Unterschrift BetreuerIn

TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Interpolation Methods and Position-Dependent Effective Mass for ViennaWD

ausgeführt am TU Wien - Institute for Microelectronics
der Technischen Universität Wien

unter der Anleitung von
Assoc.Prof. Lado Filipovic
Assoc.Prof. Josef Weinbub

durch

Philipp Haslhofer

Gußhausstraße 27-29 / E360
Vienna Austria

April 8, 2024

Unterschrift StudentIn

Abstract

Modern nanoelectric devices typically require the combination of multiple materials in a highly layered structure. The complex geometries and interfaces are meticulously designed to optimize the electrical performance of such devices. Therefore, it is of utmost importance that simulation tools are able to model these structures and study the behavior of charge carriers at their various interfaces. Among the various available approaches to model the quantum electron transport problem, which are capable to describe such phenomena, the particle-based Wigner function approach, utilized by ViennaWD, stands out. Due to its representation in phase space, this method allows for the adoption of scattering models and analogies from semi-classical transport, thus retaining many classical concepts and notions. This provides important advantages for quantum mechanically simulating electron dynamics.

A quantum-mechanical Wigner-based simulator should, therefore, be able to support (1) imported external quantities, such as the electric potential defined on arbitrary 2D grids, as well as (2) transport domains with different material parameters. These two aspects are represented by (1) an interpolation problem of mapping an externally generated quantity onto the ViennaWD grid structure and (2) the implementation of a position-dependent effective mass to capture the varying charge-carrier mobility in different transport domains. The means by which these two aspects can be introduced to the existing framework are assessed, and the optimal solution is implemented in ViennaWD.

These additions to ViennaWD and their applicability to representative encountered data are evaluated with the help of various simulations. The developed interpolation mechanism is shown to capture a variety of different geometries, allowing for the import of diverse external quantities. Further, proof-of-concept simulations show that the effective mass functionality can be successfully implemented into ViennaWD, enabling the study of cutting-edge nanoelectric devices.

Acknowledgement

First and foremost, I want to thank Assoc.Prof. Lado Filipovic and Assoc.Prof. Josef Weinbub for supervising this thesis. Their guidance and feedback throughout the process were essential to staying on track and crafting every chapter of this document. Further, I want to thank Mihail Nedjalkov, Mauro Ballicchia, and Clemens Etl for our weekly group meetings, which broadened my horizons when it came to understanding the nature of working in the academic field. The preparations for papers that the group published, as well as the discussion of scientific papers published by others, were vital and much-appreciated insights that helped me tremendously when writing this thesis. These meetings expanded my knowledge and increased my appreciation for the effort and passion that go into research.

Last but not least, I want to mention the support from my parents, Rudolf and Regina, as well as my fellow colleagues, who helped me to pursue and finish my studies at the Technical University of Vienna. Without their support, I would not have been able to undertake the challenges of my chosen field of studies and, therefore, finish this very interesting and rewarding journey.

Contents

1 Introduction

Developing semiconductor devices is a complex and expensive process that requires a lot of time and resources. To support these efforts, computational electronics is the field of research that aims to develop and improve computer simulations of semiconductor devices to understand the physics of such systems better and further improve upon the technologies used in the fabrication of semiconductor devices. The field of computational electronics can be seen to be a focus area of the broader field of computational science and engineering, a term defined already in the 1990s, for example, by Stevenson and Panoff [1] as:

> "The interdisciplinary involvement in the identification and elimination of unwarranted assumptions and approximations in scientific models and the complete integration of computation into these models to constitute a whole new scientific technique on a par with hypothesis and experimentation. “

Within the electronics community, simulation tools used in this line of research are often coined under the term technology computer-aided design (TCAD), which refers to the use of computer simulations to design, develop, and optimize semiconductor processing technologies, device structures, and circuits [2]. With the help of TCAD tools, the pursuit of high-performance, high-reliability, and low-power consumption semiconductor devices can be continuously improved, as demanded by the extremely high pace of innovation in micro- and nanoelectronics research. Current transistors, as a representation of one of today's most important semiconductor devices, in modern processors have been made with nano-scale dimensions for more than a decade [3], facing increasingly challenging obstacles with respect to device dimensions and material limitations, requiring novel designs, e.g., gate-all-around field effect transistors [4, 5], novel materials, e.g., 2D materials and

stacks thereof [6], and novel concepts for information processing using quantum effects, e.g., qubits [7]. These developments require quantum transport models to accurately describe the physical reality in these extremely scaled systems [8]. ViennaWD provides a two-dimensional (2D) Wigner particle Monte Carlo simulator developed at the Institute for Microelectronics, TU Wien [9]. The theoretical foundation for the implemented algorithms is the Wigner description from the field of quantum mechanics, allowing the simulation of electron transport dynamics in phase space and the study of decoherence processes [10, 11, 12]. ViennaWD has been widely used to study novel quantum effects in solid-state single-electron systems [13, 14]. However, among the limitations are two key aspects that limit the practical usability of the simulator:

- Input electric scalar potential profile limited to internal grid data structure: No externally provided potential profile can be imported, which is likely defined on a different discretization. Consequently, no interface to external simulation tools is possible, thereby not allowing the simulation of cutting-edge, practically relevant simulation scenarios defined by external tools.

- Single-valued effective mass applies to the entire simulation domain: Only a single material for the entire simulation can be defined. This, obviously, does not allow the simulation of modern heterostructure devices, such as 2D material stacks, which would require the assignment of different effective masses to different parts of the simulation domain.

Therefore, this thesis focuses on overcoming these limitations by advancing the code base accordingly. First, an interpolation routine was implemented to support the loading of externally generated quantities, e.g., an electric scalar distribution. The need for such a capability arises from ViennaWD's inherent design, where the simulation is performed on an equidistant 2D grid structure. Therefore, an arbitrary imported quantity must be defined on the internal simulation grid, which, of course, is not the same grid used by the external source. To this end, different approaches to this interpolation problem were implemented and rigorously evaluated. Second, to enable studying transport channels with different materials, the effective mass, which is a key parameter entering the Wigner transport

equation, was implemented to be spatially dependent and assigned to the point elements of the grid data structure. Together, these two advancements enable the use of ViennaWD for the future study of practically more relevant, multi-material nanoelectronic systems.

