmal keine Dateien laden
    load_files = False

    # ----------------------------------------------------------------------
    # Prüfe, ob die Simulation bereits beendet wurde
    # Ladungsträger und Fermienergie laden
    #
    # prüfe ob die Ladungsträger einer vorhergehenden Simulation vorhanden sind
    if os.path.exists(fn_car):
        # Datei einlesen
        with open(fn_car, 'r', encoding='utf-8') as f:
            data = f.read().split('\n')
        # Versuche aus der Datei alle Zeitpunkte und die letzte Position
        # zu bestimmen. Wenn dabei ein Fehler auftritt, wird der Trial neu
        # gestartet.
        try:
            # alle Zeitpunkte einlesen
            times = [ float(line.split()[-1]) for line in data if line.startswith('#
time')]
            # letzte Position des Ladungsträgers
            carrier_final = np.array(data[-4:-1], dtype=np.int)
        except:
            if debugmode:
                print('could not load previous data in trial', trial)
        else:
            # bestimme die Anzahl der bisherigen Simulationsschritte
            a = int((len(times) - 1) * position_steps)
            if debugmode:
                print('number of completed simulation steps', a)
            # überprüfe, ob der Trial bereits beendet wurde
            if a >= simulation_steps:
                return (None, None, None, None, None, None)
            # Fermienergie aus zweiter Zeile einlesen
            e_Fermi = float(data[1].split()[-1])
            if debugmode:
                print('read fermi energy from previous simulation', e_Fermi)
            # Dateien nur laden, wenn alle existieren
            if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                os.path.exists(fn_occu) and os.path.exists(fn_rand):
```

```
            load_files = True
            if debugmode:
                print('previous data will be loaded')
                print('previous carrier list loaded')

# ------------------------------------------------------------------------
# Zufallsgenerator initialisieren
#
# Zustand des Generators laden, wenn vorhanden
if os.path.exists(fn_rand) and load_files:
    # lade den Zustand aus der Datei
    with open(fn_rand, 'rb') as f:
        random_state = pickle.load(f)
    # setze den Zustand
    rnd.set_state(random_state)
    if debugmode:
        print('previous random number generator state read')
# Generator erstmalig initialisieren mit Trialnummer
else:
    # zufällige Energielandschaft, oder die von Trial 0?
    if random_landscape:
        rnd.seed(trial)
        # prüfe Sinnhaftigkeit bei load_morphology, kann bei Bedarf auch weg
        if load_morphology:
            raise ValueError('morphology fixed, but energies random???')
    else:
        rnd.seed(0)
    if debugmode:
        print('random number generator freshly initialized')

# ------------------------------------------------------------------------
# Morphologie initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_morph) and load_files:
    # lade vorhergehende Morphologie
    morphology = np.load(fn_morph)
    if debugmode:
        print('previous morphology loaded')
# eine Morphologie für alle Trials und Parameter?
elif load_morphology:
    # lade die Morphologie aus dem Hauptordner
    morphology = np.load(fn_morph_all)
    # kopiere sie in den Ordner für den jeweiligen Parameter
    if save_data:
        np.save(fn_morph, morphology)
    # Anteil der Fallen berechnen
    if interface_traps:
        eta_traps = len(np.where(morphology < 0)[0]) / np.prod(box[:2])
    else:
        eta_traps = len(np.where(morphology < 0)[0]) / np.prod(box)
    # mittlere Kettenlänge bestimmen
    numbers, counts = np.unique(morphology, return_counts=True)
    chainlength_mean = np.average(counts[np.where(numbers > 0)])
    eta = (chainlength_mean, eta_traps)
    if debugmode:
        print('morphology loaded from main folder')
# Morphologie neu erstellen
else:
    if debugmode:
        print('new morphology')
    # Morphologie erstellen
    morphology, eta = CreateMorphology(morphology_type)
    # Morphologie speichern
    if save_data:
        np.save(fn_morph, morphology)

# ------------------------------------------------------------------------
# Energielandschaft initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_E) and load_files:
    # lade vorhergehende Energielandschaft
    E = np.load(fn_E)
    if debugmode:
        print('previous energy landscape loaded')
# Energielandschaft neu berechnen
else:
    if debugmode:
        print('new energy landscape')
    s = time.perf_counter()
    # Energielandschaft neu mit Morphologie berechnen
    E = Energylandscape(box, sigma, morphology, model=energy_model)
    e = time.perf_counter()
    if debugmode:
        print('time for energy landscape', e - s)
    # Energielandschaft speichern
    if save_data:
        np.save(fn_E, E)

# ------------------------------------------------------------------------
# besetzte Zustände
#
# überprüfe, ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_occu) and load_files:
    # lade vorherige Besetzung
    occupied_sites = np.load(fn_occu)
    # wenn das Coulombpotential beim Auffüllen berücksichtigt werden soll
    if fill_with_coulomb:
        # Potential eines Ladungsträgers
        V_single = SingleCarrierPotential()
        # berechne das Potential der Ladungsträger
        potential = CalculatePotential(V_single, occupied_sites)
        # addiere das Potential auf die Energielandschaft
        E += potential
    if debugmode:
        print('previous occupied sites loaded')
# Zustände neu füllen
else:
    if debugmode:
```

```
        print('new occupied sites')
    # Anzahl der besetzten Ketten/Gitterplätze
    if morphology_type == 'pointgrid':
        #TODO mittlere Kettenlänge bei Punktgitter noch berechnen
        chainlength_mean = eta[0]
    elif morphology_type == 'crystallite':
        chainlength_mean = 1
    N = int(N_carriers / chainlength_mean) - 1
    # Fermienergie wird bei Kristalliten nicht benötigt
    if morphology_type == 'crystallite':
        #!!! Kristallite können jetzt auch die Funktion Fermienergie verwenden.
        # Bei Wällen für die Korngrenzen muss occupy_traps = True sein.
        e_Fermi = np.nan
        occupied_sites = np.empty((0,3), dtype=np.int)
        number_carriers = 0
    elif fill_with_coulomb:
        e_Fermi = np.nan
        # Plätze besetzen unter Berücksichtigung der Coulombwechselwirkung
        occupied_sites, potential = FillSitesWCoulomb(E, N)
        # Anzahl der Ladungsträger entspricht der gewünschten Anzahl
        number_carriers = N
        # Potential der Ladungsträger auf die Energielandschaft addieren
        E += potential
        if debugmode:
            print('sites filled with Coulomb interaction')
    elif fill_with_fermi:
        s = time.perf_counter()
        # Fermienergie mit durchschnittlicher Kettenlänge
        e_Fermi = Fermienergie(E, morphology, N)
        e = time.perf_counter()
        if debugmode:
            print('time for calculating fermi energy', e - s)
            print('e_Fermi', e_Fermi)
        # Gitterplätze mit Ladungsträgern füllen, ist zufällig
        s = time.perf_counter()
        occupied_sites, number_carriers = FillSitesWFermi(E, morphology, e_Fermi)
        e = time.perf_counter()
        if debugmode:
            print('sites filled up to the fermi level')
            print('time for filling sites', e - s)
            print('number of charge carriers from gate voltage', N_carriers)
            print('number of filled sites', len(occupied_sites))
    else:
        e_Fermi = np.nan
        # Plätze besetzen ohne Berücksichtigung der Coulombwechselwirkung
        occupied_sites = FillSites(E, morphology, N)
        # Anzahl der Ladungsträger entspricht der gewünschten Anzahl
        number_carriers = N
        if debugmode:
            print('sites filled neither with Coulomb nor with fermi energy')
    # besetzten Zustände speichern
    if save_data:
        np.save(fn_occu, occupied_sites)

# ----------------------------------------------------------------------------
    # wenn keine Dateien geladen wurden, Ladungsträger neu verteilen
    #
    if not load_files:
        # immer die gleiche Energielandschaft? Dann jedes Mal andere Ladungsträger
        if not random_landscape:
            rnd.seed(trial)
        # Platziere sich bewegenden Ladungsträger
        carrier_final = PlaceCarrier(box, morphology, occupied_sites)
        # Globale Zeit
        global_time = 0.
        # Zähler der Simulationsschritte
        a = 0
        # wenn die Datei mit den Positionen existiert, wird sie gelöscht
        if os.path.exists(fn_car):
            os.remove(fn_car)
        # Kopfzeilen in der Ausgabedatei schreiben
        WriteOutputHeader(trial, number_carriers, e_Fermi, eta)

    return global_time, a, carrier_final, morphology, E, occupied_sites

def WriteOutputHeader(trial, N, e_Fermi, eta):
    """
    header-Zeilen der Ausgabedatei erstellen.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    N : int
        Anzahl der Ladungsträger im Device, inkl. sich bewegender Ladungsträger.
    e_Fermi : float
        Fermienergie
    eta : float oder tuple
        Bruchteil, je nach Morphologieart
    """
    fn = os.path.join(folder_out, 'carrier_trial' + str(trial) + '.asc')
    # Kopfzeilen der Datei
    header = '# N_carriers ' + str(N) + '\n' + \
             '# e_Fermi ' + str(e_Fermi) + '\n'
    if morphology_type == 'pointgrid':
        header += '# eta_traps ' + str(eta[1]) + '\n'
    elif morphology_type == 'crystallite':
        header += '# eta_boundary ' + str(eta) + '\n'
    # schreibe die Kopfzeilen
    with open(fn, 'wb') as f:
        f.write(header.encode())

def WriteOutput(trial, position_list, time_list):
    """
    Schreibe die Liste mit den Positionen des Ladungsträgers zu den jeweiligen
    Zeiten in die Ausgabedatei.
    Speichere den Zustand des Zufallsgenerators.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    position_list : 2D array
        Liste der Positionen des Ladungsträgers
```

```
    time_list : 1D array
        Liste der Zeitpunkte zu den Positionen
    """
    fn = os.path.join(folder_out, 'carrier_trial' + str(trial) + '.asc')
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')

    # schreibe die Liste der Ladungsträger und den Zeitpunkt in die Datei
    with open(fn, 'ab') as f:
        for i in range(len(position_list)):
            header = 'time ' + str(time_list[i])
            np.savetxt(f, position_list[i], delimiter='\t', fmt='%1i', \
                       header=header)
    # speichere den Zustand des Zufallszahlengenerators
    with open(fn_rand, 'wb') as f:
        pickle.dump(rnd.get_state(), f, protocol=3)

################################################################################
# Funktionen für Parallelisierung
# ------------------------------------------------------------------------------

def StartSimulation(inQ, outQ, trial_list):
    """
    Schicke Daten an die Prozesse und warte bis alle fertig sind.
    inQ : queue
    outQ : queue
    trial_list : list
        Liste mit den Nummern der Trials
    """
    for i in trial_list:
        inQ.put(i)

    # warte darauf, dass alle Trials beendet sind
    for i in trial_list:
        trial = outQ.get()
        print('Trial', trial, 'finished.')

def Worker(inQ, outQ):
    """
    Trials parallel berechnen
    """
    if debugmode and N_nodes > 1:
        print('process', os.getpid(), 'on node', MPI.Get_processor_name())
    while True:
        # eine Nummer aus der Auftragsliste ziehen
        trial = inQ.get()
        if trial is None:
            break

        # starte MonteCarlo-Trial
        MonteCarloTrial(trial)

        # Lösche die Dateien der Morphologie und Energielandschaft
        if delete_data:
            # Dateinamen
            fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) +
'.npy')
            fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
            fn_occu = os.path.join(folder_out, 'occupation_trial' + str(trial) +
'.npy')
            fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) +
'.pkl')
            if os.path.exists(fn_morph):
                os.remove(fn_morph)
            if os.path.exists(fn_E):
                os.remove(fn_E)
            if os.path.exists(fn_occu):
                os.remove(fn_occu)
            if os.path.exists(fn_rand):
                os.remove(fn_rand)

        # schicke Nummer des fertigen Trials
        outQ.put(trial)

################################################################################
# Monte-Carlo-Simulation
# ------------------------------------------------------------------------------

def MonteCarloTrial(trial):
    """
    Funktion für MonteCarlo-Trial.
    """
    # --------------------------------------------------------------------------
    # Initialisierung des Trials
    global_time, a, carrier_final, morphology, E, occupied_sites = Initialize(trial)
    # wenn die Simulation schon beendet ist, abbrechen
    if global_time is None:
        return
    # Ladungsträger in der Simulationsbox (mit periodischer Randbedingung)
    carrier = carrier_final % box

    # --------------------------------------------------------------------------
    # Sprungraten berechnen
    #
    s = time.perf_counter()
    # Berechne die Sprungraten für jede Position des ladungsträgers
    jumprates = CalculateRates(E, morphology, occupied_sites)
    # kumulative Summe für die Auswahl des Sprungs vorher berechnen
    jumprates = np.cumsum(jumprates, axis=0)
    e = time.perf_counter()
    if debugmode:
        print('time for jumprates', e - s)

    # Liste mit den letzten write_every Positionen des Ladungsträgers
    position_list = np.empty((write_every, 3), dtype=np.int)
    # Liste mit den letzten write_every Zeitpunkten
    time_list = np.empty(write_every)

    # --------------------------------------------------------------------------
    # Monte-Carlo-Schritte
```

```
    #
    # MonteCarlo-Schritt je Schleifendurchlauf
    while True:
        # alle position_steps Sprünge Position und Zeit des Ladungsträgers speichern
        if a % position_steps == 0:
            i = int(a/position_steps % write_every)
            position_list[i] = carrier_final
            time_list[i] = global_time
            # gespeicherte Positionen in Datei schreiben
            if i == write_every - 1:
                WriteOutput(trial, position_list, time_list)

        # Zufallszahl um ein Ereignis auszuwählen
        r = rnd.random()*jumprates[-1, carrier[0], carrier[1], carrier[2]]
        # Waiting time für jeden Schritt berechnen
        global_time += -np.log(rnd.random()) / jumprates[-1, carrier[0], carrier[1],
carrier[2]]
        for i in range(number_jumps):
            if r < jumprates[i, carrier[0], carrier[1], carrier[2]]:
                # bewege Ladungsträger
                carrier += jumps[i]
                carrier_final += jumps[i]
                # periodische Randbedingung
                carrier[0] = carrier[0] % x_length
                carrier[1] = carrier[1] % y_length
                break

        # Abbruch der Simualtion, wenn die Anzahl an Simulationsschritten
überschritten wurde
        if a >= simulation_steps - 1:
            i = int(a / position_steps % write_every)
            # schreibe die restlichen gepufferten Einträge
            if not i == write_every - 1:
                WriteOutput(trial, position_list[:i+1], time_list[:i+1])
            WriteOutput(trial, [carrier_final], [global_time])
            return
        a += 1

################################################################################
# Hauptfunktion zur Steuerung des Programmablaufs
# ------------------------------------------------------------------------------

def Main():
    """
    Hauptfunktion, die die arbeitenden Prozesse startet und anschließend die
    Simulation. Nach Ende der Simulation werden die Prozesse wieder angehalten.

    Ist etwas schneller als Parallelisierung mit pool.map()
    """
    # Verwende MPI nur bei mehr als einem Knoten
    if N_nodes > 1:
        # Kommunikator erzeugen
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        size = comm.Get_size()
        processor_name = MPI.Get_processor_name()
        if debugmode:
            print('rank', rank, 'size', size, 'on', processor_name)
        # Anzahl der Trials für jeden Knoten
        n = int(number_trials / size) + 1
        # Liste mit den Nummern der Trials für jeden Knoten
        trial_list = np.full(n, -1, dtype=np.int64)
        # Liste mit allen Trialnummern
        all_trials = None
        if rank == 0:
            all_trials = np.arange(number_trials)
        # verteile die Liste
        comm.Scatterv(all_trials, trial_list, root=0)
        # Liste kürzen
        trial_list = trial_list[np.where(trial_list >= 0)]
        if debugmode:
            print('rank', rank, 'received trials', trial_list)
    else:
        trial_list = np.arange(number_trials)

    # prüfe ob die Datei carrier_alltrials.asc schon existiert
    fn_alltrials = os.path.join(folder_out, 'carrier_alltrials.asc')
    # wenn ja, breche ab
    if os.path.exists(fn_alltrials):
        return
    # initialisiere Queues
    inQ = mp.Queue()
    outQ = mp.Queue()

    # Prozesse aufbauen
    workers = [mp.Process(target=Worker, args=(inQ, outQ))
               for i in range(N_processors)]

    # start the workers
    for w in workers:
        w.start()

    # Starte Simulation
    s = time.perf_counter()
    StartSimulation(inQ, outQ, trial_list)
    e = time.perf_counter()
    print('time for all trials', e - s)

    # Prozesse beenden
    for i in range(N_processors):
        inQ.put(None)
    # warte auf das Ende aller Prozesse
    for w in workers:
        w.join()

################################################################################
# Kommandozeilen-Optionen festlegen
# ------------------------------------------------------------------------------

# ---------------------------------
```

```
# mögliche Kommandozeilen-Optionen:
# --------------------------------
#   Nummer des Parametersatzes: Zahl, muss erste Option sein
#   alle weiteren Optionen sind vom Format key=value
#   Anzahl der Prozesse zur Parallelisierung mit 'cores='

# Werden Optionen übergeben? Ist der Fall, wenn auf dem Cluster gerechnet wird.
cluster_rechnen = len(sys.argv) != 1

# Anzahl der Prozesse auf dem lokalen Rechner
N_processors = 4
# Anzahl der Knoten, default 1
N_nodes = 1

if cluster_rechnen:
    for option in sys.argv[2:]:
        key, value = option.split('=')
        # Anzahl der Prozesse
        if key == 'cores':
            N_processors = int(value)
        elif key == 'nodes':
            N_nodes = int(value)
        else:
            raise IOError('Kommandozeilenoption ' + str(option) + \
                          ' nicht bekannt.')

# ------------------------------------------------------------------------------
# Nummer des Parametersatzes
#

if cluster_rechnen:
    # Nummer des Parametersatzes ermitteln
    parametersatz = int(sys.argv[1])
else:
    # Testordner auf lokalem Rechner
    folder_local = 'C:/Daten/TobiasMeier/Simulationen/Tests'
    # wechsele in das lokale Verzeichnis
    os.chdir(folder_local)
    # Nummer des Parametersatzes
    parametersatz = 1

##############################################################################
# Parameter aus Datei einlesen
# ------------------------------------------------------------------------------

# Dateinamen für den Parametersatz
fn_params = os.path.join('Parameters' + str(parametersatz), \
                         'parameters' + str(parametersatz) + '.asc')
# Dictionary aus der Parameterdatei lesen
d = LoadParameters(fn_params)

# Gitterabstände (nm)
dx = d['dx']
dy = d['dy']
dz = d['dz']

# Boxlänge (Anzahl der Gitterplätze)
x_length = int(d['x_length'])
y_length = int(d['y_length'])
z_length = int(d['z_length'])

# Anzahl der Trials
number_trials = int(d['number_trials'])

# Soll eine Morphologie für alle Trials geladen werden? Muss sich im Hauptordner
befinden
load_morphology = d['load_morphology']

# Soll die Energielandschaft zufällig sein? (sonst immer wie Trial 0)
random_landscape = d['random_landscape']

# maximale Simulationszeit (ns)
simulation_steps = d['sim_steps']

# ------------------------------------------------------------------------------
# Eigenschaften der Morphologie

# Art der Morphologie
morphology_type = d['morphology']
# Ist die erste Schicht ausschließlich mit Fallen besetzt?
trap_layer = d['trap_layer']

# Kristallite
if morphology_type == 'crystallite':
    # Kantenlängen der Kristallite (nm)
    crystal_a, crystal_b = d['crystal_a'], d['crystal_b']
    # Winkel der Parallelogramme für die Kristallite (°)
    crystal_gamma = d['crystal_gamma']
    # Anzahl der Startpunkte für Kristallitwachstum
    number_seeds = int(d['number_seeds'])
    # Dicke der Grain boundaries
    boundary_gap = d['boundary_gap']
# Punktgitter
elif morphology_type == 'pointgrid':
    # sollen beim Besetzen der Plätze die Coulombwechselwirkung berücksichtigt werden?
    fill_with_coulomb = d['fill_with_coulomb']
    # Soll für das Besetzen der Gitterplätze die Fermienergie berechnet werden?
    fill_with_fermi = d['fill_with_fermi']
    # Fallenverteilungsfunktion
    trap_distribution = d['trap_distribution']
    # Sollen sich die Fallen nur in der Grenzschicht befinden?
    interface_traps = d['interface_traps']
    # Dichte der Fallenzustände
    trapdensity = d['trapdensity']
    # Kettenlänge
    chain_length = int(d['chain_length'])
    # Breite der Verteilung der Kettenlänge
    chain_sigma = d['chain_sigma']
    # fester Versatz zwischen Ketten
    chain_offset = d['chain_offset']
```

```
    if chain_offset != 'random':
        chain_offset = int(chain_offset)

# ------------------------------------------------------------------------------
# Elektrische und elektronische Eigenschaften
#
# Verwendetes Energiemodell
energy_model = d['energy_model']
# Breite der Energieverteilung sigma (eV)
sigma = d['sigma']
# Zentrum der Energieverteilung der Fallenzustände (eV)
E_trap = d['E_trap']
# Breite der Energieverteilung der Fallenzustände (eV)
sigma_trap = d['sigma_trap']
# Dehnungsfaktor für den exponentiellen Anteil gegenüber dem Gaußanteil
# sigma_exp = stretch_exp * sigma_trap
if trap_distribution == 'gaussexp':
    stretch_exp = d['stretch_exp']
# Dicke des Gate-Dielektrikums (nm)
d_gate = d['d_gate']
# relative Dielektrizitätskonstante des Gate-Dielektrikums
epsilon_gate = d['epsilon_gate']
# Gate-Spannung (V)
U_Gate = d['U_Gate']
# Drain-Spannung (V)
U_Drain = d['U_Drain']
# relative Dielektrizitätskonstante
epsilon_rel = d['epsilon_rel']