This work is structured as follows. Chapter 2, to set the stage, provides a short overview of the Wigner formalism and the defining equation for quantum-mechanical transport phenomena, the Wigner-transport equation (WTE). Furthermore, a brief overview of the simulator is given, including software design and implementation basics. Chapter 3 introduces the theory behind interpolation techniques and discusses different implementation approaches. In Chapter 4, implementing a spatial-dependent effective mass mechanism to the simulator will be discussed, as well as certain implications. Simulations will be discussed using the example of state-of-the-art semiconductor devices. In Chapter 5, the implementations are evaluated with the help of a set of representative simulations. Finally, Chapter 6 summarizes this work and highlights the key contributions.

2 Wigner Background and the Vienna WD simulator

Device TCAD for ultra-scaled devices has evolved into a complex endeavor, necessitating the integration of diverse models across electrical, optical, and thermal domains spanning multiple scales [15]. This drives the need for multiscale, multiphysics simulations and robust coupling of tools and models, particularly in commercial TCAD development. Furthermore, advancements in models, especially in accurately describing physics, propel TCAD development forward. Charge carrier transport models are essential for studying device electrical performance. Still, only a few devices have exploited the underlying quantum mechanical principles, for example, resonant tunneling diodes [16, 17]. However, novel research fields, such as electron quantum optics, exploit the wave nature of electrons for information processing [18]. Understanding and designing quantum devices necessitates sophisticated simulation tools, specifically electron transport simulation in semiconductors, a foundational capability in nanoelectronics research.

The ViennaWD simulator contributes to tackling this challenge by utilizing the Wigner formalism for quantum mechanically modeling electron transport dynamics. The Wigner formalism is an attractive alternative (because of, e.g., reduced computational effort) to conventional modeling approaches based on the Schrödinger equation [19] or non-equilibrium Green's functions [20] and provides an intuitive description of quantum mechanics, allowing the adoption of models (e.g., scattering) and analogies from semi-classical transport. Scattering in quantum transport is imperative for studying decoherence processes of (entangled) electron states, which is pivotal when utilizing quantum effects for device operation [21]. Additionally, time-resolved quantum transport simulations offer insights into the behavior of highly miniaturized circuits dominated by quantum

effects, such as oscillations, which would otherwise not be explainable by classical theory. The Wigner-Boltzmann equation is the sole computationally viable formalism for scattering-aware, time-resolved quantum transport [22]. It has emerged as a cornerstone for comprehensively studying electron transport and decoherence in nanoscale structures. These simulations provide crucial insights into the dynamic behavior of quantum devices, thereby advancing our understanding of nanoelectronic systems and paving the way for developing novel device designs with enhanced functionality and performance [23].

2.1 Transport Models

To adequately describe the interactions of semiconductor devices and nanostructures with their environment through leads/contacts, phonons, or electromagnetic fields, simulation tools must be able to capture these essential phenomena of the system. The effect of these interactions, especially regarding their relative strength compared to modern device dimensions, necessitates the description of charge carriers via non-equilibrium distributions that can be determined by solving a transport equation describing the influence of external forces to obtain the correct statistical properties of the system. A short overview (Figure 2.1) of the essential semi-classical transport and quantum transport models [24] will be given before the Wigner formalism, which bridges the two limiting transport regimes, is introduced, and the Wigner Transport equation is motivated and presented.

The Boltzmann Transport Equation (BTE) (Equation (2.1)) [25] is the central quantity in the realm of semi-classical microscopic transport and describes the evolution of the distribution function $f_b(\mathbf{r}, \mathbf{k}, t)$ of a particle in phase space $\mathbf{r}, \mathbf{k}$ at time t. One of the main advantages of the BTE is that it can be used to describe both the ballistic and the diffusive transport regime.

$$\frac{df_b}{dt} = \frac{\partial f_b}{\partial t} + \frac{d\mathbf{r}(t)}{dt}\nabla_r f_b + \frac{d\mathbf{k}(t)}{dt}\nabla_k f_b = \mathcal{C}\{f_b(\mathbf{r}, \mathbf{k}, t)\} \tag{2.1}$$

But as mentioned previously, in today's world of nano-electronics, the scale of the devices is well below the extent to which classical models adequately describe the physical phenomena within them. Therefore, quantum transport models

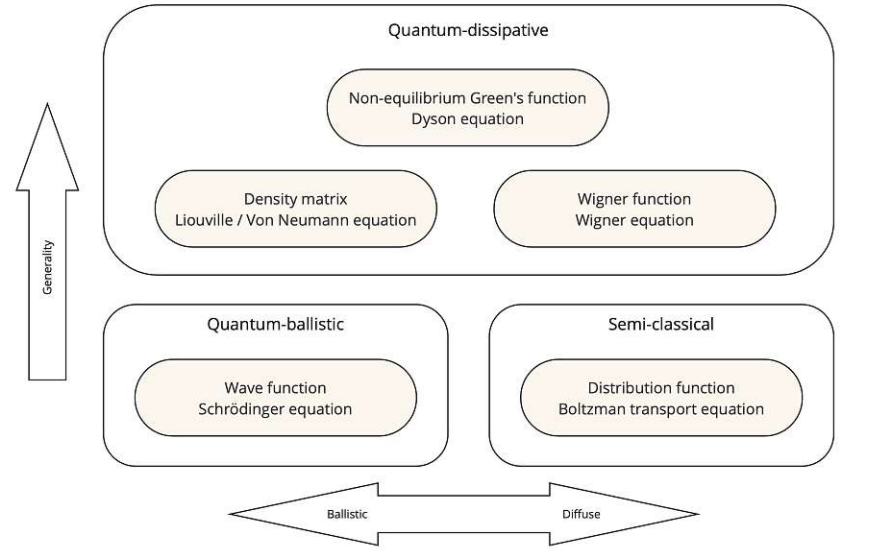


Figure 2.1: Overview of the different microscopic transport models [22].

are necessary to describe the quantum effects of charge carriers in nanoelectronic devices. Figure 2.1 shows some different approaches to the quantum transport problem that have been pursued and will in the following be discussed briefly.

The Schrödinger equation 2.2 as the fundamental equation of motion describing the evolution of a pure quantum state can be adapted to an open system necessary to describe quantum transport [26].

$$\mathrm{i}\hbar\frac{\partial}{\partial t}\left|\psi\right\rangle = H\left|\psi\right\rangle, \tag{2.2}$$

In Eq. (2.2), $\hbar$ denotes the reduced Planck constant, $|\psi\rangle$ is the wave function of the quantum state and H the Hamiltonian operator.
However, the Schrödinger equation is only well-suited to describe ballistic transport, and proposed models that include out-scattering have proven problematic in their numerical implementation [27].

The density operator formalism, also known as the density matrix formalism, is a powerful tool in quantum mechanics, particularly for dealing with mixed states and open quantum systems. The evolution of the density operator ρ as given in equation 2.3,

$$\rho(\mathbf{r},\mathbf{r}',t) = \psi^*(\mathbf{r}',t)\psi(\mathbf{r},t) \tag{2.3}$$

, is governed by the Liouville/Von-Neumann equation

$$\mathrm{i}\hbar\frac{\partial}{\partial t}\rho = [H, \rho], \tag{2.4}$$

where $[H, \rho] = H\rho - \rho H$ denotes the commutator bracket of ρ and H, with ρ the density operator and H the Hamiltonian of the system.

However, applying the density operator formalism to quantum transport has some limitations. Such being the integration of positive-definite scattering operators into the Liouville/Von Neumann equation 2.4 and the non-local nature of the density matrix, which makes the interpretation of the results more difficult [28].

2.1.1 Wigner Transport Equation (WTE)

The Wigner function, related to the density matrix through a unitary Fourier transform, provides a phase space description of quantum mechanics. Analogous to the semi-classical Boltzmann transport equation, the Wigner transport equation describes the evolution of a Wigner function $f_w(\mathbf{r}, \mathbf{k}, t)$ [29] over time. Applying the Wigner transform to the density operator yields the Wigner function,

$$f_w(\mathbf{r}, \mathbf{k}, t) = \int_{-\infty}^{\infty} \mathrm{d}\mathbf{s}\, e^{-\mathrm{i}\mathbf{k}\cdot\mathbf{s}} \rho\Big(\mathbf{r} + \frac{\mathbf{s}}{2}, \mathbf{r} - \frac{\mathbf{s}}{2}, t\Big) \tag{2.5}$$

The Wigner function f_w, defined over the phase space $\mathbf{r}, \mathbf{k}$, encompassing all possible combinations of the position $\mathbf{r}$ and the wave-vector $\mathbf{k}$ assignable to a particle [22], like the distribution function in the Boltzmann case, represents the number of particles per unit volume at time t. However, the Wigner function is not a proper probability density function since it may attain negative values, manifesting the uncertainty relation in the phase space [30]. Regardless of the above statement, the critical property

$$\int\int \mathrm{d}\mathbf{r}\, \mathrm{d}\mathbf{k}\, f_w(\mathbf{r}, \mathbf{k}, t) = 1, \quad \forall t \tag{2.6}$$

of a probability distribution is still retained, which means that physical averages can still be calculated using the same expressions as in the Boltzmann case, which classifies the Wigner function as a so-called quasi-distribution function [22].

To arrive at the Wigner transport equation (WTE), the Wigner transformation is similarly applied to the Liouville/Von-Neumann equation (see Equation (2.4)), which describes the evolution of the density matrix and is given here already in shifted coordinates and with the expanded Hamiltonian operator

$$\frac{\partial}{\partial t}\rho\Big(\mathbf{r}+\frac{\mathbf{s}}{2},\mathbf{r}-\frac{\mathbf{s}}{2},t\Big)=\frac{1}{\mathrm{i}\hbar}\Big\{-\frac{\hbar^2}{2m^*}\frac{\partial^2}{\partial\mathbf{r}\partial\mathbf{s}}+\Big(V\Big(\mathbf{r}+\frac{\mathbf{s}}{2}\Big)-V\Big(\mathbf{r}-\frac{\mathbf{s}}{2}\Big)\Big)\Big\}\rho\Big(\mathbf{r}+\frac{\mathbf{s}}{2},\mathbf{r}-\frac{\mathbf{s}}{2},t\Big) \tag{2.7}$$

, where ρ denotes the density operator, m^* the effective mass of the charge carriers, and V the electric potential. Yielding the evolution equation for the associated Wigner function, the WTE (shown here for the electrostatic case and in the absence of scattering)

$$\frac{\partial}{\partial t}f_w(\mathbf{r},\mathbf{k},t)+\frac{\hbar\mathbf{k}}{2m^*}\frac{\partial}{\partial\mathbf{r}}f_w(\mathbf{r},\mathbf{k},t)=\int \mathrm{d}\mathbf{k}'\,V_w(\mathbf{r},\mathbf{k}'-\mathbf{k},t)f_w(\mathbf{r},\mathbf{k},t) \tag{2.8}$$

where,

$$V_w(\mathbf{r},\mathbf{k},t)=-\frac{1}{\mathrm{i}\hbar(2\pi)^3}\int \mathrm{d}\mathbf{s}\,e^{\mathrm{i}\mathbf{s}\cdot(\mathbf{k}-\mathbf{k}')}\Big\{V(\mathbf{r}-\frac{\mathbf{s}}{2})-V(\mathbf{r}+\frac{\mathbf{s}}{2})\Big\} \tag{2.9}$$

denotes the Wigner potential obtained via a Wigner transform of the electric potential V as shown in Equation (2.7).