# ------------------------------------------------------------------------------
# Parameter für Sprünge
#
# Temperatur
temperature = d['temperature']
# Hüpfmechanismus
hopping = d['hopping']
# Miller-Abrahams
if hopping == 'MA':
    # Attempt-to-hop frequency (1/ns)
    nu0 = d['nu0']
# Marcus-Rate
elif hopping == 'marcus':
    # Reorganisationenergie
    reorg_energy = d['reorg_energy']
    # Transferintegral
    J0 = d['J0']
else:
    raise ValueError('unknown hopping mechanism')
# inverse Lokalisierungskonstanten für Hoppingrate nach Miller-Abrahams (1/nm)
# innerhalb einer Polymerkette
gamma_intra = d['gamma_intra']
# Kopplungskonstante in x-Richtung
gamma_x = d['gamma_x']
# Kopplungskonstante in y-Richtung
gamma_y = d['gamma_y']
# Kopplungskonstante in z-Richtung
gamma_z = d['gamma_z']
# Anzahl der Nächsten Nachbarn für Sprünge
nearest_neighbors = int(d['NN_hops'])


################################################################################
# Konstanten und globale Variablen
# ------------------------------------------------------------------------------

# ------------------------------------------------------------------------------
# Konstanten
#
# kT (eV)
kT = 8.617e-5 * temperature
# Cutoff-Radius für die Dipolwechselwirkung bei räumlich korrelierter Energie (nm)
cutoff_dipoles = 25
# Faktor für Coulomb-Wechselwirkung, Annahme kubisches Gitter! in eV.
faktor_Coulomb = 1.602e-19 / 4 / np.pi / 8.854e-12 / epsilon_rel * 1e9

# Potentialunterschied bei einem Schritt in x-Richtung
voltage_difference = U_Drain / x_length
# Potentialunterschied bei einem Schritt in z-Richtung durch die Gate-Spannung
gate_potential_step = epsilon_gate / epsilon_rel * abs(U_Gate) / 2 / d_gate * dz
# Anzahl der Ladungsträger durch den Kondensator
N_carriers = int(epsilon_gate * 8.854e-12 * x_length * y_length * dx * dy *
abs(U_Gate) \
                 / 1.602e-19 / d_gate * 1e-9)

# Konstanten für Marcus-Rate
if hopping == 'marcus':
    C1_marcus = 2 * np.pi / 6.582e-16 * J0**2 / np.sqrt(4 * np.pi * kT * reorg_energy)
* 1e-9
    C2_marcus = 4 * reorg_energy * kT

# ------------------------------------------------------------------------------
# Globale Variablen
#
# Array für Größe der Simulationsbox
box = np.array([x_length, y_length, z_length])
# Array für Abstände der Gitterplätze
gridspacing = np.array([dx, dy, dz])
# mögliche Sprungrichtungen
jumps = PossibleJumps(nearest_neighbors, 3)
# Anzahl der Sprungrichtungen
number_jumps = len(jumps)

# ------------------------------------------------------------------------------
# Parameter für die Dateiausgabe
#

# Verzeichnis des aktuellen Parametersatzes
folder_out = 'Parameters' + str(parametersatz)

# Anzahl der Sprünge, nach denen die Position gespeichert werden soll
```

```
if morphology_type == 'crystallite':
    position_steps = 10000
elif chain_length == 1:
    position_steps = 1000
else:
    position_steps = 10000 * np.int( \
                     np.exp(2*(gamma_x - gamma_intra)*np.min(gridspacing)) )
# Anzahl der Positionen, die zwischengespeichert werden
write_every = 1000

###############################################################################
# starte Simulation mit Hauptfunktion
# -----------------------------------------------------------------------------

if __name__ == '__main__':
    Main()

# -----------------------------------------------------------------------------
# Programmende
###############################################################################
```

A.2.2. Simulationsprogramm zur Vielteilchensimulation Teil 1

```
# -*- coding: utf-8 -*-
"""
Simulation von Ladungstransport durch OFET.

Charakteristische Merkmale:

    - Mehrteilchensimulation
    - korrelierte Energielandschaft möglich
    - exakte Coulombwechselwirkung zwischen den Ladungsträgern und periodischen
      Fortsetzungen in den angrenzenden Simulationsboxen.
    - Orthorhombisches Gitter (unterschiedliche Gitterabstände) mit periodischen
      Randbedingungen in x- und y-Richtung.
      In z-Richtung beschränkt.
    - Gitterplätze werden zu Beginn zufällig mit Ladungsträgern befüllt entsprechend
      der Kapazität des Gatedielektrikums.
    - Miller-Abrahams Sprungraten.
    - Frühere Simulation kann fortgesetzt werden.
    - Parallelisierung mit multiprocessing: bezüglich der Ladungsträger
    - Nachdem ein Trial fertig ist, kann die Datei für die Energielandschaft
      gelöscht werden.
    - Coulombpotential wird an jedem Punkt berechnet und nach jedem Sprung
      aktualisiert.
    - Verschiedene Fallenverteilungen möglich.
    - Positive und negative Gatespannung möglich.
    - Marcus hopping rate möglich.
    - Daten in shared memory werden in größeren Stücken geschrieben, sollte
      die Anzahl an Locks reduzieren, ist dann etwas schneller.
    - Trials sind reproduzierbar in den Zufallszahlen.
    - Der Zustand des Zufallsgenerator wird gespeichert und bei Fortsetzung
      wieder aufgerufen.
    - Der Zufallszahlengenerator für die Energielandschaft und die Ladungsträger
      können entkoppelt werden in der Parameterdatei. Bei Entkopplung wird immer
      die Energielandschaft von Trial 0 erzeugt.
    - Eine Morphologie kann geladen werden für alle Trials. Die Datei muss
      morphology.npy heißen und sich im Hauptordner befinden.
    - Beim Schreiben der output-Dateien wird zuerst eine temporäre Sicherungsdatei
angelegt,
      dann die ursprüngliche verändert und dann die Sicherungskopie gelöscht.

Dateiausgabe:

    - Positionen und Zeitpunkt aller Ladungsträger nach jedem 10 * N_carriers Sprung.
"""
import numpy as np
import numpy.random as rnd
import os
import itertools
import multiprocessing as mp
import sys
import time
import pickle
import shutil

# Modus für Fehlersuche
debugmode = True
# Dateien für Morphologie und Energielandschaft speichern?
# Kann sinnvoll sein bei großen Dateien diese nicht zu speichern
save_data = True
# Morphologie und Energielandschaft nach Beendigung eines Trials löschen?
delete_data = True

###############################################################################
# Funktionendefinition
# -----------------------------------------------------------------------------

# -----------------------------------------------------------------------------
# Energielandschaft
#
def Energylandscape(box, sigma, morphology, model='GDM', gatevoltage=True):
    """
    Energielandschaft erstellen.

    box : 1D array
        Größe der Simulationsbox
    sigma : float
        gewünschte Standardabweichung der Energielandschaft (eV)
    morphology : 3D array
        Morphologie
```

```
    model : str
        Gewünschtes Modell. Mögliche Modelle: gaussian disorder model ('GDM'),
        correlated disorder model ('CDM').
    gatevoltage : bool
        Soll das Gatepotential zur Energielandschaft dazuaddiert werden?

    returns
    -------
    E : 3D array
        Energiewerte für jeden Gitterplatz nach gewünschtem Modell.
    """
    if model == 'GDM':
        E = GDM(box, sigma, morphology)
    elif model == 'CDM':
        E = CDM(box, sigma)
    else:
        raise ValueError('model for energylandscape unknown:', model)

    if gatevoltage:
        # Potential durch Gateelektrode dazuaddieren
        gate_potential = np.empty(box)
        gate_potential[:] = np.linspace(0, gate_potential_step*(box[2]-1), \
                                        num=box[2]).reshape(1,1,-1)
        # in der Fallenschicht einen halben Potentialsprung dazuaddieren
        if trap_layer:
            gate_potential[:,:,0] += gate_potential_step / 2
        E += gate_potential
    return E

def GDM(box, sigma, morphology):
    """
    Gaussian disorder model.

    Erstelle eine Energielandschaft mit statischen Unordnung. Die Energien der
    Gitterplätze sind einer Gaußverteilung entnommen. Morphologie wird
    berücksichtigt.

    box : 1D array
        Größe der Simulationsbox.
    sigma : float
        Standardabweichung der Verteilung der Energien.
    morphology : 3D array
        Morphologie

    returns
    -------
    E : 3D array
        Energielandschaft der Größe `box` nach dem GDM.
    """
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_inverse=True)
    # Energiewerte für jede Kettennummer
    E_reduced = rnd.normal(loc=0, scale=sigma, size=len(numbers))
    # rekonstruiere die Morphologie
    E = E_reduced[indices].reshape(box)
    # Positionen der Fallenzustände
    traps = np.where(morphology <= -1)
    # ------------------------------
    # Energiewerte für Fallenzustände
    #
    # Gaußverteilung
    if trap_distribution == 'gauss':
        E_traps = rnd.normal(loc=E_trap, scale=sigma_trap, size=len(traps[0]))
    elif trap_distribution == 'exp':
        E_traps = E_trap - rnd.exponential(scale=sigma_trap, size=len(traps[0]))
    elif trap_distribution == 'gaussexp':
        size_gauss = int(len(traps[0]) * np.sqrt(np.pi) / (stretch_exp * np.sqrt(2) +
np.sqrt(np.pi)))
        size_exp = len(traps[0]) - size_gauss
        gauss_traps = []
        while len(gauss_traps) < size_gauss:
            x = rnd.normal(loc=E_trap, scale=sigma_trap)
            if x >= E_trap:
                gauss_traps.append(x)
        E_traps = np.append(gauss_traps, E_trap -
rnd.exponential(scale=stretch_exp*sigma_trap, size=size_exp))
    else:
        raise ValueError('trap distribution unknown', trap_distribution)
    # Energiewerte den Plätzen zuweisen
    E[traps] = E_traps
    return E

def CDM(box, sigma):
    #TODO Morphologie für CDM berücksichtigen
    """
    Correlated disorder model.

    Energielandschaft mit räumlicher Korrelation erstellen.
    Nach Phys Rev B 83, 085206 (2011), Bumblebee

    box : 1D array
        Größe der Simulationsbox
    sigma : float
        gewünschte Standardabweichung der Energielandschaft (eV)

    returns
    -------
    E : 3D array
        Energielandschaft mit räumlicher Korrelation.
    """
    # Array für Energielandschaft
    E = np.empty(box)
    # Stärke der Dipole auf den Gitterplätzen aus Standardabweichung sigma berechnen
    dipole_strength = sigma*8.854e-12*epsilon_rel*np.min(gridspacing)**2*1e-9 /
(2.35*1.602e-19)

    # Nummern aller Ketten und deren Indizes
#    numbers, indices = np.unique(morphology, return_inverse=True)

    # zufällig Dipole anordnen
```

```
    dipoles = RandomDipoles(box, dipole_strength)

    # Gitterpunkte einer Kugel mit dem Cutoffradius ohne den Punkt [0,0,0]
    indices = SphereIndices(cutoff_dipoles, gridspacing)

    # Liste aller Gitterpositionen
    x, y, z = np.mgrid[0:box[0], 0:box[1], 0:box[2]]
    pos_list = np.stack((x, y, z), axis=3).reshape(-1,3)

    # Parallelisierung mit multiprocessing
    inQ = mp.Queue()
    outQ = mp.Queue()
    # Prozesse initialisieren und starten
    workers = [mp.Process(target=WorkerCDM, args=(dipoles, indices, box, gridspacing,
inQ, outQ))
               for i in range(N_processes)]
    for w in workers:
        w.start()
    print('Prozesse gestartet')

    # Energie an jedem Gitterpunkt berechnen
    for p in pos_list:
        inQ.put(p)
    # auf Ergebnisse warten
    for i in range(len(pos_list)):
        pos, val = outQ.get()
        E[tuple(pos)] = val
    # Prozesse beenden
    for i in range(N_processes):
        inQ.put(None)
    # warte auf das Ende aller Prozesse
    for w in workers:
        w.join()

    # Energie skalieren in eV
    E *= -1.602e-19/(8.854e-12*epsilon_rel*1e-9)
    if debugmode:
        print('std von E', np.std(E))
    return E

def RandomDipoles(box, dipol_strength):
    """
    Gitter mit zufällig orientierten Dipolen füllen.

    box : 1D array
        Größe der Simulationsbox
    dipol_strength : float
        Stärke des Dipols (in Einheit der Elementarladung, nm)

    returns
    -------
    dipoles : 4D array
        Index 0: x-Koordinate, Index 1: y-Koordinate, Index 2: z-Koordinate,
        Index 3: Richtung des Dipols [x, y, z]
    """
    dipoles = np.empty((box[0], box[1], box[2], 3))
    phi = rnd.random(size=np.prod(box))*2*np.pi
    theta = rnd.random(size=np.prod(box))*np.pi
    dipoles[:,:,:,0] = np.reshape(dipol_strength*np.sin(theta)*np.cos(phi), box)
    dipoles[:,:,:,1] = np.reshape(dipol_strength*np.sin(theta)*np.sin(phi), box)
    dipoles[:,:,:,2] = np.reshape(dipol_strength*np.cos(theta), box)
    return dipoles

def SphereIndices(R, gridspacing, include_origin=False):
    """
    Ermittle die Koordinaten, die innerhalb einer Kugel mit festem Radius
    liegen. Der Kugelmittelpunkt liegt dabei auf der Null.

    R : float
        Radius der Kugel
    gridspacing : 1D array
        Gitterabstände, [dx, dy, dz]
    include_origin : bool
        Soll der Ursprungspunkt [0,0,0] mit in der Liste enthalten sein?

    returns
    -------
    indices : 2D array
        Liste der Koordinaten, die innerhalb der Kugel liegen. Index 0: Punkte,
        Index 1: [x,y,z]
    """
    # Anzahl der maximal möglichen Gitterpunkte in einer Richtung
    n = int(np.max(R/gridspacing))
    indices = np.array(list(itertools.product(range(-n,n+1), repeat=3)))
    length = np.sqrt(np.sum((indices*gridspacing)**2, axis=1))
    indices = indices[np.where(length <= R)]
    if not include_origin:
        indices = indices[np.any(np.not_equal(indices, [0,0,0]), axis=1)]
    return indices

# ----------------------------------------------------------------------------
# Morphologie erstellen
#
def CreateMorphology(morphology_type):
    """
    Erstelle die Morphologie
    - Es können verschiedene Schichten aneinandergefügt werden.
    - Die Ausrichtung der Ketten kann in jeder Schicht individuell eingestellt
      werden.
    - Fallen können in verschiedenen Schichten hinzugefügt werden.
    - Negative Zahlen sind Fallen, positive Polymersegmente. Null bedeutet leerer
      Platz und wird somit außer bei trap_layer nicht verwendet.

    box : 1D array
        Größe der Simulationsbox.
    morphology_type : str
        Art der zu erstellenden Morphologie
    returns
    -------
    morphology, eta_traps : 3D array, float
```

```
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Kette gehören.
        eta_traps ist die tatsächliche Fallendichte in der Schicht, in der
        Fallen vorhanden sind.
    """
    if morphology_type == 'pointgrid':
        if interface_traps:
            # Trapstates nur in der ersten Schicht
            interface, eta = CreatePointgrid(np.array([box[0], box[1], 1]),
chain_length, chain_sigma, chain_offset, 0, trapdensity)
            # Restliche Morphologie ohne Fallen
            morphology, e = CreatePointgrid(np.array([box[0], box[1], box[2] - 1]),
chain_length, chain_sigma, chain_offset, 0)
            # Kettennummern eindeutig machen
            morphology += np.max(interface) + 1
            morphology = np.concatenate((interface, morphology), axis=2)
        else:
            morphology, eta = CreatePointgrid(box, chain_length, chain_sigma,
chain_offset, 0, trapdensity)

    return morphology, eta

# ----------------------------------------------------------------------------

def CreatePointgrid(box, chain_length, chain_sigma, chain_offset, orientation,
trapdensity=0., periodic=True):
    """
    Erstelle einen Bereich für die Morphologie, in dem alle Ketten in einer
    Richtung verlaufen.
    - Es werden Gitterpunkte zu Ketten verbunden. Dabei werden alle Gitterpunkte
      durchnummeriert, wobei Gitterpunkte einer Kette die gleiche Nummer erhalten.
    - Es muss nicht jede Zahl als Nummer einer Kette vorkommen.
    - Kann verwendet werden, um mehrere solcher Kästen aneinanderzureihen.
    - Optionalen Parameter, wenn Ketten periodisch sein sollen.
    - Fallenzustände haben negative Nummern, jeder seinen eigenen Wert.

    box : 1D array
        Größe der Morphologie
    chain_length : int
        Länge der Ketten
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)
    trapdensity : float
        Wahrscheinlichkeit, dass ein Gitterplatz ein Fallenzustand ist
    periodic : bool
        Sollen die Ketten über die Ränder periodisch fortgesetzt werden?

    returns
    -------
    morphology, eta_traps : 3D array, float
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Polymerkette gehören. Ein Wert von -1 bedeutet
        ein Fallenzustand.
        eta_traps ist der Bruchteil der Gitterplätze mit Fallen. Wenn keine
        vorhanden sind, wird None zurückgegeben.
    """
    box = np.asarray(box)
    # die Richtung, in der die Ketten durchnummeriert werden, ist die Achse
    # der Orientierung. Vertausche die Achsen so, dass die Achse der Kettenausrichtung
    # die letzte Achse ist.
    shape = box[np.roll(np.arange(3), 2 - orientation)]
    # Achsen, die wieder zurück vertauscht werden müssen
    transpose = np.roll(np.arange(3), orientation - 2)

    # wenn keine periodische Randbedingung für die Kette gelten soll, dann eine
    # Kette mehr mitnehmen
    if not periodic:
        shape[2] += chain_length

    # ------------------------------------------------------------------------
    # Polymerketten mit bestimmter Länge erstellen
    #
    # Array für die Morphologie
    morphology = np.zeros(shape, dtype=np.int)
    # Laufindex der Kettennumer
    k = 1
    # die zwei Dimensionen durchlaufen, die nicht der Kettenausrichtung entsprechen
    for i in range(shape[0]):
        for j in range(shape[1]):
            # ein Kettenstrang (1D)
            line = []
            # Füge solange Segmente hinzu, bis der Strang voll ist
            while True:
                # Länge der neuen Ketten aus Verteilung bestimmen
                l = int(rnd.normal(loc=chain_length, scale=chain_sigma) + 0.5)
                # kann diese Kettenlänge noch hinzugefügt werden?
                if len(line) + l <= shape[2]:
                    line.extend([k]*l)
                    # Kettenindex erhöhen
                    k += 1
                else:
                    line.extend([k]*(shape[2] - len(line)))
                    # Strang zur Morphologie hinzufügen
                    morphology[i,j] = line
                    # Kettenindex erhöhen
                    k += 1
                    break

    # ------------------------------------------------------------------------
    # Fallenzustände einfügen
    #
    # Laufindex für die Fallenzustände
    trap = -1
    # Alle Gitterplätze werden durchlaufen
    for i in range(shape[0]):
        for j in range(shape[1]):
```

```
            for k in range(box[orientation]):
                # zufällig eine Falle platzieren
                if rnd.random() < trapdensity:
                    # Wenn eine Kette auseinandergerissen werden würde,
                    # wird die Falle nach hinten verschoben
                    x = k
                    while morphology[i, j, x-1] == morphology[i, j, x]:
                        x += 1
                        if x >= box[orientation]:
                            break
                    # Falle einbauen
                    if x < box[orientation]:
                        morphology[i, j] = np.append(morphology[i, j, :x], \
                                  np.append(trap, morphology[i, j, x:-1]))
                        # Laufindex für Fallen erniedrigen
                        trap -= 1
    # Versatz der Ketten nebeneinander
    offset = ChainOffset(box, chain_length, chain_offset, orientation)
    # Verschiebe die Kettennummern entsprechend dem Versatz
    for i in range(shape[0]):
        for j in range(shape[1]):
            morphology[i,j,:] = np.roll(morphology[i,j,:], -offset[i,j])
    # wenn keine periodischen Ketten, dann auf die ursprüngliche Länge kürzen
    if not periodic:
        morphology = morphology[:,:,:box[orientation]]
    # mittlere Kettenlänge bestimmen
    numbers, counts = np.unique(morphology, return_counts=True)
    if debugmode:
        print('average chain length', '{:.3g}'.format(np.average(counts)), '+-', \
          '{:.3g}'.format(np.std(counts)))
    # Anzahl der Fallen
    counts = np.where(morphology < 0)
    # Fallendichte im Bereich
    if len(counts[0]) > 0:
        eta_traps = len(counts[0])/np.prod(box)
    else:
        eta_traps = None
    # Achsen wieder zurücktauschen
    return np.transpose(morphology, axes=transpose), eta_traps

def ChainOffset(box, chain_length, chain_offset, orientation):
    """
    Erstelle zufällige Startpunkte der Ketten entlang der Richtung, in der sie
    ausgerichtet sind.