Since practical simulations necessitate a finite domain, limits are imposed on the integration of variables. This is achieved by assigning a finite value to the integration variable with $|L| = L$. Applying a finite value to the integration bounds $\pm L$, termed an isotropic coherence length, results in a discretization of the momentum space k, $\mathbf{k}\rightarrow\mathbf{q}\Delta k$.

$$f_w(\mathbf{r},\mathbf{q}\Delta k,t)=\frac{1}{L}\sum_{\mathbf{q}}e^{-i\mathbf{q}\Delta k\cdot\mathbf{s}}\rho(\mathbf{r}-\mathbf{s},\mathbf{r}+\mathbf{s},t) \tag{2.10}$$

$$\Big[\frac{\partial}{\partial t}+\frac{\hbar\mathbf{q}\Delta k}{m^*}\nabla_r\Big]f_w(\mathbf{r},\mathbf{q}\Delta k,t)=\sum_{\mathbf{q}'}V_w(\mathbf{r},\mathbf{q}-\mathbf{q}',t)f_w(\mathbf{r},\mathbf{q}'\Delta k,t) \tag{2.11}$$

$$V_w(\mathbf{r},\mathbf{q}\Delta k)=\frac{1}{\mathrm{i}L\hbar}\int_{-L/2}^{+L/2}\mathrm{d}\mathbf{s}\,e^{\mathrm{i}2\mathbf{q}\Delta k\cdot\mathbf{s}}\{V(\mathbf{r}+\mathbf{s},\mathbf{r}-\mathbf{s})\} \tag{2.12}$$

where $\mathbf{q}$ is an integer multi-index and $\Delta k=\pi/L$, which denotes the resolution of the discretized wave-vector [22].

2.1.2 Motivation for Wigner formalism-based simulations

The Wigner formalism, with its phase-space description, retains many classical concepts and notions, which makes it a convenient approach to describing the transport phenomena characterizing the evolution of charge carriers in nanostructures compared to other quantum-mechanical approaches, such as the Schrödinger equation. Therefore, this allows for the adoption of models (e.g., scattering) and analogies from semi-classical transport. Incorporating Boltzmann scattering models into the Wigner equation yields the Wigner-Boltzmann equation. The Wigner-Boltzmann equation harmonizes the two theories and facilitates a smooth transition from purely quantum (ballistic) to classical (diffusive) transport depicted in Fig. 2.1. The integration of this semi-classical scattering model into the WTE was initially suggested in [31]. It was later justified through a thorough derivation for both phonon [32, 33] and impurity scattering, demonstrating that the semi-classical scattering models can be seen as a limiting case of comprehensive quantum models [34]. The Wigner formalism allows for a semi-classical depiction of extended contact regions while also providing a quantum representation of a device's active region [35]. Beyond computational electronics, the Wigner function is widely utilized in numerous research areas, including quantum physics, quantum optics, and quantum information processing [36, 37].

2.1.3 Stochastic Solution Techniques for the WTE

The Wigner-Boltzmann equation (WBE) is a partial differential equation (PDE) in phase space, which is difficult to solve analytically. Due to this high dimensionality, deterministic solutions are computationally expensive and require large amounts of memory, making deterministic solution methods of the WBE particularly challenging even on today's hardware.

$$\left(\frac{\partial}{\partial t} + v_g(\mathbf{k})\right) f_w(\mathbf{r},\mathbf{k},t) = \int \mathrm{d}\mathbf{k}' \left(S(\mathbf{r},\mathbf{k},\mathbf{k}') + V_w(\mathbf{r},\mathbf{k}'-\mathbf{k})\right) f_w(\mathbf{r},\mathbf{k}',t) \tag{2.13}$$

$$-\lambda(\mathbf{r},\mathbf{k}) f_w(\mathbf{r},\mathbf{k},t) \tag{2.14}$$

$$\lambda(\mathbf{r},\mathbf{k}) = \int \mathrm{d}\mathbf{k}'\, S(\mathbf{r},\mathbf{k}',\mathbf{k}) f_w(\mathbf{r},\mathbf{k}',t) \tag{2.15}$$

, where the first term denotes in-scattering and the second term denotes out-scattering at rate λ.

The WBE, as shown in equation Eq. (2.13), can be transformed into an ordinary differential equation (ODE) by introducing Newton trajectories [22]. The resulting equation parameterized by the time variable τ can then be formally integrated over the interval $\tau = [t, t_0]$ yielding the following equation:

$$f_w(\mathbf{r}, \mathbf{k}, t_0) = f_{w,i}(\mathbf{r}, \mathbf{k}) e^{-\int_t^{t_0} \mu(\mathbf{R}(y), \mathbf{k}) \mathrm{d}y} \tag{2.16}$$

$$+ \int_t^{t_0} \mathrm{d}t' \Gamma(\mathbf{R}(\tau), \mathbf{k}, \mathbf{k}', \tau) f_w(\mathbf{R}(t'), \mathbf{k}', t') e^{-\int_t^{t_0} \mu(\mathbf{R}(y), \mathbf{k}) \mathrm{d}y} \tag{2.17}$$

$$\Gamma(\mathbf{R}(\tau), \mathbf{k}, \mathbf{k}', t) = S(\mathbf{R}(\tau), \mathbf{k}, \mathbf{k}') + V_w(\mathbf{R}(\tau), \mathbf{k}' - \mathbf{k}) + \gamma(\mathbf{R}(\tau)) \delta(\mathbf{k} - \mathbf{k}') \tag{2.18}$$

, here $\gamma(\mathbf{R}(\tau))$ is the scattering rate associated with the Wigner potential and $\mathbf{R}(\tau)$ the trajectory of position.

This represents the integral form of the WBE. By recasting it as a Fredholm integral equation, the concept of solving the WBE using Monte Carlo methods [38] is introduced. Integral equations in the form of Fredholm integrals can characterize a broad range of physical phenomena. Over time, a robust theory has developed around solving these Fredholm integral equations using Monte Carlo algorithms.