    box : 1D array
        Größe der Simulationsbox.
    chain_length : int
        Länge der Polymerketten.
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)

    returns
    -------
    chain_offset : 2D array
        Array der Offsets für die Polymerketten. Index 0, 1: Koordinaten ohne
        der, in der die Ketten ausgerichtet sind, z.B. y und z für Ketten in
        x-Richtung
    """
    # zyklisches Vertauschen der Dimensionen, sodass orientation die letzte Achse ist
    size = box[np.roll(np.arange(3), 2 - orientation)]
    if chain_offset == 'random':
        return rnd.randint(chain_length, size=size[:-1])
    # Ketten werden in die beiden übrigen Richtungen um festen Wert jeweils zueinander
verschoben
    else:
        offset0 = np.repeat(np.arange(size[1]).reshape((1,-1)), size[0],
axis=0)*chain_offset
        offset1 = np.repeat(np.arange(size[0]).reshape((-1,1)), size[1],
axis=1)*chain_offset
        return offset0 + offset1

# ------------------------------------------------------------------------------
# Sprungraten berechnen
#
def CalculateRates(parentP):
    """
    Wartet nur darauf, dass alle Childprozesse mit der Berechnung der Sprungraten
    fertig sind.

    parentP : list of pipes
        Liste der Pipes zum Datenaustausch mit den Prozessen,
        ist verbunden mit childP[i]
    """
    # warte darauf, dass alle Prozesse fertig sind
    for i in range(N_processes):
        parentP[i].recv()

def CalculateRatesPerCarrier(c, carriers, E):
    """
    Berechnet die Sprungraten für einen Ladungsträger in die verschiedenen Richtungen.

    c : int
        Index des Ladungsträgers
    carriers : 2D array
        Array mit den aktuellen Ladungsträgern
    E : 3D array
        Array der Energielandschaft
    potential : 3D array
        Coulomb-Potential an jedem Punkt

    returns
    -------
    k : 1D array, Länge = len(jumps)
        Sprungraten für einen Ladungsträger in alle Sprungrichtungen. Die Indizes
        entsprechen denen der Liste für die Sprungrichtungen.
```

```
    """
    k = np.empty(N_jumps)

    # alle Ladungsträger ohne den, der gerade springen soll und
    # Ladungen durch periodische Randbedingungen in x,y-Richtung
    replica_carriers = np.vstack((np.append(carriers[:c], carriers[c+1:], axis=0), \
                carriers - [x_length,0,0], carriers + [x_length,0,0], \
                carriers - [0,y_length,0], carriers + [0,y_length,0]))

    for j,jump in enumerate(jumps):
        # initial und final Position
        newpos = carriers[c] + jump
        # periodische Randbedingung in x,y-Richtung
        newpos[0] = newpos[0] % box[0]
        newpos[1] = newpos[1] % box[1]
        # Sprünge in z-Richtung sind beschränkt wegen nicht-periodischer Randbedingung
        if newpos[2] >= box[2]:
            k[j] = 0
        elif newpos[2] < 0:
            k[j] = 0
        # Sprünge auf einen schon besetzten Platz sind nicht möglich
        elif np.any(np.all(newpos == carriers, axis=1)):
            k[j] = 0
        # Sprungraten berechnen
        else:
            init = tuple(carriers[c])
            final = tuple(newpos)
            # Energie am Startgitterplatz
            E_init = E[init] + CalculatePotential_old(replica_carriers, carriers[c])
            # Energie am Zielgitterplatz
            E_final = E[final] + jump[0] * voltage_difference + jump[2] *
gate_potential_step + \
                      CalculatePotential_old(replica_carriers, newpos)
            r_gamma = np.sqrt(np.sum((jump * gridspacing * gamma)**2))
            k[j] = Hoppingrate(r_gamma, E_init, E_final)

    return k

def Hoppingrate(r_gamma, E_i, E_f):
    """
    Berechne Hüpfrate nach Marcus oder Miller-Abrahams.

    r_gamma : float
        Abstand zwischen den Gitterplätzen multipliziert mit Kopplungskonstante.
    E_i : float
        Energie am Startgitterplatz
    E_f : float
        Energie am Zielgitterplatz

    returns
    -------
    für Marcus-Rate:
    k = 2*pi / hbar * |J_ij|**2 * np.sqrt(1 / (4*pi*kB*T*lambda)) *
        np.exp(-(E_j - E_i + lambda)**2 / (4*lambda*kB*T))
    """
    if hopping == 'MA':
        if E_i >= E_f:
            return nu0 * np.exp(-2 * r_gamma)
        else:
            return nu0 * np.exp(-2 * r_gamma) * np.exp((E_i - E_f)/kT)
    else:
        return C1_marcus * np.exp(-2 * r_gamma) * np.exp(-(E_f - E_i +
reorg_energy)**2 / C2_marcus)

# -----------------------------------------------------------------------------
# Coulomb-Potential
#

def CalculatePotential_old(carriers, position):
    """
    Berechne das Potential an einer Position, das durch die Ladungsträger
    verursacht wird. Der sich bewegende Ladungsträger darf nicht in der Liste
    sein!
    Momentan wird die Coulomb-Wechselwirkung exakt berechnet.
    Nach Demeyu (2007), Phys Rev B 76, 155202

    carriers : 2D array
        Array mit den Positionen der Ladungsträger inklusive der Ladungsträger
        in den benachbarten Boxen aufgrund der periodischen Randbedingunen in
        x und y-Richtung.
    position : 1D array
        Position, an der das Potential berechnet werden soll.
    returns : float
        Potential an der zu berechnenden Position in eV.
    """
    # exakte Coulomb-Wechselwirkung
    V = np.sum(1 / np.sqrt(np.sum((carriers - position)**2, axis=-1)))
    return V * faktor_Coulomb

# -----------------------------------------------------------------------------

def PlaceCarriers(N, box, x=None, y=None, z=None):
    """
    Ein bestimmter Bruchteil der Gitterplätze wird zufällig mit Ladungsträgern
gefüllt.

    N : int
        Anzahl der zu platzierenden Ladungsträger
    box : 1D array
        Array der Boxgröße
    x,y,z = None : int
        Wenn angegeben ist, dann wird diese Koordinate festgelegt

    returns
    -------
    2D array
        Array der besetzten Gitterplätze
    """
    # Dimensionen
    dim = len(box)
```

```
    # Array der Ladungsträger
    c = np.empty((0, dim), dtype=np.int)
    # Anzahl bereits platzierte Ladungsträger
    i = 0
    while i < N:
        p = np.array(rnd.random(dim)*box, dtype=np.int)
        # setze eine Koordinate fest, wenn gewünscht
        if x is not None:
            p[0] = x
        if y is not None:
            p[1] = y
        if z is not None:
            p[2] = z
        # überprüfe, ob neue Position schon besetzt ist
        if not np.any(np.all(c == p, axis=1)):
            c = np.vstack((c,p))
            i += 1
    return c

def PossibleJumps(NN, dim):
    """
    Ermittle mögliche Sprungrichtungen.

    NN : int
        Anzahl nächster Nachbarn, bei ungleichen Gitterabständen sollte diese
        höher eingestellt werden.
    dim : int
        Anzahl der Dimensionen

    returns
    -------
    2D array
        Array mit den möglichen Sprungrichtungen, sortiert nach steigendem
Sprungabstand
    """
    all_jumps = np.array(list(itertools.product(range(-NN,NN+1), repeat=dim)))
    jump_length = np.sqrt(np.sum(np.square(all_jumps*gridspacing), axis=1))
    jump_length_unique, counts = np.unique(jump_length, return_counts=True)
    return all_jumps[np.argsort(jump_length)[1:np.sum(counts[1:NN+1])+1]]

def ConvergenceCriteria(j_average):
    """
    Kriterium dafür, dass die Stromdichte konvergiert ist. Standardabweichung
    der Mittelwerte muss kleiner als ein Grenzwert sein (10 %)

    j_average : 1D array
        Array mit den Mittelwerten der Stromdichten

    returns
    -------
    : bool
        Kriterium erfüllt oder nicht.
    """
    return np.std(j_average) / np.average(j_average) < conv_current
```

```
# ------------------------------------------------------------------------------
# Dateiein- und ausgabe
#
def LoadParameters(file):
    """
    Lade den Parametersatz aus einer Datei. Wandelt die Werte wenn möglich in
    float um.

    file : str
        Name der Parameterdatei

    returns
    -------
    parameter : dict
        Ein dictionary der Parameter
    """
    key = []
    value = []
    # Datei einlesen
    with open(file, 'r') as f:
        lines = f.read().split('\n')
    # Spalten den Namen und Werten zuordnen
    for l in lines:
        if not l == '':
            k, v = l.split('\t')
            key.append(k)
            if v == 'True':
                value.append(True)
            elif v == 'False':
                value.append(False)
            else:
                # Werte wenn möglich in Zahlen umwandeln
                try:
                    value.append(float(v))
                except:
                    value.append(v)
    return dict(zip(key, value))

def ConvertBool(value):
    """
    Konvertiere einen True/False Parameterwert nach bool in der Parameterdatei.

    value : str
        String, der nach True/False konvertiert werden soll.

    returns
    -------
    value : bool
        True oder False
    """
    if value == 'True':
        return True
    elif value == 'False':
        return False
    else:
```

```
        raise ValueError('Could not convert', value, 'to bool')

def Initialize(trial):
    """
    Zuerst wird gesucht, ob eine frühere Simulation fortgesetzt werden kann.
    Das ist der Fall, wenn alle Dateien für Ladungsträger, Energielandschaft,
    Morphologie und besetzte Zustände gleichzeitig existieren und lesbar sind.
    Ansonsten wird alles neu erstellt.
    Außerdem wird der Zufallsgenerator initialisiert bzw. aus der vorherigen
    Simulation geladen.

    trial : int
        Nummer des Trials

    returns
    -------
    dictionary mit den relevanten Variablen, siehe am Ende der Funktion

    Wenn die Simulation bereits fertig ist, wird `None` zurückgegeben.
    """
    # ------------------------------------------------------------------------
    # alle Dateinamen
    #
    # Dateiname für die Position des Ladungsträgers
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    # Dateinamen, unter dem die Morphologie gespeichert wird
    fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) + '.npy')
    # Dateiname der Morphologie, falls diese aus dem Hauptordner geladen werden soll
    fn_morph_all = 'morphology.npy'
    # Dateiname, unter dem die Energielandschaft gespeichert wird
    fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
    # Dateiname für den Zustand des Zufallszahlengenerators
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')
    # prinzipiell erstmal keine Dateien laden
    load_files = False
    # wurde der Zufallsgenerator initialisiert?
    rng_seeded = False

    # ------------------------------------------------------------------------
    # Prüfe, ob eine frühere Simulation existiert
    #
    #!!! hier kann man nochmal prüfen, wann die Simulation fortgesetzt werden kann.
Bei beschädigten Dateien kann man eh nichts machen
    # ------------------------------------------------------------------------
    # Normalfall
    # wenn die carrier und rand Datei existiert und mindestens einer ihrer .tmp
Dateien nicht
    if os.path.exists(fn_car) and (not os.path.exists(fn_car + '.tmp') or not
os.path.exists(fn_rand + '.tmp')):
        # carrier Datei einlesen
        with open(fn_car, 'r', encoding='utf-8') as f:
            data = f.read().split('\n')
        # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
        try:
            # alle Zeitpunkte einlesen
            times = [ float(line.split()[2]) for line in data if line.startswith('#
time')]
            # letzte Positionen der Ladungsträger
            carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
        except:
            if debugmode:
                print('could not read carrier file from previous simulation in trial',
trial)
        else:
            # letzten Simulationszeitpunkt
            global_time = times[-1]
            if debugmode:
                print('global_time from previous simulation', global_time)
            # bestimme die Anzahl der bisherigen Simulationsschritte
            a0 = int((len(times) - 1) * position_steps)
            if debugmode:
                print('number of completed simulation steps', a0)
            # überprüfe, ob der Trial bereits beendet wurde
            if a0 >= sim_steps:
                return None
            # Stromdichten laden
            a1, j_last, j_average = ReadCurrentdensity(trial)
            if debugmode:
                # print('j_last', j_last)
                print('j_average', j_average)
            # überprüfe, ob die Stromdichte schon konvergiert ist
            if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                return None
            # Dateien nur laden, wenn alle existieren
            if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                os.path.exists(fn_rand):
                load_files = True
                if debugmode:
                    print('previous carrier list loaded')
    # ------------------------------------------------------------------------
    # Rückfallebene, wenn die Simulation irgendwo beim Speichern abgebrochen wurde
    # wenn beide temporären Dateien (carrier und rand) existieren (ursprüngliche damit
auch)
    elif os.path.exists(fn_car + '.tmp') and os.path.exists(fn_rand + '.tmp'):
        # carrier Datei einlesen
        with open(fn_car + '.tmp', 'r', encoding='utf-8') as f:
            data = f.read().split('\n')
        # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
        try:
            # alle Zeitpunkte einlesen
            times = [ float(line.split()[2]) for line in data if line.startswith('#
time')]
            # letzte Positionen der Ladungsträger
            carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
            # überprüfe, ob die Zufallsgeneratordatei eingelesen werden kann
            with open(fn_rand + '.tmp', 'rb') as f:
                random_state = pickle.load(f)
```

```
        except:
            # ----------
            # Rückfallebene funktioniert nicht, verwende Normalfall
            if debugmode:
                print('could not read temporary files in trial', trial, ', trying
original ones')
            # carrier Datei einlesen
            with open(fn_car, 'r', encoding='utf-8') as f:
                data = f.read().split('\n')
            # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
            try:
                # alle Zeitpunkte einlesen
                times = [ float(line.split()[2]) for line in data if
line.startswith('# time')]
                # letzte Positionen der Ladungsträger
                carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
                # überprüfe, ob die Zufallsgeneratordatei eingelesen werden kann
                with open(fn_rand, 'rb') as f:
                    random_state = pickle.load(f)
            except:
                # gar nichts hat funktioniert
                if debugmode:
                    print('both temporary and original data corrupted')
            else:
                # letzten Simulationszeitpunkt
                global_time = times[-1]
                if debugmode:
                    print('global_time from previous simulation', global_time)
                # setze den Zustand des Zufallsgenerators
                rnd.set_state(random_state)
                rng_seeded = True
                if debugmode:
                    print('previous random number generator state loaded')
                # bestimme die Anzahl der bisherigen Simulationsschritte
                a0 = int((len(times) - 1) * position_steps)
                if debugmode:
                    print('number of completed simulation steps', a0)
                # überprüfe, ob der Trial bereits beendet wurde
                if a0 >= sim_steps:
                    return None
                # Stromdichten laden
                a1, j_last, j_average = ReadCurrentdensity(trial)
                if debugmode:
                    # print('j_last', j_last)
                    print('j_average', j_average)
                # überprüfe, ob die Stromdichte schon konvergiert ist
                if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                    return None
                # Dateien nur laden, wenn alle existieren
                if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                    os.path.exists(fn_rand):
                    load_files = True
                    if debugmode:
                        print('previous carrier list loaded')
            # ----------
        else:
            if debugmode:
                print('using temporary files from previous simulation')
            # letzten Simulationszeitpunkt
            global_time = times[-1]
            if debugmode:
                print('global_time from previous simulation', global_time)
            # setze den Zustand des Zufallsgenerators
            rnd.set_state(random_state)
            rng_seeded = True
            if debugmode:
                print('previous random number generator state loaded')
            # bestimme die Anzahl der bisherigen Simulationsschritte
            a0 = int((len(times) - 1) * position_steps)
            if debugmode:
                print('number of completed simulation steps', a0)
            # überprüfe, ob der Trial bereits beendet wurde
            if a0 >= sim_steps:
                return None
            # Stromdichten laden
            a1, j_last, j_average = ReadCurrentdensity(trial)
            if debugmode:
                # print('j_last', j_last)
                print('j_average', j_average)
            # überprüfe, ob die Stromdichte schon konvergiert ist
            if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                return None
            # Dateien nur laden, wenn alle existieren
            if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                os.path.exists(fn_rand):
                load_files = True
                if debugmode:
                    print('previous carrier list loaded')

    # ------------------------------------------------------------------------
    # Zufallsgenerator initialisieren
    #
    # Zustand des Generators laden, wenn vorhanden
    if os.path.exists(fn_rand) and load_files and not rng_seeded:
        # lade den Zustand des Zufallsgenerators
        with open(fn_rand, 'rb') as f:
            random_state = pickle.load(f)
        # setze den Zustand des Zufallsgenerators
        rnd.set_state(random_state)
        if debugmode:
            print('previous random number generator state loaded')
    # Generator erstmalig initialisieren mit Trialnummer
    elif not rng_seeded:
        # zufällige Energielandschaft, oder die von Trial 0?
        if random_landscape:
            rnd.seed(trial)
            if debugmode:
```

```
            print('random number generator seeded with trial number')
        # prüfe Sinnhaftigkeit bei load_morphology, kann bei Bedarf auch weg
        if load_morphology:
            raise ValueError('morphology fixed, but energies random???')
    else:
        rnd.seed(0)
        if debugmode:
            print('random number generator seeded with 0')

# ----------------------------------------------------------------------------
# Morphologie initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_morph) and load_files:
    # lade vorhergehende Morphologie
    morphology = np.load(fn_morph)
    if debugmode:
        print('previous morphology loaded')
# eine Morphologie für alle Trials und Parameter?
elif load_morphology:
    # lade die Morphologie aus dem Hauptordner
    morphology = np.load(fn_morph_all)
    # kopiere sie in den Ordner für den jeweiligen Parameter
    if save_data:
        np.save(fn_morph, morphology)
    # Anteil der Fallen berechnen
    if interface_traps:
        eta = len(np.where(morphology < 0)[0]) / np.prod(box[:2])
    else:
        eta = len(np.where(morphology < 0)[0]) / np.prod(box)
    if debugmode:
        print('morphology loaded from main folder')
# Morphologie neu erstellen
else:
    if debugmode:
        print('new morphology')
    # Morphologie erstellen
    morphology, eta = CreateMorphology(morphology_type)
    # Morphologie speichern
    if save_data:
        np.save(fn_morph, morphology)

# ----------------------------------------------------------------------------
# Energielandschaft initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_E) and load_files:
    # lade vorhergehende Energielandschaft
    E = np.load(fn_E)
    if debugmode:
        print('previous energy landscape loaded')
# Energielandschaft neu berechnen
else:
    if debugmode:
        print('new energy landscape')
        s = time.perf_counter()
        # Energielandschaft neu mit Morphologie berechnen
        E = Energylandscape(box, sigma, morphology, model=energy_model)
        e = time.perf_counter()
        if debugmode:
            print('time for energy landscape', e - s)
        # Energielandschaft speichern
        if save_data:
            np.save(fn_E, E)

    # ------------------------------------------------------------------------
    # wenn keine Dateien geladen wurden, Ladungsträger neu verteilen
    #
    if not load_files:
        # immer die gleiche Energielandschaft? Dann jedes Mal andere Ladungsträger
        if not random_landscape:
            rnd.seed()
        # Platziere sich bewegenden Ladungsträger
        carriers_final = PlaceCarriers(N_carriers, box)
        # Globale Zeit
        global_time = 0.
        # Schleifenvariablen
        a0, a1 = 0, 0
        # Array der letzen N_j_last (=1000) Stromdichten. Über die wird gemittelt.
        j_last = np.zeros(N_j_last)
        # Array der Mittelwerte der Stromdichten, N_average Mittelwerte werden im
Abstand
        # average_step (=100) gebildet
        j_average = np.zeros(N_average)
        # wenn die Datei mit den Positionen existiert, wird sie gelöscht
        if os.path.exists(fn_car):
            os.remove(fn_car)
        # Kopfzeilen in der Ausgabedatei schreiben
        WriteOutputHeader(trial, N_carriers, eta)
    if debugmode:
        print('number of charge carriers from gate voltage', N_carriers)
        print('a0, a1:', a0, a1)
    return {'load_files': load_files, 'global_time': global_time,
            'carriers_final': carriers_final, 'morphology': morphology, 'E': E,
            'a0': a0, 'a1': a1, 'j_last': j_last, 'j_average': j_average}

def ReadCurrentdensity(trial):
    """
    Lese die Stromdichten aus der carrier-Datei aus.

    trial : int
        Nummer des Trials

    returns
    -------
    a1, j_last, j_average : int, 1D array, 1D array
        Anzahl der bisherigen Simulationsschritte, Array für die letzten N
        Stromdichten und Array für die Mittelwerte über die letzten Stromdichten
    """
    # Dateiname für die Position des Ladungsträgers
```

```
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    # Datei einlesen
    with open(fn_car, 'r') as f:
        data = f.read().split('\n')
    # alle Stromdichten
    j = [ float(d.split()[-1]) for d in data if d.startswith('# time') ]
    # entferne den ersten Stromdichtewert (null), der erste Simulationspunkt
    j = j[1:]
    # Anzahl der bisherigen Simulationsschritte
    a1 = len(j)
    # Array für die Mittelwerte der Stromdichten
    j_average = np.zeros(N_average)
    # Array der letzten Stromdichten
    j_last = np.zeros(N_j_last)
    # Anzahl der möglichen Mittelwerte für die Stromdichten
    if len(j) >= N_j_last:
        ii = int((len(j) - N_j_last + average_step) / average_step)
    # zu wenig Datenpunkte
    else:
        j_last[:len(j)] = j
        return (a1, j_last, None)
    # berechne die Mittelwerte
    for i in range(max(0, ii - N_average), ii):
        j_average[i % N_average] = np.average(j[i * average_step : i * average_step +
N_j_last])
    # Array so umschichten, dass der letzte Wert bei a1 - 1 liegt, damit es
    # an der richtigen Stelle weiter geht
    j_last[:] = np.roll(j[-N_j_last:], a1)
    return (a1, j_last, j_average)

def WriteOutputHeader(trial, N, eta):
    """
    header-Zeilen der Ausgabedatei erstellen.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    N : int
        Anzahl der Ladungsträger im Device.
    eta : float oder tuple
        Bruchteil, je nach Morphologieart
    """
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')

    # Kopfzeilen der Datei
    header = '# N_carriers ' + str(N) + '\n'
    if morphology_type == 'polymer':
        header += '# fill_eta ' + str(eta[1]) + '\n'
    elif morphology_type == 'pointgrid':
        header += '# eta_traps ' + str(eta) + '\n'
    elif morphology_type == 'crystallite':
        header += '# eta_boundary ' + str(eta) + '\n'

    # schreibe die Kopfzeilen
    with open(fn_car, 'wb') as f:
        f.write(header.encode())
    # schreibe eine dummy-Datei für den Zufallszahlengenerator
    with open(fn_rand, 'w') as f:
        f.write('')

def WriteOutput(trial, carriers, global_time, current):
    """
    Speichere die Positionen der Ladungsträger zu einem Zeitpunkt.
    Speichere den Zustand des Zufallsgenerators.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    carriers : 2D array
        Liste der Positionen der Ladungsträger
    global_time : float
        aktuelle globale Simulationszeit
    current : float
        Stromdichte in x-Richtung in A / cm**2
    """
    # Dateinamen
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')

    # kopiere die bestehenden Dateien
    shutil.copy(fn_car, fn_car + '.tmp')
    shutil.copy(fn_rand, fn_rand + '.tmp')

    # schreibe die Liste der Ladungsträger und den Zeitpunkt in die Datei
    with open(fn_car, 'ab') as f:
        header = 'time ' + str(global_time) + ' j ' + str(current)
        np.savetxt(f, carriers.transpose(), delimiter='\t', fmt='%1i', \
                   header=header)
    # speichere den Zustand des Zufallszahlengenerators
    with open(fn_rand, 'wb') as f:
        pickle.dump(rnd.get_state(), f, protocol=3)

    # bennene die temporären Dateien um
    os.remove(fn_car + '.tmp')
    os.remove(fn_rand + '.tmp')


################################################################################
# Funktionen für Parallelisierung
# -------------------------------------------------------------------------------

def SplitList(liste, size):
    """
    Teile eine Liste in Unterlisten, sodass alle etwa die gleiche Länge haben.

    liste: list
        aufzuteilende Liste
    size : int
        Anzahl der Unterlisten

    returns
```

```
    -------
    list of list
        Liste mit den Unterlisten
    """
    if not isinstance(liste, list):
        liste = list(liste)
    # Länge der einzelnen Unterlisten
    L = [int(len(liste) / size)] * size
    # die letzten Unterlisten sind um eins länger
    for i in range(len(liste)%size):
        L[-i-1] += 1
    sublists = [ liste[i:j] for i,j in zip(np.cumsum(L) - L, np.cumsum(L)) ]
    return sublists

def WorkerCarrier(E, carriers, rates, indices, childP, print_lock):
    """
    Prozess, der die Sprungraten für einige, feste Ladungsträger berechnet.
    Die Ergebnisse werden über eine Pipe an den Hauptprozess geschickt.

    V_single : 3D array
        Potential eines einzelnen Ladungsträgers
    potential : 3D array
        Coulombpotential zu Beginn der Rechnung
    E : 3D array
        Energielandschaft
    carriers : 2D array
        Array der Ladungsträger
    indices : list
        Liste mit den Indizes der Ladungsträger, um die sich dieser Prozess
        kümmern soll
    childP : pipe
        Pipe zum Datenaustausch, ist verbunden mit parentP
    """
    if debugmode:
        with print_lock:
            print('process', os.getpid(), 'started')
            print('received E[0,0,0]', E[0,0,0], '\ncarrier 0', carriers[0], \
                  '\ncarrier indices for calculation', indices)
    # Schleifenvariable für die Aktualisierung des Coulombpotentials
    a = 1
    while True:
#        t0 = time.perf_counter()
        # Sprungraten für jeden Ladungsträger berechnen
        for i in indices:
            k = CalculateRatesPerCarrier(i, carriers, E)
            rates[i::N_carriers] = k
#        t1 = time.perf_counter()
        # schicke an den Parentprozess, dass die Berechnung fertig ist
        childP.send('done')
        # neuen Ladungsträger oder Ende des Prozesses erhalten
        c, jump = childP.recv()
        if c is None:
            break
        else:
            # Position des gesprungenen Ladungsträgers aktualisieren
            carriers[c] = (carriers[c] + jump) % box
            a += 1
#            if debugmode:
#                with print_lock:
#                    print('process', os.getpid(), \
#                          '\npotential[0,0,0]', potential[0,0,0], \
#                          '\nnew carrier', carriers[c])
#        t2 = time.perf_counter()
#        with print_lock:
#            print('rates', t1 - t0)
#            print('rest', t2 - t1)

def WorkerCDM(dipoles, indices, box, gridspacing, inQ, outQ):
    """
    Räumliche Korrelation der Dipole parallel berechnen.
    """
    while True:
        pos = inQ.get()
        if pos is None:
            break

        ret = np.sum( np.sum( dipoles[tuple(np.transpose((pos + indices)%box))] \
                                        *indices*gridspacing, axis=1 ) / \
                                    np.sum((indices*gridspacing)**2, axis=1)**1.5 )
        outQ.put( (pos, ret) )

###############################################################################
# Monte-Carlo-Simulation
# ----------------------------------------------------------------------------

def MonteCarloTrial(trial):
    """
    Funktion für MonteCarlo-Trial.
    """
    # ------------------------------------------------------------------------
    # Initialisierung des Trials
    #
    dict_init = Initialize(trial)
    # wenn die Simulation schon beendet ist, abbrechen
    if dict_init is None:
        if debugmode:
            print('trial already finished')
        return
    # Variablen aus der Initialisierung übertragen
    skip_initial = dict_init['load_files']
    global_time = dict_init['global_time']
    carriers_final = dict_init['carriers_final']
    # morphology = dict_init['morphology'] # <-- wird nicht verwendet
    E = dict_init['E']
    a0 = dict_init['a0']
    a1 = dict_init['a1']
    j_last = dict_init['j_last']
    j_average = dict_init['j_average']
    # Ladungsträger in der Simulationsbox (mit periodischer Randbedingung)
    carriers = carriers_final % box
```

```
    # Anzahl der Ladungsträgersprünge in x-Richtung (down-field-hops)
    dfh = 0
    # letzter Zeitpunkt für die Berechnung der Stromdichte
    t0 = global_time
    # beim ersten Start der Simulation Positionen der Ladungsträger speichern
    if not skip_initial:
        if debugmode:
            print('initial carrier position saved')
        WriteOutput(trial, carriers_final, global_time, 0)

    # ------------------------------------------------------------------------
    # Arbeitsprozesse initialisieren
    #
    # shared memory Array für die Sprungraten
    # Index: c + j * N_carriers, c: Index Ladungsträger, j: Index Sprungrichtung
    # Im Array sind die Sprungrichtungen gruppiert, dh. nacheinander kommen die
    # Ladungsträger mit gleicher Sprungrichtung. Dadurch soll das Array absteigend
    # sortiert werden, da weitere Sprünge am Ende stehen.
    rates = mp.Array('d', N_carriers * N_jumps, lock=False)
    # Liste der Indizes der Ladungsträger
    carriers_indices = np.arange(N_carriers)
    # teile die Liste der Indizes in Unterlisten auf
    sublists = SplitList(carriers_indices, N_processes)
    # Pipes erzeugen zum Datenaustausch mit den Prozessen
    parentP, childP = zip(*[ mp.Pipe() for i in range(N_processes) ])
    # Lock für die Prozesse bei print
    print_lock = mp.Lock()
    # Prozesse erzeugen
    workers = [mp.Process(target=WorkerCarrier, \
                          args=(E, carriers, rates, sublists[i], childP[i],
print_lock))
               for i in range(N_processes)]
    # Prozesse starten
    for w in workers:
        w.start()

    # ------------------------------------------------------------------------
    # Monte-Carlo-Schleife
    #
    if debugmode:
        print('--- beginning Monte Carlo loop ---')
    while 1:
        # --- Dateiausgabe ---
        # Nach einer bestimmten Anzahl an Sprüngen Position und Zeit der Ladungsträger
speichern
        if a0 % (position_steps) == 0 and a0 > 0:
            # Dateiausgabe beim ersten Mal nach einem Neustart der Simulation
überspringen
            if skip_initial:
                skip_initial = False
            else:
                # Stromdichte (A / cm^2)
                current = dfh * 1.602e-19 / (np.prod(box) * np.prod(gridspacing[1:]))
/ (global_time - t0) * 1e23
                WriteOutput(trial, carriers_final, global_time, current)
                # die letzten N Werte der Stromdichte speichern
                j_last[a1 % N_j_last] = current
                # Alle 100 Schritte wird der Mittelwert gebildet (überschneidet sich)
                if (a1 + 1 - N_j_last) % average_step == 0 and a1 >= N_j_last - 1:
                    j_average[int((a1 + 1 - N_j_last) / average_step) % N_average] =
np.average(j_last)
                    if debugmode:
                        print('j_average', j_average)
                    # sobald das Array gefüllt wurde, Konvergenzkriterium prüfen, wenn
gewünscht
                    if a1 >= N_j_last + average_step * N_average - 1 and
check_convergence_criteria:
                        # Abbruchkriterium
                        if ConvergenceCriteria(j_average):
                            if debugmode:
                                print('convergence criteria fulfilled')
                            break
                # Zeit und Zähler für Sprünge zurücksetzen
                t0 = global_time
                dfh = 0
                # Schleifenvariable für Stromdichte erhöhen
                a1 += 1

        # Sprungraten berechnen
        CalculateRates(parentP)
        # Raten aufsummieren
        k = np.cumsum(rates)
#        if debugmode:
#            # Sprungrichtung
#            jum = 4
#            print('jump rates in', jumps[jum])
#            for i,r in enumerate(rates[jum * N_carriers : (jum + 1) * N_carriers]):
#                print('carrier', i, r)
        # Zufallszahl um ein Ereignis auszuwählen
        r = rnd.random() * k[-1]
        # Waiting time für jeden Schritt berechnen
        global_time += -np.log(rnd.random()) / k[-1]
        # Wähle das entsprechende Ereignis aus
        for i in range(N_jumps * N_carriers):
            if r < k[i]:
                # Index des Ladungsträgers
                c = i % N_carriers
                # Index der Sprungrichtung
                j = int(i / N_carriers)
                # neuen Ladungsträger an alle Prozesse schicken
                for i in range(N_processes):
                    parentP[i].send((c, jumps[j]))
#                if debugmode:
#                    print('event: carrier', c, carriers[c], 'jump', jumps[j])
                # bewege Ladungsträger
                carriers[c] += jumps[j]
                carriers_final[c] += jumps[j]
                # Berücksichtige Periodizität in x- und y-Richtung
                carriers[c,0] = carriers[c,0] % x_length
```

```
                carriers[c,1] = carriers[c,1] % y_length
                # Sprung in Feldrichtung addieren
                dfh += jumps[j,0]
                break
        # Wenn die Anzahl der Simulationsschritte überschritten ist, beenden
        if a0 >= sim_steps - 1:
            if debugmode:
                print('max simulation steps reached')
            # Stromdichte (A / cm^2)
            current = dfh * 1.602e-19 / (np.prod(box) * np.prod(gridspacing[1:])) /
(global_time - t0) * 1e23
            WriteOutput(trial, carriers_final, global_time, current)
            break
#        if debugmode and a == 0:
#            break
        a0 += 1

    # Prozesse beenden
    for i,w in enumerate(workers):
        parentP[i].send((None, None))
        w.join()

###############################################################################
# Hauptfunktion zur Steuerung des Programmablaufs
# -----------------------------------------------------------------------------

def Main():
    """
    Hauptfunktion. Starte die einzelnen Trials.
    """
    # Starte Monte-Carlo-Trials
    s = time.perf_counter()
    for trial in range(number_trials):

        MonteCarloTrial(trial)
        print('Trial', trial, 'finished.')

        # Lösche die Dateien der Morphologie und Energielandschaft
        if delete_data:
            fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) +
'.npy')
            fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
            fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) +
'.pkl')
            if os.path.exists(fn_morph):
                os.remove(fn_morph)
            if os.path.exists(fn_E):
                os.remove(fn_E)
            if os.path.exists(fn_rand):
                os.remove(fn_rand)
    e = time.perf_counter()
    print('runtime all trials', e - s)

###############################################################################
# Kommandozeilen-Optionen festlegen
# -----------------------------------------------------------------------------

# ---------------------------------
# mögliche Kommandozeilen-Optionen:
# ---------------------------------
#   Nummer des Parametersatzes: Zahl, muss erste Option sein
#   alle weiteren Optionen sind vom Format key=value
#   Anzahl der Prozesse zur Parallelisierung mit 'cores='

# Werden Optionen übergeben? Ist der Fall, wenn auf dem Cluster gerechnet wird.
cluster_rechnen = len(sys.argv) != 1

# Anzahl der Prozesse auf dem lokalen Rechner
N_processes = 4
# Anzahl der Knoten, default 1
N_nodes = 1

if cluster_rechnen:
    for option in sys.argv[2:]:
        key, value = option.split('=')
        # Anzahl der Prozesse
        if key == 'cores':
            N_processes = int(value)
#        elif key == 'nodes':
#            N_nodes = int(value)
        else:
            raise IOError('Kommandozeilenoption ' + str(option) + \
                          ' nicht bekannt.')

# -----------------------------------------------------------------------------
# Nummer des Parametersatzes
#

if cluster_rechnen:
    # Nummer des Parametersatzes ermitteln
    parametersatz = int(sys.argv[1])
else:
    # Testordner auf lokalem Rechner
    folder_local = 'C:/Daten/TobiasMeier/Simulationen/Tests'
    # wechsele in das lokale Verzeichnis
    os.chdir(folder_local)
    # Nummer des Parametersatzes
    parametersatz = 1

###############################################################################
# Parameter aus Datei einlesen
# -----------------------------------------------------------------------------

# Dateinamen für den Parametersatz
fn_params = os.path.join('Parameters' + str(parametersatz), \
                         'parameters' + str(parametersatz) + '.asc')

# Dictionary aus der Parameterdatei lesen
d = LoadParameters(fn_params)
```

```
# Gitterabstände (nm)
dx = d['dx']
dy = d['dy']
dz = d['dz']

# Boxlänge (Anzahl der Gitterplätze)
x_length = int(d['x_length'])
y_length = int(d['y_length'])
z_length = int(d['z_length'])

# Anzahl der Trials
number_trials = int(d['number_trials'])

# Soll eine Morphologie für alle Trials geladen werden? Muss sich im Hauptordner
befinden
load_morphology = d['load_morphology']

# Soll die Energielandschaft zufällig sein? (sonst immer wie Trial 0)
random_landscape = d['random_landscape']

# maximale Simulationszeit (ns)
sim_steps = d['sim_steps']

# soll das Konvergenzkriterium für die Stromdichte verwendet werden?
check_convergence_criteria = d['current_conv']

# ------------------------------------------------------------------------------
# Eigenschaften der Morphologie

# Art der Morphologie
morphology_type = d['morphology']
# Ist die erste Schicht ausschließlich mit Fallen besetzt?
trap_layer = d['trap_layer']

if morphology_type == 'pointgrid':
    # Fallenverteilungsfunktion
    trap_distribution = d['trap_distribution']
    # Sollen sich die Fallen nur in der Grenzschicht befinden?
    interface_traps = d['interface_traps']
    # Dichte der Fallenzustände
    trapdensity = d['trapdensity']
    # Kettenlänge
    chain_length = int(d['chain_length'])
    # Breite der Verteilung der Kettenlänge
    chain_sigma = d['chain_sigma']
    # fester Versatz zwischen Ketten
    chain_offset = d['chain_offset']
    if chain_offset != 'random':
        chain_offset = int(chain_offset)

# ------------------------------------------------------------------------------
# Elektrische und elektronische Eigenschaften
#
# Verwendetes Energiemodell
energy_model = d['energy_model']
# Breite der Energieverteilung sigma (eV)
sigma = d['sigma']
# Zentrum der Energieverteilung der Fallenzustände (eV)
E_trap = d['E_trap']
# Breite der Energieverteilung der Fallenzustände (eV)
sigma_trap = d['sigma_trap']
# Dehnungsfaktor für den exponentiellen Anteil gegenüber dem Gaußanteil
# sigma_exp = stretch_exp * sigma_trap
if trap_distribution == 'gaussexp':
    stretch_exp = d['stretch_exp']
# Dicke des Gate-Dielektrikums (nm)
d_gate = d['d_gate']
# relative Dielektrizitätskonstante des Gate-Dielektrikums
epsilon_gate = d['epsilon_gate']
# Gate-Spannung (V)
U_Gate = d['U_Gate']
# Drain-Spannung (V)
U_Drain = d['U_Drain']
# relative Dielektrizitätskonstante
epsilon_rel = d['epsilon_rel']

# ------------------------------------------------------------------------------
# Parameter für Sprünge
#
# Temperatur
temperature = d['temperature']
# Hüpfmechanismus
hopping = d['hopping']
# Miller-Abrahams
if hopping == 'MA':
    # Attempt-to-hop frequency (1/ns)
    nu0 = d['nu0']
# Marcus-Rate
elif hopping == 'marcus':
    # Reorganisationenergie
    reorg_energy = d['reorg_energy']
    # Transferintegral
    J0 = d['J0']
else:
    raise ValueError('unknown hopping mechanism')
# inverse Lokalisierungskonstanten(1/nm)
# innerhalb einer Polymerkette
gamma_intra = d['gamma_intra']
# Kopplungskonstante in x-Richtung
gamma_x = d['gamma_x']
# Kopplungskonstante in y-Richtung
gamma_y = d['gamma_y']
# Kopplungskonstante in z-Richtung
gamma_z = d['gamma_z']
# Anzahl der Nächsten Nachbarn für Sprünge
nearest_neighbors = int(d['NN_hops'])

##############################################################################
# Konstanten und globale Variablen
# ------------------------------------------------------------------------------
```

```
# ------------------------------------------------------------------------------
# Konstanten
#
# kT (eV)
kT = 8.617e-5 * temperature
# Cut-off Radius für die Coulomb-Wechselwirkung (nm)
#cutoff_Coulomb = 100
# Cutoff-Radius für die Dipolwechselwirkung bei räumlich korrelierter Energie (nm)
cutoff_dipoles = 25
# Faktor für Coulomb-Wechselwirkung (in eV)
faktor_Coulomb = 1.602e-19 / 4 / np.pi / 8.854e-12 / epsilon_rel * 1e9
# Potentialunterschied bei einem Schritt in x-Richtung
voltage_difference = U_Drain / x_length
# Potentialunterschied bei einem Schritt in z-Richtung durch die Gate-Spannung
gate_potential_step = epsilon_gate / epsilon_rel * abs(U_Gate) / d_gate * dz
# Anzahl der Ladungsträger durch den Kondensator
N_carriers = int(epsilon_gate * 8.854e-12 * x_length * y_length * dx * dy *
abs(U_Gate) \
                 / 1.602e-19 / d_gate * 1e-9)

# Konstanten für Marcus-Rate
if hopping == 'marcus':
    C1_marcus = 2 * np.pi / 6.582e-16 * J0**2 / np.sqrt(4 * np.pi * kT * reorg_energy)
* 1e-9
    C2_marcus = 4 * reorg_energy * kT

# ------------------------------------------------------------------------------
# Globale Variablen
#
# Array für Größe der Simulationsbox
box = np.array([x_length, y_length, z_length])
# Array für Abstände der Gitterplätze
gridspacing = np.array([dx, dy, dz])
# Inverse Lokalisierungskonstanten
gamma = np.array([gamma_x, gamma_y, gamma_z])
# mögliche Sprungrichtungen (globale Variable)
jumps = PossibleJumps(nearest_neighbors, 3)
# Anzahl der Sprungrichtungen (globale Variable)
N_jumps = len(jumps)

# ------------------------------------------------------------------------------
# Parameter für die Dateiausgabe
#

# Verzeichnis des aktuellen Parametersatzes
folder_out = 'Parameters' + str(parametersatz)

# Anzahl der Schritte nach denen das Potential neu berechnet werden soll
update_potential_steps = 10000
# Anzahl der Sprünge, nach denen die Position gespeichert werden soll
position_steps = 10000
# Anzahl der Stromdichten über die der Mittelwert bestimmt werden soll
N_j_last = 1000
# Anzahl der Mittelwerte an Stromdichten, die für die Konvergenz benutzt wird
N_average = 10
# Abstand zwischen den Stromdichtemittelwerten
average_step = 100
# Konvergenzkriterium für die Stromdichte
conv_current = 0.05

################################################################################
# starte Simulation mit Parallelisierung
# ------------------------------------------------------------------------------

if __name__ == '__main__':
    Main()

# ------------------------------------------------------------------------------
# Programmende
################################################################################
```

A.2.3. Simulationsprogramm zur Vielteilchensimulation Teil 2

```
# -*- coding: utf-8 -*-
"""
Simulation von Ladungstransport durch OFET.