The computational objective involves determining the statistical average of a given physical quantity, denoted by $A(\mathbf{r}, \mathbf{k})$, at a specific time T by employing the Wigner function:

$$\langle A_T \rangle = \int \mathrm{d}\mathbf{r} \int \mathrm{d}\mathbf{k} \, f_w(\mathbf{r}, \mathbf{k}, T) A(\mathbf{r}, \mathbf{k}) \tag{2.19}$$

Expanding Eq. (2.19) into a Neumann Series allows the physical quantity to be obtained by stochastic sampling of the Neumann series using numerical particles [38]. A particle progresses through free-flight and scattering until time T, selecting a term in the series. The selected term's contribution is determined by sampling its associated integral. An algorithm for this task propagates numerical particles along the trajectories, scatters them to different wave vectors, or spawns additional particles. This approach, mirroring free-flight and scattering in semi-classical Monte Carlo simulation, allows using established algorithms.

Wigner trajectories, which are defined via this formalism, where the action of the Wigner potential operator is interpreted as scattering, give rise to the signed-particle model where the Wigner potential is interpreted as a signed particle generator [10].

2.2 Wigner Signed Particle Solution Algorithm

As mentioned in the previous section, the Wigner-Boltzmann equation interpreted as a Neumann series gives rise to the Wigner signed-particle model, as statistical means of an arbitrary physical quantity can be represented by stochastic sampling of the Neumann series using numerical particles. Here, as a variation of an affinity model (for an overview of particle models used in quantum electron transport, see [8]), only integer affinities with the values ± 1 are considered in a generation event. This leads to peculiarities and intricacies regarding the implementation and parallelization of this algorithm. The following section will give an overview of the algorithm's structure and, unless otherwise stated, will follow the description of Ellinghaus [22].

2.2.1 Program Structure

The program can be roughly divided into three general parts: Pre-processing, Simulation, and Post-processing. The program structure, specifically the simulation part and the evolution algorithm, is illustrated in Fig. 2.2.

Pre-processing

Pre-processing is concerned with generating the required input files for the simulation. This is also where part of this thesis focuses on. In particular, quantities must be mapped to the regular 2D mesh used in the simulator to incorporate arbitrarily modeled or experimentally measured quantities needed in the simulation. Therefore, it is necessary to have an easy-to-use pre-processing tool performing this interpolation and mapping task.

Simulation

The simulation part is the actual simulation of the Wigner transport equation. This is done by invoking a Monte Carlo method and the Wigner signed particle approach. The basic steps of the solution process are initializing the system from the input, setting up the geometry and potential profile, and setting the parameters. To study and understand the system's evolution through a change in material parameters, the simulation also needs to set up the effective mass profile for the system. This is the second part of the thesis focused on. The interpolation tool is also introduced here within the simulation setup to allow for a more automated approach to the task mentioned earlier.

The simulation's main part is the evolution of the particles, which repeats within a time loop until the total simulation time is reached. Since the problem is high-dimensional and the simulation is computationally expensive, it is running in a distributed fashion. Fortunately, semi-classical Monte Carlo codes parallelize efficiently due to independent particles, but Wigner Monte Carlo codes require synchronized communications for the critical annihilation step. The ViennaWD [39] implementation uses an MPI-based domain decomposition on a distributed memory architecture, dividing the global simulation domain and particle ensemble into uniformly sized subdomains and sub-ensembles. Each MPI process manages a subdomain and its corresponding sub-ensemble to minimize communication demand from scattering events. Since the algorithm works via a Monte Carlo method, the simulation is stochastic in nature, and therefore, the distribution process boils down to non-interacting particles, sampling the initial distribution function, being distributed to the subdomains. Post-annihilation, each process checks for and exchanges particles in overlapping subdomain boundaries, reducing the particle ensemble size and communication load.

Therefore, the time-loop consists of the evolution, growth prediction, annihilation, and particle transfer steps as depicted in Fig. 2.2.

Initialization: Initially, the distributor process loads the inputs that describe the geometry and other parameters, such as the electric potential and the effective mass. It then sets up an ensemble of N particles, which represent the initial condition of the evolution problem, by assigning appropriate position and momentum

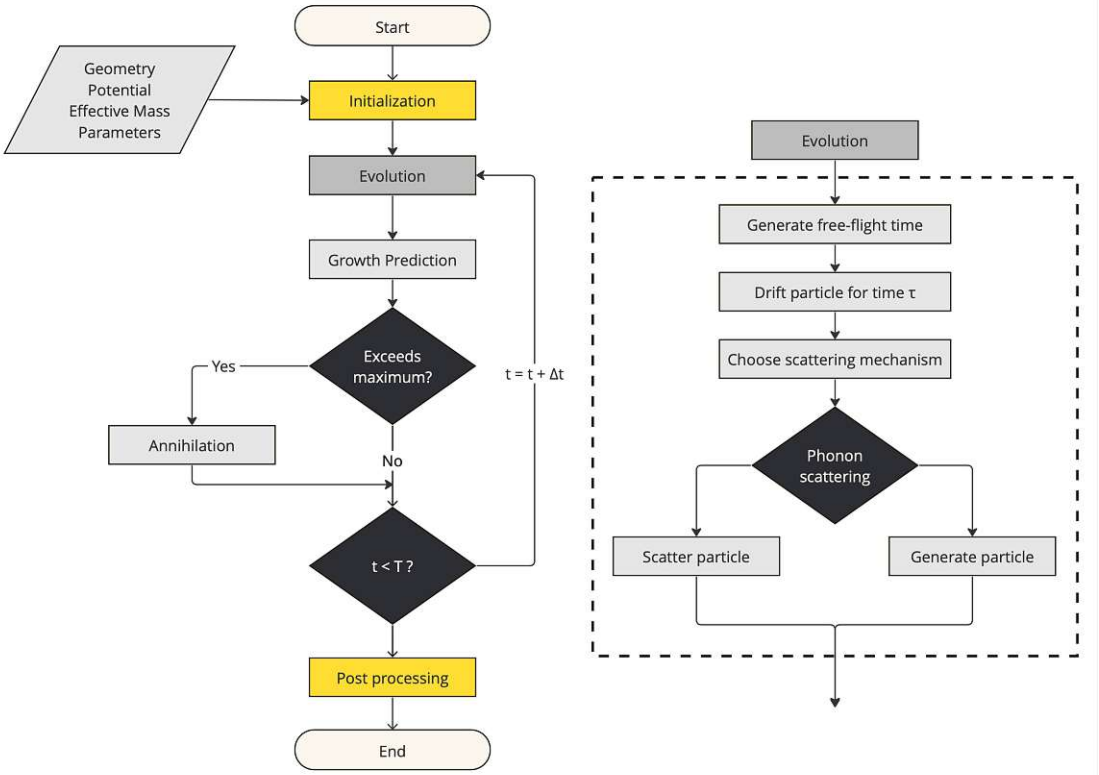


Figure 2.2: Flowchart of ViennaWD, based on Ellinghaus [22].