Charakteristische Merkmale:

    - Mehrteilchensimulation
    - korrelierte Energielandschaft möglich
    - exakte Coulombwechselwirkung zwischen den Ladungsträgern und periodischen
      Fortsetzungen in den angrenzenden Simulationsboxen.
    - Orthorhombisches Gitter (unterschiedliche Gitterabstände) mit periodischen
      Randbedingungen in x- und y-Richtung.
      In z-Richtung beschränkt.
    - Gitterplätze werden zu Beginn zufällig mit Ladungsträgern befüllt entsprechend
      der Kapazität des Gatedielektrikums.
    - Miller-Abrahams Sprungraten.
    - Energielandschaft wird für jeden Trial neu initialisiert.
    - Frühere Simulation kann fortgesetzt werden.
    - Parallelisierung mit multiprocessing: bezüglich der Ladungsträger
    - Nachdem ein Trial fertig ist, kann die Datei für die Energielandschaft
      gelöscht werden.
    - Coulombpotential wird an jedem Punkt berechnet und nach jedem Sprung
      aktualisiert.
    - Trials sind reproduzierbar in den Zufallszahlen.
    - Der Zufallszahlengenerator für die Energielandschaft und die Ladungsträger
      können entkoppelt werden in der Parameterdatei. Bei Entkopplung wird immer
      die Energielandschaft von Trial 0 erzeugt.
```

```
    - Eine Morphologie kann geladen werden für alle Trials. Die Datei muss
      morphology.npy heißen und sich im Hauptordner befinden.
    - Beim Schreiben der output-Dateien wird zuerst eine temporäre Sicherungsdatei
angelegt,
      dann die ursprüngliche verändert und dann die Sicherungskopie gelöscht.

Dateiausgabe:

    - Positionen und Zeitpunkt aller Ladungsträger nach jedem 10 * N_carriers Sprung.
"""
import numpy as np
import numpy.random as rnd
import os
import itertools
import multiprocessing as mp
import sys
import time
import pickle
import shutil

# Modus für Fehlersuche
debugmode = True
# Dateien für Morphologie und Energielandschaft speichern?
# Kann sinnvoll sein bei großen Dateien diese nicht zu speichern
save_data = True
# Morphologie und Energielandschaft nach Beendigung eines Trials löschen?
delete_data = True

################################################################################
# Funktionendefinition
# ------------------------------------------------------------------------------

# ------------------------------------------------------------------------------
# Energielandschaft
#
def Energylandscape(box, sigma, morphology, model='GDM', gatevoltage=True):
    """
    Energielandschaft erstellen.

    box : 1D array
        Größe der Simulationsbox
    sigma : float
        gewünschte Standardabweichung der Energielandschaft (eV)
    morphology : 3D array
        Morphologie
    model : str
        Gewünschtes Modell. Mögliche Modelle: gaussian disorder model ('GDM'),
        correlated disorder model ('CDM').
    gatevoltage : bool
        Soll das Gatepotential zur Energielandschaft dazuaddiert werden?

    returns
    -------
    E : 3D array
        Energiewerte für jeden Gitterplatz nach gewünschtem Modell.
    """
    if model == 'GDM':
        E = GDM(box, sigma, morphology)
    elif model == 'CDM':
        E = CDM(box, sigma)
    else:
        raise ValueError('model for energylandscape unknown:', model)

    if gatevoltage:
        # Potential durch Gateelektrode dazuaddieren
        gate_potential = np.empty(box)
        gate_potential[:] = np.linspace(0, gate_potential_step*(box[2]-1), \
                                        num=box[2]).reshape(1,1,-1)
        # in der Fallenschicht einen halben Potentialsprung dazuaddieren
        if trap_layer:
            gate_potential[:,:,0] += gate_potential_step / 2
        E += gate_potential
    return E

def GDM(box, sigma, morphology):
    """
    Gaussian disorder model.

    Erstelle eine Energielandschaft mit statischen Unordnung. Die Energien der
    Gitterplätze sind einer Gaußverteilung entnommen. Morphologie wird
    berücksichtigt.

    box : 1D array
        Größe der Simulationsbox.
    sigma : float
        Standardabweichung der Verteilung der Energien.
    morphology : 3D array
        Morphologie

    returns
    -------
    E : 3D array
        Energielandschaft der Größe `box` nach dem GDM.
    """
    # Nummern aller Ketten und deren Indizes
    numbers, indices = np.unique(morphology, return_inverse=True)
    # Energiewerte für jede Kettennummer
    E_reduced = rnd.normal(loc=0, scale=sigma, size=len(numbers))
    # rekonstruiere die Morphologie
    E = E_reduced[indices].reshape(box)
    # Positionen der Fallenzustände
    traps = np.where(morphology <= -1)
    # -------------------------------
    # Energiewerte für Fallenzustände
    #
    # Gaußverteilung
    if trap_distribution == 'gauss':
        E_traps = rnd.normal(loc=E_trap, scale=sigma_trap, size=len(traps[0]))
    elif trap_distribution == 'exp':
        E_traps = E_trap - rnd.exponential(scale=sigma_trap, size=len(traps[0]))
```

```
    elif trap_distribution == 'gaussexp':
        size_gauss = int(len(traps[0]) * np.sqrt(np.pi) / (stretch_exp * np.sqrt(2) +
np.sqrt(np.pi)))
        size_exp = len(traps[0]) - size_gauss
        gauss_traps = []
        while len(gauss_traps) < size_gauss:
            x = rnd.normal(loc=E_trap, scale=sigma_trap)
            if x >= E_trap:
                gauss_traps.append(x)
        E_traps = np.append(gauss_traps, E_trap -
rnd.exponential(scale=stretch_exp*sigma_trap, size=size_exp))
    else:
        raise ValueError('trap distribution unknown', trap_distribution)
    # Energiewerte den Plätzen zuweisen
    E[traps] = E_traps
    return E

def CDM(box, sigma):
    #TODO Morphologie für CDM berücksichtigen
    """
    Correlated disorder model.

    Energielandschaft mit räumlicher Korrelation erstellen.
    Nach Phys Rev B 83, 085206 (2011), Bumblebee

    box : 1D array
        Größe der Simulationsbox
    sigma : float
        gewünschte Standardabweichung der Energielandschaft (eV)

    returns
    -------
    E : 3D array
        Energielandschaft mit räumlicher Korrelation.
    """
    # Array für Energielandschaft
    E = np.empty(box)
    # Stärke der Dipole auf den Gitterplätzen aus Standardabweichung sigma berechnen
    dipole_strength = sigma*8.854e-12*epsilon_rel*np.min(gridspacing)**2*1e-9 /
(2.35*1.602e-19)

    # Nummern aller Ketten und deren Indizes
#    numbers, indices = np.unique(morphology, return_inverse=True)

    # zufällig Dipole anordnen
    dipoles = RandomDipoles(box, dipole_strength)

    # Gitterpunkte einer Kugel mit dem Cutoffradius ohne den Punkt [0,0,0]
    indices = SphereIndices(cutoff_dipoles, gridspacing)

    # Liste aller Gitterpositionen
    x, y, z = np.mgrid[0:box[0], 0:box[1], 0:box[2]]
    pos_list = np.stack((x, y, z), axis=3).reshape(-1,3)

    # Parallelisierung mit multiprocessing
    inQ = mp.Queue()
    outQ = mp.Queue()
    # Prozesse initialisieren und starten
    workers = [mp.Process(target=WorkerCDM, args=(dipoles, indices, box, gridspacing,
inQ, outQ))
               for i in range(N_processes)]
    for w in workers:
        w.start()
    print('Prozesse gestartet')

    # Energie an jedem Gitterpunkt berechnen
    for p in pos_list:
        inQ.put(p)
    # auf Ergebnisse warten
    for i in range(len(pos_list)):
        pos, val = outQ.get()
        E[tuple(pos)] = val
    # Prozesse beenden
    for i in range(N_processes):
        inQ.put(None)
    # warte auf das Ende aller Prozesse
    for w in workers:
        w.join()

    # Energie skalieren in eV
    E *= -1.602e-19/(8.854e-12*epsilon_rel*1e-9)
    if debugmode:
        print('std von E', np.std(E))
    return E

def RandomDipoles(box, dipol_strength):
    """
    Gitter mit zufällig orientierten Dipolen füllen.

    box : 1D array
        Größe der Simulationsbox
    dipol_strength : float
        Stärke des Dipols (in Einheit der Elementarladung, nm)

    returns
    -------
    dipoles : 4D array
        Index 0: x-Koordinate, Index 1: y-Koordinate, Index 2: z-Koordinate,
        Index 3: Richtung des Dipols [x, y, z]
    """
    dipoles = np.empty((box[0], box[1], box[2], 3))
    phi = rnd.random(size=np.prod(box))*2*np.pi
    theta = rnd.random(size=np.prod(box))*np.pi
    dipoles[:,:,:,0] = np.reshape(dipol_strength*np.sin(theta)*np.cos(phi), box)
    dipoles[:,:,:,1] = np.reshape(dipol_strength*np.sin(theta)*np.sin(phi), box)
    dipoles[:,:,:,2] = np.reshape(dipol_strength*np.cos(theta), box)
    return dipoles

def SphereIndices(R, gridspacing, include_origin=False):
    """
```

```
    Ermittle die Koordinaten, die innerhalb einer Kugel mit festem Radius
    liegen. Der Kugelmittelpunkt liegt dabei auf der Null.

    R : float
        Radius der Kugel
    gridspacing : 1D array
        Gitterabstände, [dx, dy, dz]
    include_origin : bool
        Soll der Ursprungspunkt [0,0,0] mit in der Liste enthalten sein?

    returns
    -------
    indices : 2D array
        Liste der Koordinaten, die innerhalb der Kugel liegen. Index 0: Punkte,
        Index 1: [x,y,z]
    """
    # Anzahl der maximal möglichen Gitterpunkte in einer Richtung
    n = int(np.max(R/gridspacing))
    indices = np.array(list(itertools.product(range(-n,n+1), repeat=3)))
    length = np.sqrt(np.sum((indices*gridspacing)**2, axis=1))
    indices = indices[np.where(length <= R)]
    if not include_origin:
        indices = indices[np.any(np.not_equal(indices, [0,0,0]), axis=1)]
    return indices

# ------------------------------------------------------------------------------
# Morphologie erstellen
#
def CreateMorphology(morphology_type):
    """
    Erstelle die Morphologie
    - Es können verschiedene Schichten aneinandergefügt werden.
    - Die Ausrichtung der Ketten kann in jeder Schicht individuell eingestellt
      werden.
    - Fallen können in verschiedenen Schichten hinzugefügt werden.
    - Negative Zahlen sind Fallen, positive Polymersegmente. Null bedeutet leerer
      Platz und wird somit außer bei trap_layer nicht verwendet.

    box : 1D array
        Größe der Simulationsbox.
    morphology_type : str
        Art der zu erstellenden Morphologie
    returns
    -------
    morphology, eta_traps : 3D array, float
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Kette gehören.
        eta_traps ist die tatsächliche Fallendichte in der Schicht, in der
        Fallen vorhanden sind.
    """
    if morphology_type == 'pointgrid':
        if interface_traps:
            # Trapstates nur in der ersten Schicht
            interface, eta = CreatePointgrid(np.array([box[0], box[1], 1]),
chain_length, chain_sigma, chain_offset, 0, trapdensity)
            # Restliche Morphologie ohne Fallen
            morphology, e = CreatePointgrid(np.array([box[0], box[1], box[2] - 1]),
chain_length, chain_sigma, chain_offset, 0)
            # Kettennummern eindeutig machen
            morphology += np.max(interface) + 1
            morphology = np.concatenate((interface, morphology), axis=2)
        else:
            morphology, eta = CreatePointgrid(box, chain_length, chain_sigma,
chain_offset, 0, trapdensity)
    return morphology, eta

# ------------------------------------------------------------------------------

def CreatePointgrid(box, chain_length, chain_sigma, chain_offset, orientation,
trapdensity=0., periodic=True):
    """
    Erstelle einen Bereich für die Morphologie, in dem alle Ketten in einer
    Richtung verlaufen.
    - Es werden Gitterpunkte zu Ketten verbunden. Dabei werden alle Gitterpunkte
      durchnummeriert, wobei Gitterpunkte einer Kette die gleiche Nummer erhalten.
    - Es muss nicht jede Zahl als Nummer einer Kette vorkommen.
    - Kann verwendet werden, um mehrere solcher Kästen aneinanderzureihen.
    - Optionalen Parameter, wenn Ketten periodisch sein sollen.
    - Fallenzustände haben negative Nummern, jeder seinen eigenen Wert.

    box : 1D array
        Größe der Morphologie
    chain_length : int
        Länge der Ketten
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)
    trapdensity : float
        Wahrscheinlichkeit, dass ein Gitterplatz ein Fallenzustand ist
    periodic : bool
        Sollen die Ketten über die Ränder periodisch fortgesetzt werden?

    returns
    -------
    morphology, eta_traps : 3D array, float
        Array mit den Nummern der Ketten. Gleiche Zahl bedeutet, dass die
        Gitterplätze zur gleichen Polymerkette gehören. Ein Wert von -1 bedeutet
        ein Fallenzustand.
        eta_traps ist der Bruchteil der Gitterplätze mit Fallen. Wenn keine
        vorhanden sind, wird None zurückgegeben.
    """
    box = np.asarray(box)
    # die Richtung, in der die Ketten durchnummeriert werden, ist die Achse
    # der Orientierung. Vertausche die Achsen so, dass die Achse der Kettenausrichtung
    # die letzte Achse ist.
    shape = box[np.roll(np.arange(3), 2 - orientation)]
    # Achsen, die wieder zurück vertauscht werden müssen
```

```
    transpose = np.roll(np.arange(3), orientation - 2)

    # wenn keine periodische Randbedingung für die Kette gelten soll, dann eine
    # Kette mehr mitnehmen
    if not periodic:
        shape[2] += chain_length

    # -----------------------------------------------------------------------
    # Polymerketten mit bestimmter Länge erstellen
    #
    # Array für die Morphologie
    morphology = np.zeros(shape, dtype=np.int)
    # Laufindex der Kettennumer
    k = 1
    # die zwei Dimensionen durchlaufen, die nicht der Kettenausrichtung entsprechen
    for i in range(shape[0]):
        for j in range(shape[1]):
            # ein Kettenstrang (1D)
            line = []
            # Füge solange Segmente hinzu, bis der Strang voll ist
            while True:
                # Länge der neuen Ketten aus Verteilung bestimmen
                l = int(rnd.normal(loc=chain_length, scale=chain_sigma) + 0.5)
                # kann diese Kettenlänge noch hinzugefügt werden?
                if len(line) + l <= shape[2]:
                    line.extend([k]*l)
                    # Kettenindex erhöhen
                    k += 1
                else:
                    line.extend([k]*(shape[2] - len(line)))
                    # Strang zur Morphologie hinzufügen
                    morphology[i,j] = line
                    # Kettenindex erhöhen
                    k += 1
                    break

    # -----------------------------------------------------------------------
    # Fallenzustände einfügen
    #
    # Laufindex für die Fallenzustände
    trap = -1
    # Alle Gitterplätze werden durchlaufen
    for i in range(shape[0]):
        for j in range(shape[1]):
            for k in range(box[orientation]):
                # zufällig eine Falle platzieren
                if rnd.random() < trapdensity:
                    # Wenn eine Kette auseinandergerissen werden würde,
                    # wird die Falle nach hinten verschoben
                    x = k
                    while morphology[i, j, x-1] == morphology[i, j, x]:
                        x += 1
                        if x >= box[orientation]:
                            break
                    # Falle einbauen
                    if x < box[orientation]:
                        morphology[i, j] = np.append(morphology[i, j, :x], \
                                  np.append(trap, morphology[i, j, x:-1]))
                        # Laufindex für Fallen erniedrigen
                        trap -= 1
    # Versatz der Ketten nebeneinander
    offset = ChainOffset(box, chain_length, chain_offset, orientation)
    # Verschiebe die Kettennummern entsprechend dem Versatz
    for i in range(shape[0]):
        for j in range(shape[1]):
            morphology[i,j,:] = np.roll(morphology[i,j,:], -offset[i,j])
    # wenn keine periodischen Ketten, dann auf die ursprüngliche Länge kürzen
    if not periodic:
        morphology = morphology[:,:,:box[orientation]]
    # mittlere Kettenlänge bestimmen
    numbers, counts = np.unique(morphology, return_counts=True)
    if debugmode:
        print('average chain length', '{:.3g}'.format(np.average(counts)), '+-', \
          '{:.3g}'.format(np.std(counts)))
    # Anzahl der Fallen
    counts = np.where(morphology < 0)
    # Fallendichte im Bereich
    if len(counts[0]) > 0:
        eta_traps = len(counts[0])/np.prod(box)
    else:
        eta_traps = None
    # Achsen wieder zurücktauschen
    return np.transpose(morphology, axes=transpose), eta_traps

def ChainOffset(box, chain_length, chain_offset, orientation):
    """
    Erstelle zufällige Startpunkte der Ketten entlang der Richtung, in der sie
    ausgerichtet sind.

    box : 1D array
        Größe der Simulationsbox.
    chain_length : int
        Länge der Polymerketten.
    chain_offset : 'random' oder integer
        Sollen die Ketten einen zufälligen, keinen oder einen bestimmten
        Versatz haben.
    orientation : int
        Richtung, in der die Ketten ausgerichtet sein sollen. Entspricht der
        Achse des Arrays (0=x, 1=y, 2=z)

    returns
    -------
    chain_offset : 2D array
        Array der Offsets für die Polymerketten. Index 0, 1: Koordinaten ohne
        der, in der die Ketten ausgerichtet sind, z.B. y und z für Ketten in
        x-Richtung
    """
    # zyklisches Vertauschen der Dimensionen, sodass orientation die letzte Achse ist
    size = box[np.roll(np.arange(3), 2 - orientation)]
    if chain_offset == 'random':
```

```
        return rnd.randint(chain_length, size=size[:-1])
    # Ketten werden in die beiden übrigen Richtungen um festen Wert jeweils zueinander
verschoben
    else:
        offset0 = np.repeat(np.arange(size[1]).reshape((1,-1)), size[0],
axis=0)*chain_offset
        offset1 = np.repeat(np.arange(size[0]).reshape((-1,1)), size[1],
axis=1)*chain_offset
        return offset0 + offset1

# ------------------------------------------------------------------------------
# Sprungraten berechnen
#
def CalculateRates(parentP):
    """
    Wartet nur darauf, dass alle Childprozesse mit der Berechnung der Sprungraten
    fertig sind.

    parentP : list of pipes
        Liste der Pipes zum Datenaustausch mit den Prozessen,
        ist verbunden mit childP[i]
    """
    # warte darauf, dass alle Prozesse fertig sind
    for i in range(N_processes):
        parentP[i].recv()

def CalculateRatesPerCarrier(c, carriers, E, potential):
    """
    Berechnet die Sprungraten für einen Ladungsträger in die verschiedenen Richtungen.

    c : int
        Index des Ladungsträgers
    carriers : 2D array
        Array mit den aktuellen Ladungsträgern
    E : 3D array
        Array der Energielandschaft
    potential : 3D array
        Coulomb-Potential an jedem Punkt

    returns
    -------
    k : 1D array, Länge = len(jumps)
        Sprungraten für einen Ladungsträger in alle Sprungrichtungen. Die Indizes
        entsprechen denen der Liste für die Sprungrichtungen.
    """
    k = np.empty(N_jumps)

    for j,jump in enumerate(jumps):
        # initial und final Position
        newpos = carriers[c] + jump
        # periodische Randbedingung in x,y-Richtung
        newpos[0] = newpos[0]%box[0]
        newpos[1] = newpos[1]%box[1]
        # Sprünge in z-Richtung sind beschränkt wegen nicht-periodischer Randbedingung
        if newpos[2] >= box[2]:
            k[j] = 0
        elif newpos[2] < 0:
            k[j] = 0
        # Sprünge auf einen schon besetzten Platz sind nicht möglich
        elif np.any(np.all(newpos == carriers, axis=1)):
            k[j] = 0
        # Sprungraten berechnen
        else:
            init = tuple(carriers[c])
            final = tuple(newpos)
            # Energie am Startgitterplatz
            E_init = E[init] + potential[init]
            # Energie am Zielgitterplatz
            E_final = E[final] + potential[final] + jump[0] * voltage_difference +
correction_potential[j]
            r_gamma = np.sqrt(np.sum((jump * gridspacing * gamma)**2))
            k[j] = Hoppingrate(r_gamma, E_init, E_final)

    return k

def Hoppingrate(r_gamma, E_i, E_f):
    """
    Berechne Hüpfrate nach Marcus oder Miller-Abrahams.

    r_gamma : float
        Abstand zwischen den Gitterplätzen multipliziert mit Kopplungskonstante.
    E_i : float
        Energie am Startgitterplatz
    E_f : float
        Energie am Zielgitterplatz

    returns
    -------
    für Marcus-Rate:
    k = 2*pi / hbar * |J_ij|**2 * np.sqrt(1 / (4*pi*kB*T*lambda)) *
        np.exp(-(E_j - E_i + lambda)**2 / (4*lambda*kB*T))
    """
    if hopping == 'MA':
        if E_i >= E_f:
            return nu0 * np.exp(-2 * r_gamma)
        else:
            return nu0 * np.exp(-2 * r_gamma) * np.exp((E_i - E_f)/kT)
    else:
        return C1_marcus * np.exp(-2 * r_gamma) * np.exp(-(E_f - E_i +
reorg_energy)**2 / C2_marcus)

# ------------------------------------------------------------------------------
# Coulomb-Potential
#
def CalculatePotential(V_single, carriers):
    """
    Berechne das Coulomb-Potential an jeder Position, das durch die Ladungsträger
    verursacht wird.

    V_single : 3D array
```

```
        Potential eines Ladungsträgers, inklusive optionaler periodischer Fortsetzung
        shape = (2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1)
    carriers : 2D array
        Array mit den Positionen der Ladungsträger.

    returns
    -------
    : float
        Coulomb-Potential in eV.
    """
    V = np.zeros(box)
    for c in carriers:
        V += V_single[box[0] - c[0] : 2*box[0] - c[0], \
                      box[1] - c[1] : 2*box[1] - c[1], \
                      box[2] - 1 - c[2] : 2*box[2] - 1 - c[2]]
    return V

def UpdatePotential(V_single, potential, old_pos, jump):
    """
    V_single : 3D array
        Potential eines einzelnen Ladungsträgers.
    potential : 3D array
        Aktuelles Coulombpotential
    old_pos : 1D array
        Ehemalige Position des Ladungsträgers
    jump : 1D array
        Sprungrichtung

    returns
    -------
    : 3D array
        Potential nach dem Sprung des Ladungsträgers
    """
    # Potential an der ehemaligen Stelle abziehen
    potential -= V_single[box[0] - old_pos[0] : 2*box[0] - old_pos[0], \
                          box[1] - old_pos[1] : 2*box[1] - old_pos[1], \
                          box[2] - 1 - old_pos[2] : 2*box[2] - 1 - old_pos[2]]
    # neue Position
    new_pos = (old_pos + jump) % box
    # Potential an der neuen Stelle dazuaddieren
    potential += V_single[box[0] - new_pos[0] : 2*box[0] - new_pos[0], \
                          box[1] - new_pos[1] : 2*box[1] - new_pos[1], \
                          box[2] - 1 - new_pos[2] : 2*box[2] - 1 - new_pos[2]]
    return potential

def SingleCarrierPotential():
    """
    Berechne das Potential eines Ladungsträgers mit periodischer Fortsetzung.
    Der Ladungsträger wird einmal in beide x-Richtungen und einmal in beide
    y-Richtungen kopiert, sowie in die vier diagonalen Ecken. Das Potential
    wird an jedem Punkt berechnet, wobei immer der Abstand zur nächstliegenden
    Ladung genommen wird. Der Ladungsträger erzeugt somit an keinem Punkt
    mehrmals ein Potential durch periodische Kopien.

    returns
    -------
    : 3D array
        Potential an jeder Stelle, shape = (2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1)
    """
    # Doppelte Boxgröße
    x2, y2, z2 = np.meshgrid(np.arange(2*box[0] + 1), np.arange(2*box[1] + 1),
np.arange(2*box[2] - 1), \
                         indexing='ij')
    # Potential
    V_single = np.zeros((2*box[0] + 1, 2*box[1] + 1, 2*box[2] - 1))

    # Liste mit den Positionen der Ladungsträger (periodisch)
    carrier_list = np.array([[0, 0, box[2] - 1], [box[0], 0, box[2] - 1], [box[0] * 2,
0, box[2] - 1],
                     [0, box[1], box[2] - 1], [box[0], box[1], box[2] - 1], [box[0] *
2, box[1], box[2] - 1],
                     [0, box[1] * 2, box[2] - 1], [box[0], box[1] * 2, box[2] - 1],
[box[0] * 2, box[1] * 2, box[2] - 1]])
    # berechne das Potential eines jeden Ladungsträgers an der Stelle, an der es
größer als der bisherige Wert ist
    for carrier in carrier_list:
        # Potential eines Ladungsträgers in der gesamten Box
        with np.errstate(divide='ignore'):
            V = faktor_Coulomb / np.sqrt(
                ((x2 - carrier[0]) * gridspacing[0])**2 + ((y2 - carrier[1]) *
gridspacing[1])**2 +
                ((z2 - carrier[2]) * gridspacing[2])**2)
        # Bereich, in dem das neue Potential größer ist als das bisherige
        rng = V > V_single
        V_single[rng] = V[rng]
    # setze das Potential an den Stellen der Ladungsträger null
    V_single[tuple(carrier_list.transpose())] = 0

    return V_single

def CorrectionPotential():
    """
    Jeder Ladungsträger verursacht ein Potential. Bei der Berechnung der Sprungraten
    müsste der Ladungsträger in seinem eigenen Potential springen. Deswegen
    muss dieser Beitrag korrigiert werden.
    Der Korrekturwert wird einfach draufaddiert.

    returns
    -------
    : 1D array
        Korrekturwerte, Index entspricht der Sprungrichtung im Array jumps.
    """
    corr = -faktor_Coulomb/np.sqrt(np.sum((jumps * gridspacing)**2, axis=1))
    return corr

# ------------------------------------------------------------------------------

def PlaceCarriers(N, box, x=None, y=None, z=None):
    """
```

```
    Ein bestimmter Bruchteil der Gitterplätze wird zufällig mit Ladungsträgern
gefüllt.

    N : int
        Anzahl der zu platzierenden Ladungsträger
    box : 1D array
        Array der Boxgröße
    x,y,z = None : int
        Wenn angegeben ist, dann wird diese Koordinate festgelegt

    returns
    -------
    2D array
        Array der besetzten Gitterplätze
    """
    # Dimensionen
    dim = len(box)
    # Array der Ladungsträger
    c = np.empty((0, dim), dtype=np.int)
    # Anzahl bereits platzierte Ladungsträger
    i = 0
    while i < N:
        p = np.array(rnd.random(dim)*box, dtype=np.int)
        # setze eine Koordinate fest, wenn gewünscht
        if x is not None:
            p[0] = x
        if y is not None:
            p[1] = y
        if z is not None:
            p[2] = z
        # überprüfe, ob neue Position schon besetzt ist
        if not np.any(np.all(c == p, axis=1)):
            c = np.vstack((c,p))
            i += 1
    return c

def PossibleJumps(NN, dim):
    """
    Ermittle mögliche Sprungrichtungen.

    NN : int
        Anzahl nächster Nachbarn, bei ungleichen Gitterabständen sollte diese
        höher eingestellt werden.
    dim : int
        Anzahl der Dimensionen

    returns
    -------
    2D array
        Array mit den möglichen Sprungrichtungen, sortiert nach steigendem
Sprungabstand
    """
    all_jumps = np.array(list(itertools.product(range(-NN,NN+1), repeat=dim)))
    jump_length = np.sqrt(np.sum(np.square(all_jumps*gridspacing), axis=1))
    jump_length_unique, counts = np.unique(jump_length, return_counts=True)
    return all_jumps[np.argsort(jump_length)[1:np.sum(counts[1:NN+1])+1]]

def ConvergenceCriteria(j_average):
    """
    Kriterium dafür, dass die Stromdichte konvergiert ist. Standardabweichung
    der Mittelwerte muss kleiner als ein Grenzwert sein (10 %)

    j_average : 1D array
        Array mit den Mittelwerten der Stromdichten

    returns
    -------
    : bool
        Kriterium erfüllt oder nicht.
    """
    return np.std(j_average) / np.average(j_average) < conv_current

# ------------------------------------------------------------------------------
# Dateiein- und ausgabe
#
def LoadParameters(file):
    """
    Lade den Parametersatz aus einer Datei. Wandelt die Werte wenn möglich in
    float um.

    file : str
        Name der Parameterdatei

    returns
    -------
    parameter : dict
        Ein dictionary der Parameter
    """
    key = []
    value = []
    # Datei einlesen
    with open(file, 'r') as f:
        lines = f.read().split('\n')
    # Spalten den Namen und Werten zuordnen
    for l in lines:
        if not l == '':
            k, v = l.split('\t')
            key.append(k)
            if v == 'True':
                value.append(True)
            elif v == 'False':
                value.append(False)
            else:
                # Werte wenn möglich in Zahlen umwandeln
                try:
                    value.append(float(v))
                except:
                    value.append(v)
    return dict(zip(key, value))
```

```
def ConvertBool(value):
    """
    Konvertiere einen True/False Parameterwert nach bool in der Parameterdatei.

    value : str
        String, der nach True/False konvertiert werden soll.

    returns
    -------
    value : bool
        True oder False
    """
    if value == 'True':
        return True
    elif value == 'False':
        return False
    else:
        raise ValueError('Could not convert', value, 'to bool')

def Initialize(trial):
    """
    Zuerst wird gesucht, ob eine frühere Simulation fortgesetzt werden kann.
    Das ist der Fall, wenn alle Dateien für Ladungsträger, Energielandschaft,
    Morphologie und besetzte Zustände gleichzeitig existieren und lesbar sind.
    Ansonsten wird alles neu erstellt.
    Außerdem wird der Zufallsgenerator initialisiert bzw. aus der vorherigen
    Simulation geladen.

    trial : int
        Nummer des Trials

    returns
    -------
    dictionary mit den relevanten Variablen, siehe am Ende der Funktion

    Wenn die Simulation bereits fertig ist, wird `None` zurückgegeben.
    """
    # ------------------------------------------------------------------------
    # alle Dateinamen
    #
    # Dateiname für die Position des Ladungsträgers
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    # Dateinamen, unter dem die Morphologie gespeichert wird
    fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) + '.npy')
    # Dateiname der Morphologie, falls diese aus dem Hauptordner geladen werden soll
    fn_morph_all = 'morphology.npy'
    # Dateiname, unter dem die Energielandschaft gespeichert wird
    fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
    # Dateiname für den Zustand des Zufallszahlengenerators
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')
    # prinzipiell erstmal keine Dateien laden
    load_files = False
    # wurde der Zufallsgenerator initialisiert?
    rng_seeded = False
    # ------------------------------------------------------------------------
    # Prüfe, ob eine frühere Simulation existiert
    #
    #!!! hier kann man nochmal prüfen, wann die Simulation fortgesetzt werden kann.
Bei beschädigten Dateien kann man eh nichts machen
    # ------------------------------------------------------------------------
    # Normalfall
    # wenn die carrier und rand Datei existiert und mindestens einer ihrer .tmp
Dateien nicht
    if os.path.exists(fn_car) and (not os.path.exists(fn_car + '.tmp') or not
os.path.exists(fn_rand + '.tmp')):
        # carrier Datei einlesen
        with open(fn_car, 'r', encoding='utf-8') as f:
            data = f.read().split('\n')
        # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
        try:
            # alle Zeitpunkte einlesen
            times = [ float(line.split()[2]) for line in data if line.startswith('#
time')]
            # letzte Positionen der Ladungsträger
            carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
        except:
            if debugmode:
                print('could not read carrier file from previous simulation in trial',
trial)
        else:
            # letzten Simulationszeitpunkt
            global_time = times[-1]
            if debugmode:
                print('global_time from previous simulation', global_time)
            # bestimme die Anzahl der bisherigen Simulationsschritte
            a0 = int((len(times) - 1) * position_steps)
            if debugmode:
                print('number of completed simulation steps', a0)
            # überprüfe, ob der Trial bereits beendet wurde
            if a0 >= sim_steps:
                return None
            # Stromdichten laden
            a1, j_last, j_average = ReadCurrentdensity(trial)
            if debugmode:
                # print('j_last', j_last)
                print('j_average', j_average)
            # überprüfe, ob die Stromdichte schon konvergiert ist
            if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                return None
            # Dateien nur laden, wenn alle existieren
            if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                os.path.exists(fn_rand):
                load_files = True
                if debugmode:
                    print('previous carrier list loaded')
    # ------------------------------------------------------------------------
    # Rückfallebene, wenn die Simulation irgendwo beim Speichern abgebrochen wurde
```

```
    # wenn beide temporären Dateien (carrier und rand) existieren (ursprüngliche damit
auch)
    elif os.path.exists(fn_car + '.tmp') and os.path.exists(fn_rand + '.tmp'):
        # carrier Datei einlesen
        with open(fn_car + '.tmp', 'r', encoding='utf-8') as f:
            data = f.read().split('\n')
        # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
        try:
            # alle Zeitpunkte einlesen
            times = [ float(line.split()[2]) for line in data if line.startswith('#
time')]
            # letzte Positionen der Ladungsträger
            carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
            # überprüfe, ob die Zufallsgeneratordatei eingelesen werden kann
            with open(fn_rand + '.tmp', 'rb') as f:
                random_state = pickle.load(f)
        except:
            # ----------
            # Rückfallebene funktioniert nicht, verwende Normalfall
            if debugmode:
                print('could not read temporary files in trial', trial, ', trying
original ones')
            # carrier Datei einlesen
            with open(fn_car, 'r', encoding='utf-8') as f:
                data = f.read().split('\n')
            # Versuche den letzten Zeitpunkt und die letzte Position zu bestimmen
            try:
                # alle Zeitpunkte einlesen
                times = [ float(line.split()[2]) for line in data if
line.startswith('# time')]
                # letzte Positionen der Ladungsträger
                carriers_final = np.array([ d.split() for d in data[-4:-1] ],
dtype=np.int).transpose()
                # überprüfe, ob die Zufallsgeneratordatei eingelesen werden kann
                with open(fn_rand, 'rb') as f:
                    random_state = pickle.load(f)
            except:
                # gar nichts hat funktioniert
                if debugmode:
                    print('both temporary and original data corrupted')
            else:
                # letzten Simulationszeitpunkt
                global_time = times[-1]
                if debugmode:
                    print('global_time from previous simulation', global_time)
                # setze den Zustand des Zufallsgenerators
                rnd.set_state(random_state)
                rng_seeded = True
                if debugmode:
                    print('previous random number generator state loaded')
                # bestimme die Anzahl der bisherigen Simulationsschritte
                a0 = int((len(times) - 1) * position_steps)
                if debugmode:
                    print('number of completed simulation steps', a0)
                # überprüfe, ob der Trial bereits beendet wurde
                if a0 >= sim_steps:
                    return None
                # Stromdichten laden
                a1, j_last, j_average = ReadCurrentdensity(trial)
                if debugmode:
                    # print('j_last', j_last)
                    print('j_average', j_average)
                # überprüfe, ob die Stromdichte schon konvergiert ist
                if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                    return None
                # Dateien nur laden, wenn alle existieren
                if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                    os.path.exists(fn_rand):
                    load_files = True
                    if debugmode:
                        print('previous carrier list loaded')
            # ----------
        else:
            if debugmode:
                print('using temporary files from previous simulation')
            # letzten Simulationszeitpunkt
            global_time = times[-1]
            if debugmode:
                print('global_time from previous simulation', global_time)
            # setze den Zustand des Zufallsgenerators
            rnd.set_state(random_state)
            rng_seeded = True
            if debugmode:
                print('previous random number generator state loaded')
            # bestimme die Anzahl der bisherigen Simulationsschritte
            a0 = int((len(times) - 1) * position_steps)
            if debugmode:
                print('number of completed simulation steps', a0)
            # überprüfe, ob der Trial bereits beendet wurde
            if a0 >= sim_steps:
                return None
            # Stromdichten laden
            a1, j_last, j_average = ReadCurrentdensity(trial)
            if debugmode:
                # print('j_last', j_last)
                print('j_average', j_average)
            # überprüfe, ob die Stromdichte schon konvergiert ist
            if j_average is not None and check_convergence_criteria and
ConvergenceCriteria(j_average):
                return None
            # Dateien nur laden, wenn alle existieren
            if os.path.exists(fn_E) and os.path.exists(fn_morph) and \
                os.path.exists(fn_rand):
                load_files = True
                if debugmode:
                    print('previous carrier list loaded')

    # ------------------------------------------------------------------------
```

```
# Zufallsgenerator initialisieren
#
# Zustand des Generators laden, wenn vorhanden
if os.path.exists(fn_rand) and load_files and not rng_seeded:
    # lade den Zustand des Zufallsgenerators
    with open(fn_rand, 'rb') as f:
        random_state = pickle.load(f)
    # setze den Zustand des Zufallsgenerators
    rnd.set_state(random_state)
    if debugmode:
        print('previous random number generator state loaded')
# Generator erstmalig initialisieren mit Trialnummer
elif not rng_seeded:
    # zufällige Energielandschaft, oder die von Trial 0?
    if random_landscape:
        rnd.seed(trial)
        if debugmode:
            print('random number generator seeded with trial number')
        # prüfe Sinnhaftigkeit bei load_morphology, kann bei Bedarf auch weg
        if load_morphology:
            raise ValueError('morphology fixed, but energies random???')
    else:
        rnd.seed(0)
        if debugmode:
            print('random number generator seeded with 0')

# ----------------------------------------------------------------------------
# Morphologie initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_morph) and load_files:
    # lade vorhergehende Morphologie
    morphology = np.load(fn_morph)
    if debugmode:
        print('previous morphology loaded')
# eine Morphologie für alle Trials und Parameter?
elif load_morphology:
    # lade die Morphologie aus dem Hauptordner
    morphology = np.load(fn_morph_all)
    # kopiere sie in den Ordner für den jeweiligen Parameter
    if save_data:
        np.save(fn_morph, morphology)
    # Anteil der Fallen berechnen
    if interface_traps:
        eta = len(np.where(morphology < 0)[0]) / np.prod(box[:2])
    else:
        eta = len(np.where(morphology < 0)[0]) / np.prod(box)
    if debugmode:
        print('morphology loaded from main folder')
# Morphologie neu erstellen
else:
    if debugmode:
        print('new morphology')
    # Morphologie erstellen
    morphology, eta = CreateMorphology(morphology_type)
    # Morphologie speichern
    if save_data:
        np.save(fn_morph, morphology)

# ----------------------------------------------------------------------------
# Energielandschaft initialisieren
#
# überprüfe ob eine vorhergehende Simulation vorgesetzt werden soll
if os.path.exists(fn_E) and load_files:
    # lade vorhergehende Energielandschaft
    E = np.load(fn_E)
    if debugmode:
        print('previous energy landscape loaded')
# Energielandschaft neu berechnen
else:
    if debugmode:
        print('new energy landscape')
    s = time.perf_counter()
    # Energielandschaft neu mit Morphologie berechnen
    E = Energylandscape(box, sigma, morphology, model=energy_model)
    e = time.perf_counter()
    if debugmode:
        print('time for energy landscape', e - s)
    # Energielandschaft speichern
    if save_data:
        np.save(fn_E, E)

# ----------------------------------------------------------------------------
# wenn keine Dateien geladen wurden, Ladungsträger neu verteilen
#
if not load_files:
    # immer die gleiche Energielandschaft? Dann jedes Mal andere Ladungsträger
    if not random_landscape:
        rnd.seed()
    # Platziere sich bewegenden Ladungsträger
    carriers_final = PlaceCarriers(N_carriers, box)
    # Globale Zeit
    global_time = 0.
    # Schleifenvariablen
    a0, a1 = 0, 0
    # Array der letzen N_j_last (=1000) Stromdichten. Über die wird gemittelt.
    j_last = np.zeros(N_j_last)
    # Array der Mittelwerte der Stromdichten, N_average Mittelwerte werden im
Abstand
    # average_step (=100) gebildet
    j_average = np.zeros(N_average)
    # wenn die Datei mit den Positionen existiert, wird sie gelöscht
    if os.path.exists(fn_car):
        os.remove(fn_car)
    # Kopfzeilen in der Ausgabedatei schreiben
    WriteOutputHeader(trial, N_carriers, eta)
if debugmode:
    print('number of charge carriers from gate voltage', N_carriers)
    print('a0, a1:', a0, a1)
return {'load_files': load_files, 'global_time': global_time,
```

```
            'carriers_final': carriers_final, 'morphology': morphology, 'E': E,
            'a0': a0, 'a1': a1, 'j_last': j_last, 'j_average': j_average}

def ReadCurrentdensity(trial):
    """
    Lese die Stromdichten aus der carrier-Datei aus.

    trial : int
        Nummer des Trials

    returns
    -------
    a1, j_last, j_average : int, 1D array, 1D array
        Anzahl der bisherigen Simulationsschritte, Array für die letzten N
        Stromdichten und Array für die Mittelwerte über die letzten Stromdichten
    """
    # Dateiname für die Position des Ladungsträgers
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    # Datei einlesen
    with open(fn_car, 'r') as f:
        data = f.read().split('\n')
    # alle Stromdichten
    j = [ float(d.split()[-1]) for d in data if d.startswith('# time') ]
    # entferne den ersten Stromdichtewert (null), der erste Simulationspunkt
    j = j[1:]
    # Anzahl der bisherigen Simulationsschritte
    a1 = len(j)
    # Array für die Mittelwerte der Stromdichten
    j_average = np.zeros(N_average)
    # Array der letzten Stromdichten
    j_last = np.zeros(N_j_last)
    # Anzahl der möglichen Mittelwerte für die Stromdichten
    if len(j) >= N_j_last:
        ii = int((len(j) - N_j_last + average_step) / average_step)
    # zu wenig Datenpunkte
    else:
        j_last[:len(j)] = j
        return (a1, j_last, None)
    # berechne die Mittelwerte
    for i in range(max(0, ii - N_average), ii):
        j_average[i % N_average] = np.average(j[i * average_step : i * average_step +
N_j_last])
    # Array so umschichten, dass der letzte Wert bei a1 - 1 liegt, damit es
    # an der richtigen Stelle weiter geht
    j_last[:] = np.roll(j[-N_j_last:], a1)
    return (a1, j_last, j_average)

def WriteOutputHeader(trial, N, eta):
    """
    header-Zeilen der Ausgabedatei erstellen.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    N : int
        Anzahl der Ladungsträger im Device.
    eta : float oder tuple
        Bruchteil, je nach Morphologieart
    """
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')

    # Kopfzeilen der Datei
    header = '# N_carriers ' + str(N) + '\n'
    if morphology_type == 'polymer':
        header += '# fill_eta ' + str(eta[1]) + '\n'
    elif morphology_type == 'pointgrid':
        header += '# eta_traps ' + str(eta) + '\n'
    elif morphology_type == 'crystallite':
        header += '# eta_boundary ' + str(eta) + '\n'

    # schreibe die Kopfzeilen
    with open(fn_car, 'wb') as f:
        f.write(header.encode())
    # schreibe eine dummy-Datei für den Zufallszahlengenerator
    with open(fn_rand, 'w') as f:
        f.write('')

def WriteOutput(trial, carriers, global_time, current):
    """
    Speichere die Positionen der Ladungsträger zu einem Zeitpunkt.
    Speichere den Zustand des Zufallsgenerators.

    trial : int
        Nummer des Trials, wird für den Dateinamen benötigt
    carriers : 2D array
        Liste der Positionen der Ladungsträger
    global_time : float
        aktuelle globale Simulationszeit
    current : float
        Stromdichte in x-Richtung in A / cm**2
    """
    # Dateinamen
    fn_car = os.path.join(folder_out, 'carriers_trial' + str(trial) + '.asc')
    fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) + '.pkl')

    # kopiere die bestehenden Dateien
    shutil.copy(fn_car, fn_car + '.tmp')
    shutil.copy(fn_rand, fn_rand + '.tmp')

    # schreibe die Liste der Ladungsträger und den Zeitpunkt in die Datei
    with open(fn_car, 'ab') as f:
        header = 'time ' + str(global_time) + ' j ' + str(current)
        np.savetxt(f, carriers.transpose(), delimiter='\t', fmt='%1i', \
                   header=header)
    # speichere den Zustand des Zufallszahlengenerators
    with open(fn_rand, 'wb') as f:
        pickle.dump(rnd.get_state(), f, protocol=3)

    # bennene die temporären Dateien um
    os.remove(fn_car + '.tmp')
```

```
    os.remove(fn_rand + '.tmp')


################################################################################
# Funktionen für Parallelisierung
# ------------------------------------------------------------------------------

def SplitList(liste, size):
    """
    Teile eine Liste in Unterlisten, sodass alle etwa die gleiche Länge haben.

    liste: list
        aufzuteilende Liste
    size : int
        Anzahl der Unterlisten

    returns
    -------
    list of list
        Liste mit den Unterlisten
    """
    if not isinstance(liste, list):
        liste = list(liste)
    # Länge der einzelnen Unterlisten
    L = [int(len(liste) / size)] * size
    # die letzten Unterlisten sind um eins länger
    for i in range(len(liste)%size):
        L[-i-1] += 1
    sublists = [ liste[i:j] for i,j in zip(np.cumsum(L) - L, np.cumsum(L)) ]
    return sublists

def WorkerCarrier(V_single, potential, E, carriers, rates, indices, childP,
print_lock):
    """
    Prozess, der die Sprungraten für einige, feste Ladungsträger berechnet.
    Die Ergebnisse werden über eine Pipe an den Hauptprozess geschickt.