values. Following this, the distributor process distributes these particles to the receiver processes based on the domain decomposition, along with the potential profile, effective mass profile, and additional global parameters. Each process subsequently initializes localized versions of the necessary data structures using the initial values specific to its subdomain. A commonly used initial condition is the Gaussian minimum uncertainty wave packet.

$$f_w(\mathbf{r}, \mathbf{q}) = \mathcal{N} exp\left[-\frac{(\mathbf{r} - \mathbf{r_0})^2}{\sigma^2}\right] exp\left[-(\mathbf{q}\Delta k - \mathbf{k_0})^2\sigma^2\right] \tag{2.20}$$

where $\mathbf{r_0}$ and $\mathbf{k_0}$ represent the mean position and the mean wave-vector, respectively; σ is the standard spatial deviation and $\mathcal{N}$ represents a normalization constant.

Evolution: Each process performs the evolution of its ensemble of particles for a single time step in the evolution step. Each particle separately evolves in time by the drift and scattering steps. During the drift phase, the particle propagates a Newtonian trajectory with no forces acting on it, and the wave vector remains constant. The new position is calculated by:

$$\partial \mathbf{r} = \frac{\hbar \mathbf{q} \Delta k}{m^*} \min\{\tau, \delta t\} \tag{2.21}$$

This implies that the particle continues to drift until either the current time step concludes, δt represents the remaining time in the time step Δt, or the next scattering event occurs, symbolized by the free-flight time τ. The value of τ is established by generating a uniformly distributed random number r, a characteristic feature of the Monte Carlo method. As per Eq. (2.21), the particle is first propagated and then scattered. A scattering event is selected, again through a random variable r, from a normalized scattering table. This event could be either a phonon-scattering event or a particle-generation event. In the former case, the particle is scattered according to the chosen scattering mechanism, altering its wave vector $\mathbf{k}$. In the latter case, two additional particles with signs ± 1 and wave-vectors $\mathbf{k} \pm \mathbf{l}$ are generated, with the offset $\mathbf{l}$ determined by the Wigner potential.

Growth Prediction: In the growth prediction step, each process predicts the growth of its ensemble of particles after the evolution step. This becomes necessary as continuous particle generation results in an exponential increase in the number of particles present in the simulation. This would lead to an infeasible computational burden; therefore, an annihilation step is later introduced to reduce the number of particles.

To determine if this annihilation procedure will be performed in a subsequent time step, each process performs a growth prediction for its sub-ensemble of particles after the evolution step and compares if the number of particles exceeds a specific maximum. It is advisable to overestimate the particle increase. Therefore, the maximum value of the generation rate γ for all particles is used, which yields an upper bound on the particle growth:

$$N_{t+\Delta t} = N_t \Big(1 + \max_i \gamma(\mathbf{r_i}) \Delta t\Big) \tag{2.22}$$

where $N_{t+\Delta t}$ and N_t represent the number of particles at times $t + \Delta t$ and t, respectively, and Δt is the time increment between two consecutive time steps.

Annihilation: In the annihilation process, all processes must simultaneously execute their local annihilation step for particles within the subdomain, which is crucial for maintaining the global statistics of the Wigner function. All processes must reciprocate if any process necessitates an annihilation step based on its local growth prediction. This synchronized operation is ensured by communicating each process's growth prediction result to the distributor process via an annihilation flag. The distributor process collects these flags and, if any are true, broadcasts a global annihilation flag, prompting all processes to perform an annihilation step. If all flags are negative, no annihilation occurs. The actual annihilation is executed based on phase space cells, each associated with a specific wave vector value as per the semi-discrete Wigner equation (2.13). If two particles with opposite signs are in the same cell, they annihilate each other and are removed from the ensemble, as all particles within a cell are considered identical and indistinguishable because of the Markovian nature of the Monte Carlo method.

Particle Transfer: After the annihilation step, a synchronization barrier ensures that all transfers of particles located in the overlapping boundaries of the subdomains are complete before the next time step commences. Performing the transfer of particles after an annihilation step dramatically reduces the size of the particle ensemble to be transferred, which is beneficial for the communication burden.

Post-Processing

The distributor process does not issue a global reduction step to collect the resulting data to avoid a central communication bottleneck at the end of the simulation. Instead, at each output time step, each process writes the simulation results of each subdomain to disks locally. This design decision necessitates merging the simulation results, handled by external scripts, in the post-processing step. After all data has been merged, different post-processing steps, such as analysis, examination, and evaluation, can then take place to reproduce the desired physical quantities and visualize the simulation results.

2.3 Parallel Computing

Optimizing the numerical calculations in scientific computation is paramount to obtaining simulation results in a reasonable amount of time. Since the increase in clock speed of single-core processors has stagnated [40] since the start of the millennium, parallel computation is now the primary method to speed up or even enable expensive computational tasks.