    V_single : 3D array
        Potential eines einzelnen Ladungsträgers
    potential : 3D array
        Coulombpotential zu Beginn der Rechnung
    E : 3D array
        Energielandschaft
    carriers : 2D array
        Array der Ladungsträger
    indices : list
        Liste mit den Indizes der Ladungsträger, um die sich dieser Prozess
        kümmern soll
    childP : pipe
        Pipe zum Datenaustausch, ist verbunden mit parentP
    """
    if debugmode:
        with print_lock:
            print('process', os.getpid(), 'started')
            print('received E[0,0,0]', E[0,0,0], '\ncarrier 0', carriers[0], \
                  '\ncarrier indices for calculation', indices)
    # Schleifenvariable für die Aktualisierung des Coulombpotentials
    a = 1
    while True:
#        t0 = time.perf_counter()
        # Sprungraten für jeden Ladungsträger berechnen
        for i in indices:
            k = CalculateRatesPerCarrier(i, carriers, E, potential)
            rates[i::N_carriers] = k
#        t1 = time.perf_counter()
        # schicke an den Parentprozess, dass die Berechnung fertig ist
        childP.send('done')
        # neuen Ladungsträger oder Ende des Prozesses erhalten
        c, jump = childP.recv()
        if c is None:
            break
        else:
            # Coulombpotential alle 10000 Sprünge neu berechnen, habe ich geprüft
            if a % update_potential_steps == 0:
                potential = CalculatePotential(V_single, carriers)
                # if debugmode:
                #     with print_lock:
                #         print('difference between update und recalculation of
potential',
                #               np.sum(np.abs(potential - UpdatePotential(V_single,
potential, carriers[c], jump))))
            else:
                potential = UpdatePotential(V_single, potential, carriers[c], jump)
            # Position des gesprungenen Ladungsträgers aktualisieren
            carriers[c] = (carriers[c] + jump) % box
            a += 1
#            if debugmode:
#                with print_lock:
#                    print('process', os.getpid(), \
#                          '\npotential[0,0,0]', potential[0,0,0], \
#                          '\nnew carrier', carriers[c])
#        t2 = time.perf_counter()
#        with print_lock:
#            print('rates', t1 - t0)
#            print('rest', t2 - t1)

def WorkerCDM(dipoles, indices, box, gridspacing, inQ, outQ):
    """
    Räumliche Korrelation der Dipole parallel berechnen.
    """
    while True:
        pos = inQ.get()
        if pos is None:
            break

        ret = np.sum( np.sum( dipoles[tuple(np.transpose((pos + indices)%box))] \
                                        *indices*gridspacing, axis=1 ) / \
                                    np.sum((indices*gridspacing)**2, axis=1)**1.5 )
        outQ.put( (pos, ret) )
```

```
###############################################################################
# Monte-Carlo-Simulation
# ------------------------------------------------------------------------------

def MonteCarloTrial(trial):
    """
    Funktion für MonteCarlo-Trial.
    """
    # --------------------------------------------------------------------------
    # Initialisierung des Trials
    #
    dict_init = Initialize(trial)
    # wenn die Simulation schon beendet ist, abbrechen
    if dict_init is None:
        if debugmode:
            print('trial already finished')
        return
    # Variablen aus der Initialisierung übertragen
    skip_initial = dict_init['load_files']
    global_time = dict_init['global_time']
    carriers_final = dict_init['carriers_final']
    # morphology = dict_init['morphology'] # <-- wird nicht verwendet
    E = dict_init['E']
    a0 = dict_init['a0']
    a1 = dict_init['a1']
    j_last = dict_init['j_last']
    j_average = dict_init['j_average']
    # Ladungsträger in der Simulationsbox (mit periodischer Randbedingung)
    carriers = carriers_final % box
    # Anzahl der Ladungsträgersprünge in x-Richtung (down-field-hops)
    dfh = 0
    # letzter Zeitpunkt für die Berechnung der Stromdichte
    t0 = global_time
    # beim ersten Start der Simulation Positionen der Ladungsträger speichern
    if not skip_initial:
        if debugmode:
            print('initial carrier position saved')
        WriteOutput(trial, carriers_final, global_time, 0)

    # --------------------------------------------------------------------------
    # Arbeitsprozesse initialisieren
    #
    # shared memory Array für die Sprungraten
    # Index: c + j * N_carriers, c: Index Ladungsträger, j: Index Sprungrichtung
    # Im Array sind die Sprungrichtungen gruppiert, dh. nacheinander kommen die
    # Ladungsträger mit gleicher Sprungrichtung. Dadurch soll das Array absteigend
    # sortiert werden, da weitere Sprünge am Ende stehen.
    rates = mp.Array('d', N_carriers * N_jumps, lock=False)
    # Potential eines einzelnen Ladungsträgers
    V_single = SingleCarrierPotential()
    # Coulomb-Potential aller Ladungsträger
    potential = CalculatePotential(V_single, carriers)
    # Liste der Indizes der Ladungsträger
    carriers_indices = np.arange(N_carriers)
    # teile die Liste der Indizes in Unterlisten auf
    sublists = SplitList(carriers_indices, N_processes)
    # Pipes erzeugen zum Datenaustausch mit den Prozessen
    parentP, childP = zip(*[ mp.Pipe() for i in range(N_processes) ])
    # Lock für die Prozesse bei print
    print_lock = mp.Lock()
    # Prozesse erzeugen
    workers = [mp.Process(target=WorkerCarrier, \
                          args=(V_single, potential, E, carriers, rates, sublists[i],
childP[i], print_lock))
               for i in range(N_processes)]
    # Prozesse starten
    for w in workers:
        w.start()

    # --------------------------------------------------------------------------
    # Monte-Carlo-Schleife
    #
    if debugmode:
        print('--- beginning Monte Carlo loop ---')
    while 1:
        # --- Dateiausgabe ---
        # Nach einer bestimmten Anzahl an Sprüngen Position und Zeit der Ladungsträger
speichern
        if a0 % (position_steps) == 0 and a0 > 0:
            # Dateiausgabe beim ersten Mal nach einem Neustart der Simulation
überspringen
            if skip_initial:
                skip_initial = False
            else:
                # Stromdichte (A / cm^2)
                current = dfh * 1.602e-19 / (np.prod(box) * np.prod(gridspacing[1:]))
/ (global_time - t0) * 1e23
                WriteOutput(trial, carriers_final, global_time, current)

                # die letzten N Werte der Stromdichte speichern
                j_last[a1 % N_j_last] = current
                # Alle 100 Schritte wird der Mittelwert gebildet (überschneidet sich)
                if (a1 + 1 - N_j_last) % average_step == 0 and a1 >= N_j_last - 1:
                    j_average[int((a1 + 1 - N_j_last) / average_step) % N_average] =
np.average(j_last)
                    if debugmode:
                        print('j_average', j_average)
                    # sobald das Array gefüllt wurde, Konvergenzkriterium prüfen, wenn
gewünscht
                    if a1 >= N_j_last + average_step * N_average - 1 and
check_convergence_criteria:
                        # Abbruchkriterium
                        if ConvergenceCriteria(j_average):
                            if debugmode:
                                print('convergence criteria fulfilled')
                            break
                # Zeit und Zähler für Sprünge zurücksetzen
                t0 = global_time
                dfh = 0
                # Schleifenvariable für Stromdichte erhöhen
```

```
                a1 += 1

        # Sprungraten berechnen
        CalculateRates(parentP)
        # Raten aufsummieren
        k = np.cumsum(rates)
#        if debugmode:
#            # Sprungrichtung
#            jum = 4
#            print('jump rates in', jumps[jum])
#            for i,r in enumerate(rates[jum * N_carriers : (jum + 1) * N_carriers]):
#                print('carrier', i, r)
        # Zufallszahl um ein Ereignis auszuwählen
        r = rnd.random() * k[-1]
        # Waiting time für jeden Schritt berechnen
        global_time += -np.log(rnd.random()) / k[-1]
        # Wähle das entsprechende Ereignis aus
        for i in range(N_jumps * N_carriers):
            if r < k[i]:
                # Index des Ladungsträgers
                c = i % N_carriers
                # Index der Sprungrichtung
                j = int(i / N_carriers)
                # neuen Ladungsträger an alle Prozesse schicken
                for i in range(N_processes):
                    parentP[i].send((c, jumps[j]))
#                if debugmode:
#                    print('event: carrier', c, carriers[c], 'jump', jumps[j])
                # bewege Ladungsträger
                carriers[c] += jumps[j]
                carriers_final[c] += jumps[j]
                # Berücksichtige Periodizität in x- und y-Richtung
                carriers[c,0] = carriers[c,0] % x_length
                carriers[c,1] = carriers[c,1] % y_length
                # Sprung in Feldrichtung addieren
                dfh += jumps[j,0]
                break
        # Wenn die Anzahl der Simulationsschritte überschritten ist, beenden
        if a0 >= sim_steps - 1:
            if debugmode:
                print('max simulation steps reached')
            # Stromdichte (A / cm^2)
            current = dfh * 1.602e-19 / (np.prod(box) * np.prod(gridspacing[1:])) /
(global_time - t0) * 1e23
            WriteOutput(trial, carriers_final, global_time, current)
            break
#        if debugmode and a == 0:
#            break
        a0 += 1

    # Prozesse beenden
    for i,w in enumerate(workers):
        parentP[i].send((None, None))
        w.join()
```

```
################################################################################
# Hauptfunktion zur Steuerung des Programmablaufs
# ------------------------------------------------------------------------------

def Main():
    """
    Hauptfunktion. Starte die einzelnen Trials.
    """
    # Starte Monte-Carlo-Trials
    s = time.perf_counter()
    for trial in range(number_trials):

        MonteCarloTrial(trial)
        print('Trial', trial, 'finished.')

        # Lösche die Dateien der Morphologie und Energielandschaft
        if delete_data:
            fn_morph = os.path.join(folder_out, 'morphology_trial' + str(trial) +
'.npy')
            fn_E = os.path.join(folder_out, 'energy_trial' + str(trial) + '.npy')
            fn_rand = os.path.join(folder_out, 'randstate_trial' + str(trial) +
'.pkl')
            if os.path.exists(fn_morph):
                os.remove(fn_morph)
            if os.path.exists(fn_E):
                os.remove(fn_E)
            if os.path.exists(fn_rand):
                os.remove(fn_rand)
    e = time.perf_counter()
    print('runtime all trials', e - s)

################################################################################
# Kommandozeilen-Optionen festlegen
# ------------------------------------------------------------------------------

# ---------------------------------
# mögliche Kommandozeilen-Optionen:
# ---------------------------------
#   Nummer des Parametersatzes: Zahl, muss erste Option sein
#   alle weiteren Optionen sind vom Format key=value
#   Anzahl der Prozesse zur Parallelisierung mit 'cores='

# Werden Optionen übergeben? Ist der Fall, wenn auf dem Cluster gerechnet wird.
cluster_rechnen = len(sys.argv) != 1

# Anzahl der Prozesse auf dem lokalen Rechner
N_processes = 4
# Anzahl der Knoten, default 1
N_nodes = 1

if cluster_rechnen:
    for option in sys.argv[2:]:
        key, value = option.split('=')
        # Anzahl der Prozesse
        if key == 'cores':
```

```
            N_processes = int(value)
#        elif key == 'nodes':
#            N_nodes = int(value)
        else:
            raise IOError('Kommandozeilenoption ' + str(option) + \
                          ' nicht bekannt.')

# ----------------------------------------------------------------------------
# Nummer des Parametersatzes
#

if cluster_rechnen:
    # Nummer des Parametersatzes ermitteln
    parametersatz = int(sys.argv[1])
else:
    # Testordner auf lokalem Rechner
    folder_local = 'C:/Daten/TobiasMeier/Simulationen/Tests'
    # wechsele in das lokale Verzeichnis
    os.chdir(folder_local)
    # Nummer des Parametersatzes
    parametersatz = 1

############################################################################
# Parameter aus Datei einlesen
# ----------------------------------------------------------------------------

# Dateinamen für den Parametersatz
fn_params = os.path.join('Parameters' + str(parametersatz), \
                         'parameters' + str(parametersatz) + '.asc')

# Dictionary aus der Parameterdatei lesen
d = LoadParameters(fn_params)

# Gitterabstände (nm)
dx = d['dx']
dy = d['dy']
dz = d['dz']

# Boxlänge (Anzahl der Gitterplätze)
x_length = int(d['x_length'])
y_length = int(d['y_length'])
z_length = int(d['z_length'])

# Anzahl der Trials
number_trials = int(d['number_trials'])

# Soll eine Morphologie für alle Trials geladen werden? Muss sich im Hauptordner
befinden
load_morphology = d['load_morphology']

# Soll die Energielandschaft zufällig sein? (sonst immer wie Trial 0)
random_landscape = d['random_landscape']

# maximale Simulationszeit (ns)
sim_steps = d['sim_steps']

# soll das Konvergenzkriterium für die Stromdichte verwendet werden?
check_convergence_criteria = d['current_conv']

# ----------------------------------------------------------------------------
# Eigenschaften der Morphologie

# Art der Morphologie
morphology_type = d['morphology']
# Ist die erste Schicht ausschließlich mit Fallen besetzt?
trap_layer = d['trap_layer']

if morphology_type == 'pointgrid':
    # Fallenverteilungsfunktion
    trap_distribution = d['trap_distribution']
    # Sollen sich die Fallen nur in der Grenzschicht befinden?
    interface_traps = d['interface_traps']
    # Dichte der Fallenzustände
    trapdensity = d['trapdensity']
    # Kettenlänge
    chain_length = int(d['chain_length'])
    # Breite der Verteilung der Kettenlänge
    chain_sigma = d['chain_sigma']
    # fester Versatz zwischen Ketten
    chain_offset = d['chain_offset']
    if chain_offset != 'random':
        chain_offset = int(chain_offset)

# ----------------------------------------------------------------------------
# Elektrische und elektronische Eigenschaften
#
# Verwendetes Energiemodell
energy_model = d['energy_model']
# Breite der Energieverteilung sigma (eV)
sigma = d['sigma']
# Zentrum der Energieverteilung der Fallenzustände (eV)
E_trap = d['E_trap']
# Breite der Energieverteilung der Fallenzustände (eV)
sigma_trap = d['sigma_trap']
# Dehnungsfaktor für den exponentiellen Anteil gegenüber dem Gaußanteil
# sigma_exp = stretch_exp * sigma_trap
if trap_distribution == 'gaussexp':
    stretch_exp = d['stretch_exp']
# Dicke des Gate-Dielektrikums (nm)
d_gate = d['d_gate']
# relative Dielektrizitätskonstante des Gate-Dielektrikums
epsilon_gate = d['epsilon_gate']
# Gate-Spannung (V)
U_Gate = d['U_Gate']
# Drain-Spannung (V)
U_Drain = d['U_Drain']
# relative Dielektrizitätskonstante
epsilon_rel = d['epsilon_rel']

# ----------------------------------------------------------------------------
```

```
# Parameter für Sprünge
#
# Temperatur
temperature = d['temperature']
# Hüpfmechanismus
hopping = d['hopping']
# Miller-Abrahams
if hopping == 'MA':
    # Attempt-to-hop frequency (1/ns)
    nu0 = d['nu0']
# Marcus-Rate
elif hopping == 'marcus':
    # Reorganisationenergie
    reorg_energy = d['reorg_energy']
    # Transferintegral
    J0 = d['J0']
else:
    raise ValueError('unknown hopping mechanism')
# inverse Lokalisierungskonstanten(1/nm)
# innerhalb einer Polymerkette
gamma_intra = d['gamma_intra']
# Kopplungskonstante in x-Richtung
gamma_x = d['gamma_x']
# Kopplungskonstante in y-Richtung
gamma_y = d['gamma_y']
# Kopplungskonstante in z-Richtung
gamma_z = d['gamma_z']
# Anzahl der Nächsten Nachbarn für Sprünge
nearest_neighbors = int(d['NN_hops'])

###############################################################################
# Konstanten und globale Variablen
# -----------------------------------------------------------------------------

# -----------------------------------------------------------------------------
# Konstanten
#
# kT (eV)
kT = 8.617e-5 * temperature
# Cut-off Radius für die Coulomb-Wechselwirkung (nm)
#cutoff_Coulomb = 100
# Cutoff-Radius für die Dipolwechselwirkung bei räumlich korrelierter Energie (nm)
cutoff_dipoles = 25
# Faktor für Coulomb-Wechselwirkung (in eV)
faktor_Coulomb = 1.602e-19 / 4 / np.pi / 8.854e-12 / epsilon_rel * 1e9
# Potentialunterschied bei einem Schritt in x-Richtung
voltage_difference = U_Drain / x_length
# Potentialunterschied bei einem Schritt in z-Richtung durch die Gate-Spannung
gate_potential_step = epsilon_gate / epsilon_rel * abs(U_Gate) / d_gate * dz
# Anzahl der Ladungsträger durch den Kondensator
N_carriers = int(epsilon_gate * 8.854e-12 * x_length * y_length * dx * dy *
abs(U_Gate) \
                 / 1.602e-19 / d_gate * 1e-9)

# Konstanten für Marcus-Rate
if hopping == 'marcus':
    C1_marcus = 2 * np.pi / 6.582e-16 * J0**2 / np.sqrt(4 * np.pi * kT * reorg_energy)
* 1e-9
    C2_marcus = 4 * reorg_energy * kT

# -----------------------------------------------------------------------------
# Globale Variablen
#
# Array für Größe der Simulationsbox
box = np.array([x_length, y_length, z_length])
# Array für Abstände der Gitterplätze
gridspacing = np.array([dx, dy, dz])
# Inverse Lokalisierungskonstanten
gamma = np.array([gamma_x, gamma_y, gamma_z])
# mögliche Sprungrichtungen (globale Variable)
jumps = PossibleJumps(nearest_neighbors, 3)
# Anzahl der Sprungrichtungen (globale Variable)
N_jumps = len(jumps)

# Korrektur für das Potential eines Ladungsträgers in der Ratenberechnung
correction_potential = CorrectionPotential()

# -----------------------------------------------------------------------------
# Parameter für die Dateiausgabe
#

# Verzeichnis des aktuellen Parametersatzes
folder_out = 'Parameters' + str(parametersatz)

# Anzahl der Schritte nach denen das Potential neu berechnet werden soll
update_potential_steps = 10000
# Anzahl der Sprünge, nach denen die Position gespeichert werden soll
position_steps = 10000
# Anzahl der Stromdichten über die der Mittelwert bestimmt werden soll
N_j_last = 1000
# Anzahl der Mittelwerte an Stromdichten, die für die Konvergenz benutzt wird
N_average = 10
# Abstand zwischen den Stromdichtemittelwerten
average_step = 100
# Konvergenzkriterium für die Stromdichte
conv_current = 0.05

###############################################################################
# starte Simulation mit Parallelisierung
# -----------------------------------------------------------------------------

if __name__ == '__main__':
    Main()

# -----------------------------------------------------------------------------
# Programmende
###############################################################################
```

A.2.4. Skript zum Erstellen der Parametersätze für die Einteilchensimulationen mit effektiver Energielandschaft

```
# -*- coding: utf-8 -*-
"""
Parameterdatei für OFET-Simulation erstellen.