In principle, Monte Carlo methods stand out for their "embarrassingly parallel" nature, which indicates a high level of parallel efficiency that can be readily achieved. This attribute arises from handling smaller subensembles independently across separate computational units without necessitating communication. However, parallelizing the here-considered Wigner Monte Carlo implementation encounters complexities due to the annihilation step, which poses challenges on two fronts. The annihilation step mandates synchronization among computational units. This introduces a need for communication and synchronization among computational units, detracting from the ideal scenario of independent parallelization. The enormity of the numerical particles in the simulated particle ensemble, typically ranging from 10^6 to 10^8, underscores the need for a parallel solution scheme to attain practical simulation runtimes. Consequently, parallelization techniques become indispensable to enhance application performance.

Modern system architectures for high-performance computing (HPC) most commonly consist of several nodes, each comprising several processing units [41]. These processing units can typically be a multi-core central processing unit (CPU), a graphics processing unit (GPU), or a specialized accelerator such as a field-programmable gate array (FPGA). An efficient utilization of high-performance computation resources requires consideration of the memory layout and the communication between the processing units. Since large (HPC) applications mainly involve a distributed memory system, the following section provides a brief overview of the Message Passing Interface (MPI), one of the most commonly used parallelization techniques in scientific computing and used for the parallel execution of ViennaWD.

Message Passing Interface (MPI) The stochastic solution method of ViennaWD is considered to necessitate data transfers between the computing units since they cannot access the same memory on a distributed memory system. MPI [42] is a standard that the MPI Forum maintains. It provides a communication layer for so-called processes, and software vendors provide different implementations of the standard. An MPI process manages the scheduling of operations and the allocation of processor resources [43]. Each process is assigned to a distinct rank within an MPI communicator with which it can be uniquely identified. A straightforward approach initializes an MPI program, and the workload is distributed using the distributor-receiver model. Here, one process called the distributor manages the input and output, broadcasts instructions, receives results from the receivers, and performs reduction operations at the end of a simulation. The receivers, including the distributor process, perform their instructions in parallel before returning the results to the distributor process.

The distributor-receiver model is also used in ViennaWD (Section 2.2.1), with an important distinction being that the distributor process does not perform the reduction step at the end of the simulation, but this is handled by external scripts, which collect the results from the individual processes and merge them into a single file as described in the paragraph Post-Processing in Section 2.2.1.

3 Interpolation

Interpolation plays a fundamental role in various scientific and engineering applications, serving as a tool to estimate values within a set of known data points. In computational tasks, interpolation routines are indispensable for generating continuous functions from discrete data, facilitating analysis, visualization, and prediction. The aim of implementing an interpolation routine into ViennaWD is to provide the possibility to perform a mapping of imported, arbitrary quantities needed for a simulation onto the simulator's grid data structure as described in Section 2.2.1.

The focus will be on importing an electric potential or effective mass distribution into ViennaWD since those are the main two quantities that will regularly be provided externally through experimental measurements or modeling. This problem can be tackled by first interpolating the quantity on the external grid, also often called *knots* or *support knots* in the mathematical literature [44]. This results in an interpolating function whose values are known not only at the support knots but also in the total domain of the quantity. Then, the interpolating function is evaluated at the desired ViennaWD grid points. The following sections will briefly introduce the theory behind interpolation, provide a short overview of the interpolation techniques used in this thesis, explain the reasoning behind the choice of interpolation techniques, and provide a short overview of the implementation of the interpolation routines.

3.1 Interpolation Background

Interpolation serves as a fundamental concept in mathematics, providing a means to estimate values between known data points. It encompasses a diverse range of

techniques aimed at constructing continuous functions or curves that pass through or approximate given discrete data points. The theoretical foundation of interpolation draws upon various mathematical principles, including polynomial interpolation, spline interpolation, and radial basis function interpolation, among others. These methods leverage mathematical constructs such as polynomials, piecewise functions, and radial basis functions to interpolate data points and approximate the underlying behavior of a function or dataset. Through interpolation, data can be analyzed, predicted, and visualized with enhanced accuracy and precision, making it a cornerstone of computational sciences. Though the term interpolation is used in a variety of contexts, its core objective is the following [45]:

Definition. Given $(x_i, f_i), i = 0, \ldots, n$, find

$$p \in \mathcal{K}: \quad p(x_i) = f_i, \qquad i = 0, \ldots, n \tag{3.1}$$

In other words, the goal is to find an interpolating function p, belonging to some fixed class of functions $\mathcal{K}$ that are defined at least on $\Delta = [a, b]$ (e.g., $f \in \mathcal{P}_n$ the set of algebraic polynomials of degree $\leq n$), that matches the given data points (x_i, f_i) for $i = 0, \ldots, n$, where $p(x_i) = f_i$ are given.
Here, x_i is a set of points, and f_i are the function values at those points. Applications are, for example:

- "Extrapolation": typically $f_i = f(x_i)$ for an (unknown) function f. For $\bar{x} \notin x_0, \ldots, x_n$ the value $p(\bar{x})$ yields an approximation to $f(\bar{x})$.
- "Dense output/plotting of f", if only the values $f_i = f(x_i)$ are given (or, e.g., function evaluations are too expensive)
- Approximation of f: integration or differentiation of $f \rightarrow$ integrate or differentiate the interpolating function p

As motivated in the introduction to this chapter, the second example listed above reflects the use case for this work. The interpolation of an arbitrary quantity to the desired parameters of the simulation grid is an application of interpolation in the sense of a dense output of f.

Although the interpolated arbitrarily varying quantity considered will not be represented as an analytical function on the domain, it will still be referred to as f in the following. The function f is given by a set of points (x_i, f_i) for $i = 0, ..., n$, where $f_i = f(x_i)$ are given.

Many interpolation methods exist, each tailored to specific requirements such as accuracy, computational efficiency, and smoothness of the interpolated function. Classical techniques include [45]

- 0-th Order Interpolation: Constant interpolation
- 1-st Order Interpolation: Linear interpolation
- n-th Order Interpolation: Polynomial interpolation

While these methods are straightforward and widely applicable, they may suffer from limitations such as overfitting or oscillations, especially with sparse or noisy data [46]. To address these challenges, more advanced interpolation techniques have been developed [47]. Splines, for instance, provide a flexible and smooth interpolation by fitting piecewise polynomial functions to subsets of data points. Cubic splines, in particular, are widely used due to their simplicity and ability to maintain smoothness while passing through all data points [44].