    - Einteilchensimulation.
    - Veränderliche Parameter werden automatisch bestimmt, einfach dazu eine
      Liste bei den jeweiligen Parametern eintragen.
    - Je nach Typ der Morphologie werden nur die benötigten Parameter in die
      Datei geschrieben.
    - Der Zufallszahlengenerator für die Energielandschaft und die Ladungsträger
      können entkoppelt werden. Bei Entkopplung wird immer die Energielandschaft
      von Trial 0 erzeugt.
"""
import numpy as np
import os
import itertools

# Ordner, in dem die Parameterdatei erstellt werden soll
folder = '2022-01-19_ET_woCoulomb'

# Startindex der Parametersätze
number_params = 1

# Wurzelverzeichnis auf dem Cluster, wird zusammengefügt mit dem Ordnernamen
folder_cluster = '//home-pc.uni-bayreuth.de/home/05/bt******/Simulation/OFET'

# Testordner auf lokalem Rechner verwenden?
local_folder = False

################################################################################
# Parametersatz
# ------------------------------------------------------------------------------

# Gitterabstände (nm)
dx = 1
dy = 1
dz = 1

# Boxlänge (Anzahl der Gitterplätze)
x_length = 100
y_length = 50
z_length = 3

# Anzahl der Trials
number_trials = 1000

# Soll eine Morphologie für alle Trials geladen werden? Muss sich im Hauptordner
befinden
load_morphology = False

# Soll für jeden Trial eine zufällige Energielandschaft verwendet werden?
# sonst wird immer die von Trial 0 erzeugt
random_landscape = True

# maximale Anzahl an Simulationsschritten (ns)
sim_steps = 1e8

# ------------------------------------------------------------------------------
# Morphologie

# Typ der Morphologie ['polymer', 'crystallite', 'pointgrid']
morphology = ['polymer', 'crystallite', 'pointgrid'][2]

# ---------------------------
# Parameter für Polymerketten
#
# Kettenlänge
chain_length = 1
# Breite der Verteilung der Kettenlängen
chain_sigma = 0
# Richtungsfaktor chi
chi = 0.95
# Gitter ganz ausfüllen?
fill_grid = False
# sollen die Ketten ab der Hälfte einknicken?
kink = False
# wenn ja, ab welcher Gesamtkettenlänge soll in der Mitte geknickt werden
if kink:
    kink_length = chain_length - 2
else:
    kink_length = 0

# -------------------------
# Parameter für Kristallite
#
# Kantenlängen der Kristallite (nm)
crystal_a, crystal_b = 1.2, 1
# Winkel der Parallelogramme für die Kristallite (°)
crystal_gamma = 60
# Anzahl der Startpunkte für Kristallitwachstum
number_seeds = 4
# Dicke der Grain boundaries
boundary_gap = 1

# -------------------------
# Parameter für Punktgitter
#
# sollen die Gitterplätze unter Berücksichtigung der gegenseitigen
Coulombwechselwirkung befüllt werden?
```

```
fill_with_coulomb = False
# Soll für das Besetzen der Gitterplätze die Fermienergie berechnet werden?
fill_with_fermi = False
# Ist die erste Schicht ausschließlich mit Fallen besetzt?
trap_layer = False
# Sollen sich die Fallen nur in der Grenzschicht befinden?
interface_traps = True
# Dichte der Fallenzustände
trapdensity = 0.
# fester Versatz zwischen zwei Ketten, kann parallel (null), eine Zahl oder 'random'
sein
chain_offset = 'random'
# Kettenlänge (siehe oben)
# Breite der Kettenlängenverteilung (siehe oben)

# ------------------------------------------------------------------------------
# Energielandschaft
#
# Modell für die Energielandschaft (GDM oder CDM)
energy_model = 'GDM'
# Breite der Energieverteilung sigma (eV)
sigma = 0.05
# Fallenverteilungsfunktion
trap_distribution = ['gauss', 'exp', 'gaussexp'][0]
# Zentrum der Energieverteilung der Fallenzustände (eV)
E_trap = 0.
# Breite der Energieverteilung der Fallenzustände (eV)
sigma_trap = 0.
# Dehnungsfaktor für den exponentiellen Anteil gegenüber dem Gaußanteil
# nur wenn trap_distribution == 'gaussexp'
# sigma_exp = stretch_exp * sigma_trap
stretch_exp = 1
# Dicke des Gate-Dielektrikums (nm)
d_gate = 100
# relative Dielektrizitätskonstante des Gate-Dielektrikums
epsilon_gate = 4
# Gate-Spannung (V)
U_Gate = [0.5,1,2,5,10,20,50]
# relative Dielektrizitätskonstante
epsilon_rel = 4
# Drain-Spannung (V)
U_Drain = -1

# ------------------------------------------------------------------------------
# Parameter für Sprünge
#
# Temperatur (K)
temperature = [100,120,140,160,180,200,250,300,350,400]
# Modell für Hüpfraten
hopping = ['MA', 'marcus'][0]
# ------------------------------
# Miller-Abrahams
# Attempt-to-hop frequency (1/ns)
nu0 = 1e4
# ------------------------------
# Marcus-Rate
# Reorganisationsenergie (eV)
reorg_energy = 0.3
# Transferintegral J_0 (eV)
J0 = 2.009
# ------------------------------
# inverse Lokalisierungskonstanten für Hoppingrate nach Miller-Abrahams (1/nm)
# innerhalb einer Polymerkette
gamma_intra = 2.697
# Kopplungskonstante in x-Richtung
gamma_x = 5
# Kopplungskonstante in y-Richtung
gamma_y = 5
# Kopplungskonstante in z-Richtung
gamma_z = 5
# Anzahl der Nächsten Nachbarn für Sprünge
NN_hops = 6

################################################################################

def WriteParameterRecord(params):
    """
    params_keys : dict
        Dictionary der Parameter
    """
    global number_params
    # Liste mit den Namen der Parameter
    params_keys = list(params.keys())
    # Liste mit den Parameterwerten
    params_values = list(params.values())
    # Ordner für den Parametersatz erstellen
    folder_output = os.path.join(folder_root, 'Parameters' + str(number_params))
    # erhöhe die Nummer des Parametersatzes, falls dieser schon existiert
    while os.path.exists(folder_output):
        number_params += 1
        folder_output = os.path.join(folder_root, 'Parameters' + str(number_params))

    os.makedirs(folder_output)
    # Name für die Datei mit den Parametern
    fn_para = os.path.join(folder_output, 'parameters' + str(number_params) + '.asc')
    np.savetxt(fn_para, np.vstack((params_keys, params_values)).transpose(), \
               delimiter='\t', fmt='%s')
    print('Parametersatz', number_params, 'erstellt.')


def DetermineMorphology(morphology):
    """
    morphology : str
        Art der Morphologie

    returns
    -------
    params_keys : list
        Liste der Parameternamen
    params_values : list
```

```
        Liste der Parameterwerte
    """
    if morphology == 'polymer':
        params_keys = params_polymer.keys()
        params_values = params_polymer.values()
    elif morphology == 'crystallite':
        params_keys = params_crystallite.keys()
        params_values = params_crystallite.values()
    elif morphology == 'pointgrid':
        params_keys = params_pointgrid.keys()
        params_values = params_pointgrid.values()
    return list(params_keys), list(params_values)

def Hoppingmechanism(hopping):
    """
    Hüpfmechanismus zur Parameterliste hinzufügen.
    """
    # Parameter die für die Hüpfrate benötigt werden
    if hopping == 'MA':
        p = {'nu0': nu0}
    elif hopping == 'marcus':
        p = {'reorg_energy': reorg_energy,
             'J0': J0}
    else:
        raise ValueError('unknown hopping rate')
    params_polymer.update(p)
    params_crystallite.update(p)
    params_pointgrid.update(p)

################################################################################

folder_root = os.path.join(folder_cluster, folder)

if local_folder:
    folder_root = 'C:/Daten/TobiasMeier/Simulationen/Tests'

# ------------------------------------------------------------------------------
# Wörterbücher für die Parameter aller möglichen Morphologien
#
# Polymerketten
params_polymer = {
'morphology': morphology,
'load_morphology': load_morphology,
'random_landscape': random_landscape,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'chain_length': chain_length,
'chain_sigma': chain_sigma,
'chi': chi,
'fill_grid': fill_grid,
'kink': kink,
'kink_length': kink_length,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'number_trials': number_trials,
'NN_hops': NN_hops,
'sim_steps': sim_steps}

# Kristallite
params_crystallite = {
'morphology': morphology,
'load_morphology': load_morphology,
'random_landscape': random_landscape,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'crystal_a': crystal_a,
'crystal_b': crystal_b,
'crystal_gamma': crystal_gamma,
'number_seeds': number_seeds,
'boundary_gap': boundary_gap,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
```

```
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'number_trials': number_trials,
'NN_hops': NN_hops,
'sim_steps': sim_steps}

# Punktgitter
params_pointgrid = {
'morphology': morphology,
'load_morphology': load_morphology,
'random_landscape': random_landscape,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'interface_traps': interface_traps,
'trapdensity': trapdensity,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'chain_length': chain_length,
'chain_sigma': chain_sigma,
'chain_offset': chain_offset,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'number_trials': number_trials,
'NN_hops': NN_hops,
'fill_with_coulomb': fill_with_coulomb,
'fill_with_fermi': fill_with_fermi,
'sim_steps': sim_steps}

# Hüpfmechanismus hinzufügen
Hoppingmechanism(hopping)

#Auswahl der passenden Parameterliste anhand von gewählter Morphologie
params_keys, params_values = DetermineMorphology(morphology)

# ------------------------------------------------------------------------------

# veränderliche Parameter und Namen dazu
variable_params = []
variable_keys = []
# konstante Parameter und Namen dazu
constant_params = []
constant_keys = []

# Rausfinden welche Parameter veränderlich sind und welche nicht
for i,p in enumerate(params_values):
    # handelt es sich um eine Liste bei den Parametern?
    if isinstance(p, list):
        variable_params.append(p)
        variable_keys.append(params_keys[i])
    else:
        constant_params.append(p)
        constant_keys.append(params_keys[i])
print('veränderliche Parameter', variable_keys)

# alle möglichen Kombinationen der veränderlichen Parameter
for p in itertools.product(*variable_params):
    # konstante und einen Satz veränderlichen Parameter zusammensetzen
    d = dict(zip(variable_keys + constant_keys, list(p) + constant_params))
    WriteParameterRecord(d)
```

A.2.5. Skript zum Erstellen der Parametersätze für die Vielteilchensimulationen

```
# -*- coding: utf-8 -*-
"""
Parameterdatei für OFET-Simulation erstellen.

    - Einteilchensimulation.
    - Veränderliche Parameter werden automatisch bestimmt, einfach dazu eine
      Liste bei den jeweiligen Parametern eintragen.
    - Je nach Typ der Morphologie werden nur die benötigten Parameter in die
      Datei geschrieben.
    - Der Zufallszahlengenerator für die Energielandschaft und die Ladungsträger
      können entkoppelt werden. Bei Entkopplung wird immer die Energielandschaft
      von Trial 0 erzeugt.
"""
import numpy as np
import os
import itertools

# Ordner, in dem die Parameterdatei erstellt werden soll
folder = '2022-07-11_VT1'

# Startindex der Parametersätze
```

```
number_params = 1

# Wurzelverzeichnis auf dem Cluster, wird zusammengefügt mit dem Ordnernamen
folder_cluster = '//home-pc.uni-bayreuth.de/home/05/bt******/Simulation/OFET'

# Testordner auf lokalem Rechner verwenden?
local_folder = False

################################################################################
# Parametersatz
# ------------------------------------------------------------------------------

# Gitterabstände (nm)
dx = 1
dy = 1
dz = 1

# Boxlänge (Anzahl der Gitterplätze)
x_length = 100
y_length = 50
z_length = 3

# Anzahl der Trials, None wenn sie aus der Anzahl der Ladungsträger berechnet
# werden soll
number_trials = None
# Anzahl der Ladungsträger, die insgesamt in allen Trials simuliert werden
# Daraus bestimmt sich die Anzahl der Trials für jeden Parametersatz
number_carriers = 100

# Soll eine Morphologie für alle Trials geladen werden? Muss sich im Hauptordner
befinden
load_morphology = False

# Soll für jeden Trial eine zufällige Energielandschaft verwendet werden?
# sonst wird immer die von Trial 0 erzeugt
random_landscape = True

# maximale Anzahl an Simulationsschritten
sim_steps = 1e8

# soll die Stromdichte auf das Konvergenzkriterium geprüft werden?
check_convergence_criteria = False

# ------------------------------------------------------------------------------
# Morphologie

# Typ der Morphologie ['polymer', 'crystallite', 'pointgrid']
morphology = ['polymer', 'crystallite', 'pointgrid'][2]

# --------------------------
# Parameter für Polymerketten
#
# Kettenlänge
chain_length = 1
# Breite der Verteilung der Kettenlängen
chain_sigma = 0
# Richtungsfaktor chi
chi = 0.95
# Gitter ganz ausfüllen?
fill_grid = False
# sollen die Ketten ab der Hälfte einknicken?
kink = False
# wenn ja, ab welcher Gesamtkettenlänge soll in der Mitte geknickt werden
kink_length = chain_length - 2

# --------------------------
# Parameter für Kristallite
#
# Kantenlängen der Kristallite (nm)
crystal_a, crystal_b = 1.2, 1
# Winkel der Parallelogramme für die Kristallite (°)
crystal_gamma = 60
# Anzahl der Startpunkte für Kristallitwachstum
number_seeds = 4
# Dicke der Grain boundaries
boundary_gap = 1

# --------------------------
# Parameter für Punktgitter
#
# Dichte der Fallenzustände
trapdensity = 0
# Ist die erste Schicht ausschließlich mit Fallen besetzt?
trap_layer = False
# Sollen sich die Fallen nur in der Grenzschicht befinden?
interface_traps = True
# fester Versatz zwischen zwei Ketten, kann parallel (null), eine Zahl oder 'random'
sein
chain_offset = 0
# Kettenlänge (siehe oben)
# Breite der Kettenlängenverteilung (siehe oben)

# ------------------------------------------------------------------------------
# Energielandschaft
#
# Modell für die Energielandschaft (GDM oder CDM)
energy_model = 'GDM'
# Breite der Energieverteilung sigma (eV)
sigma = 0.05
# Fallenverteilungsfunktion
trap_distribution = ['gauss', 'exp', 'gaussexp'][0]
# Zentrum der Energieverteilung der Fallenzustände (eV)
E_trap = 0
# Breite der Energieverteilung der Fallenzustände (eV)
sigma_trap = 0
# Dehnungsfaktor für den exponentiellen Anteil gegenüber dem Gaußanteil
# nur wenn trap_distribution == 'gaussexp'
# sigma_exp = stretch_exp * sigma_trap
stretch_exp = 1
# Dicke des Gate-Dielektrikums (nm)
```

```
d_gate = 100
# relative Dielektrizitätskonstante des Gate-Dielektrikums
epsilon_gate = 4
# Gate-Spannung (V)
U_Gate = [1,2,5,10,20,50]
# relative Dielektrizitätskonstante
epsilon_rel = 4
# Drain-Spannung (V)
U_Drain = -1

# ------------------------------------------------------------------------------
# Parameter für Sprünge
#
# Temperatur (K)
temperature = [100,120,140,160,180,200,250,300,350,400]
# Modell für Hüpfraten
hopping = ['MA', 'marcus'][0]
# -------------------------------
# Miller-Abrahams
# Attempt-to-hop frequency (1/ns)
nu0 = 1e4
# -------------------------------
# Marcus-Rate
# Reorganisationsenergie (eV)
reorg_energy = 0.3
# Transferintegral J_0 (eV)
J0 = 2.009
# -------------------------------
# inverse Lokalisierungskonstanten für Hoppingrate (1/nm)
# innerhalb einer Polymerkette
gamma_intra = 2.697
# Kopplungskonstante in x-Richtung
gamma_x = 5
# Kopplungskonstante in y-Richtung
gamma_y = 5
# Kopplungskonstante in z-Richtung
gamma_z = 5
# Anzahl der Nächsten Nachbarn für Sprünge
NN_hops = 3

###############################################################################

def WriteParameterRecord(params):
    """
    params_keys : dict
        Dictionary der Parameter
    """
    global number_params
    # Liste mit den Namen der Parameter
    params_keys = list(params.keys())
    # Liste mit den Parameterwerten
    params_values = list(params.values())
    # Ordner für den Parametersatz erstellen
    folder_output = os.path.join(folder_root, 'Parameters' + str(number_params))
    # erhöhe die Nummer des Parametersatzes, falls dieser schon existiert
    while os.path.exists(folder_output):
        number_params += 1
        folder_output = os.path.join(folder_root, 'Parameters' + str(number_params))

    os.makedirs(folder_output)
    # Name für die Datei mit den Parametern
    fn_para = os.path.join(folder_output, 'parameters' + str(number_params) + '.asc')
    np.savetxt(fn_para, np.vstack((params_keys, params_values)).transpose(), \
               delimiter='\t', fmt='%s')
    print('Parametersatz', number_params, 'erstellt.')

def DetermineMorphology(morphology):
    """
    morphology : str
        Art der Morphologie

    returns
    -------
    params_keys : list
        Liste der Parameternamen
    params_values : list
        Liste der Parameterwerte
    """
    if morphology == 'polymer':
        params_keys = params_polymer.keys()
        params_values = params_polymer.values()
    elif morphology == 'crystallite':
        params_keys = params_crystallite.keys()
        params_values = params_crystallite.values()
    elif morphology == 'pointgrid':
        params_keys = params_pointgrid.keys()
        params_values = params_pointgrid.values()
    return list(params_keys), list(params_values)

def Hoppingmechanism(hopping):
    """
    Hüpfmechanismus zur Parameterliste hinzufügen.
    """
    # Parameter die für die Hüpfrate benötigt werden
    if hopping == 'MA':
        p = {'nu0': nu0}
    elif hopping == 'marcus':
        p = {'reorg_energy': reorg_energy,
             'J0': J0}
    else:
        raise ValueError('unknown hopping rate')
    params_polymer.update(p)
    params_crystallite.update(p)
    params_pointgrid.update(p)

###############################################################################

folder_root = os.path.join(folder_cluster, folder)

if local_folder:
```

```
    folder_root = 'C:/Daten/TobiasMeier/Simulationen/Tests'

# ------------------------------------------------------------------------------
# Wörterbücher für die Parameter aller möglichen Morphologien
#
# Polymerketten
params_polymer = {
'morphology': morphology,
'load_morphology': load_morphology,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'chain_length': chain_length,
'chain_sigma': chain_sigma,
'chi': chi,
'fill_grid': fill_grid,
'kink': kink,
'kink_length': kink_length,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'NN_hops': NN_hops,
'random_landscape': random_landscape,
'current_conv': check_convergence_criteria,
'sim_steps': sim_steps}

# Kristallite
params_crystallite = {
'morphology': morphology,
'load_morphology': load_morphology,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'crystal_a': crystal_a,
'crystal_b': crystal_b,
'crystal_gamma': crystal_gamma,
'number_seeds': number_seeds,
'boundary_gap': boundary_gap,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'NN_hops': NN_hops,
'random_landscape': random_landscape,
'current_conv': check_convergence_criteria,
'sim_steps': sim_steps}

# Punktgitter
params_pointgrid = {
'morphology': morphology,
'load_morphology': load_morphology,
'temperature': temperature,
'U_Gate': U_Gate,
'U_Drain': U_Drain,
'sigma': sigma,
'trap_layer': trap_layer,
'trap_distribution': trap_distribution,
'stretch_exp': stretch_exp,
'interface_traps': interface_traps,
'trapdensity': trapdensity,
'E_trap': E_trap,
'sigma_trap': sigma_trap,
'chain_length': chain_length,
'chain_sigma': chain_sigma,
'chain_offset': chain_offset,
'x_length': x_length,
'y_length': y_length,
'z_length': z_length,
'dx': dx,
'dy': dy,
'dz': dz,
'hopping': hopping,
'gamma_intra': gamma_intra,
'gamma_x': gamma_x,
'gamma_y': gamma_y,
```

```
'gamma_z': gamma_z,
'energy_model': energy_model,
'd_gate': d_gate,
'epsilon_gate': epsilon_gate,
'epsilon_rel': epsilon_rel,
'NN_hops': NN_hops,
'random_landscape': random_landscape,
'current_conv': check_convergence_criteria,
'sim_steps': sim_steps}

# Hüpfmechanismus hinzufügen
Hoppingmechanism(hopping)

#Auswahl der passenden Parameterliste anhand von gewählter Morphologie
params_keys, params_values = DetermineMorphology(morphology)

# ------------------------------------------------------------------------------

# veränderliche Parameter und Namen dazu
variable_params = []
variable_keys = []
# konstante Parameter und Namen dazu
constant_params = []
constant_keys = []

# Rausfinden welche Parameter veränderlich sind und welche nicht
for i,p in enumerate(params_values):
    # handelt es sich um eine Liste bei den Parametern?
    if isinstance(p, list):
        variable_params.append(p)
        variable_keys.append(params_keys[i])
    else:
        constant_params.append(p)
        constant_keys.append(params_keys[i])
print('veränderliche Parameter', variable_keys)

# alle möglichen Kombinationen der veränderlichen Parameter
for p in itertools.product(*variable_params):
    # konstante und einen Satz veränderlichen Parameter zusammensetzen
    d = dict(zip(variable_keys + constant_keys, list(p) + constant_params))
    if number_trials is None:
        # Anzahl der Ladungsträger aus Gatespannung berechnen
        N = int( (d['epsilon_gate'] * 8.854e-12*d['x_length'] * d['y_length'] *d['dx']
* d['dy'] * 1e-9) / \
                (1.602e-19 * d['d_gate']) * abs(d['U_Gate']) )
        # Anzahl der Trials daraus
        d['number_trials'] = int(np.ceil(number_carriers / N))
    else:
        d['number_trials'] = number_trials
    WriteParameterRecord(d)
```