In addition to spline-based methods, radial basis function interpolation offers an alternative approach [48], employing localized basis functions centered at each data point to construct the interpolated function. This technique is particularly effective for irregularly spaced data or when the underlying data distribution is not well-behaved.

Furthermore, machine learning-based interpolation methods, such as Kriging and Gaussian processes [49], have gained popularity for their ability to capture complex relationships in data while providing uncertainty estimates. These techniques leverage statistical models to interpolate data points and make predictions based on the underlying covariance structure. However, these are not further considered in this work but provide alternative avenues for future research.

Overall, the choice of interpolation routine depends on the characteristics of the data, the desired accuracy, and computational constraints. In practice, combining

multiple methods or adapting existing techniques to specific problem domains often leads to optimal interpolation results. The following sections will provide a short overview of the interpolation methods investigated in this thesis.

3.1.1 Piece-wise interpolation / Splines

Since the problem considered is not to interpolate an analytical function f but rather an arbitrarily varying quantity, there is no imperative for using polynomial interpolation routines of any order standalone of the shortcomings as mentioned above. Therefore, spline interpolation, where for each subset of points to be interpolated through, a polynomial of order k is calculated, which in the most abstract case can be of any order necessary to fit the data, is a promising approach.

Splines are piece-wise polynomials on a partition Δ of an interval $[a, b]$. The partition Δ is described by the knots $a = x_0 < x_1 < \cdots x_n = b$. The elements in this partition between the knots are denoted by $I_i = (x_i, x_{i+1}), i = 0, \cdots, n-1$. For a partition Δ, described by the knots $x_i, i = 0, \ldots, n)$ and $p, r \in N_0$, the spline space $S^{p,r}(\Delta)$ is defined as

---

Definition.

$$S^{p,r}(\Delta) := \left\{ u \in C^r([a,b]) \; |u|_{I_i} \in \mathcal{P}_p \quad \forall i \right\} \tag{3.2}$$

Given values $f_i, i = 0, \cdots, n$, $s \in S^{p,r}(\Delta)$ is said to be the interpolating spline if

$$s(x_i) = f_i, \qquad i = 0, \ldots, n$$

The classical cubic spline space is given by the choices $p = 3$ and $r = 2$, meaning that the piecewise polynomials are of third order; therefore, in $\mathcal{P}_3$ and that the overall interpolating function is twice continuously differentiable therefore in $\mathcal{C}^2$. The interpolation problem is therefore:

Definition.

$$\text{Given } f_i, \quad i = 0, \ldots, n, \tag{3.3}$$

$$\text{find } s \in S^{3,2}(\Delta) \tag{3.4}$$

$$\text{such that } s(x_i) = f_i, \quad i = 0, \ldots, n \tag{3.5}$$

Since Eq. (3.5) represents a system of $n+1$ equations and $dim S^{3,2}(\Delta) = n+3$ there have to be additional constraints imposed. Equation (3.5) yields $n+1$ interpolation conditions. Hence, two more conditions have to be imposed. These two extra conditions are selected depending on the application. One of the following four choices is typically made [45]:

1. Complete/clamped spline: The user provides two additional values $f_0', fn' \in \mathbb{R}$ and imposes the following two additional conditions:

$$s'(x_0) = f_0', \quad s'(x_n) = f_n'. \tag{3.6}$$

2. Periodic spline: one assumes $f_0 = f_n$ and imposes additionally

$$s'(x_0) = s'(x_n), \quad s''(x_0) = s''(x_n). \tag{3.7}$$

3. Natural spline:

$$s''(x_0) = 0, \quad s''(x_n) = 0. \tag{3.8}$$

4. "not-a-knot condition": one requires that the third derivative (jerk) of s at the knots x_1 and x_{n-1} to be zero:

$$\lim_{x \to x_1-} s'''(x) = \lim_{x \to x_1+} s'''(x), \lim_{x \to x_{n-1}-} s'''(x) = \lim_{x \to x_{n-1}+} s'''(x) \tag{3.9}$$

In particular, the spline interpolation problem is uniquely solvable in each of these cases.

3.1.2 Radial Basis Functions

Radial Basis Functions (RBFs) stand as a formidable tool in the realm of interpolation, offering a versatile and powerful approach to approximating unknown functions from scattered data points [48]. Their widespread adoption stems from their unique ability to capture complex relationships between data points while circumventing some of the limitations associated with traditional interpolation techniques [50]. RBFs are especially known for their use in mesh-free interpolation. Although the interpolation problem discussed is not inherently mesh-free, the

added convenience and reassurance of this functionality are not to be overlooked. RBF interpolation originated in the 1970s [51] and has since been successfully used in a variety of fields such as geophysics, computer graphics, medical imaging, finance, environmental modeling, aerospace engineering, and machine learning [52]. This highlights the versatility of RBFs in diverse interpolation scenarios.

The interpolant takes the form of a weighted sum of RBFs where the approximating function $y(\mathbf{x})$ is represented as a sum of n RBFs, each associated with a different center $\mathbf{x}_i$, and weighted by an appropriate coefficient λ_i. Therefore, the interpolation problem takes the form:

Definition.

$$\text{Given } f_i, \quad i = 0, \ldots, n, \tag{3.10}$$

$$\text{find } s(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^d \tag{3.11}$$

$$\text{such that } s(\mathbf{x}_i) = f_i, \quad i = 0, \ldots, n \tag{3.12}$$

$$s(\mathbf{x}) = \sum_{i=1}^{n} \lambda_i \phi(|\mathbf{x} - \mathbf{x}_i|), \quad \mathbf{x} \in \mathbb{R}^d \tag{3.13}$$

$$\text{with} \quad \sum_{i=1}^{n} \lambda_i \phi(\mathbf{x}_j - \mathbf{x}_i) = f_j \tag{3.14}$$

Some classical choices for RBF kernels can be seen in Table 3.1 with $r = \|\mathbf{x} - \mathbf{x}_i\|$.

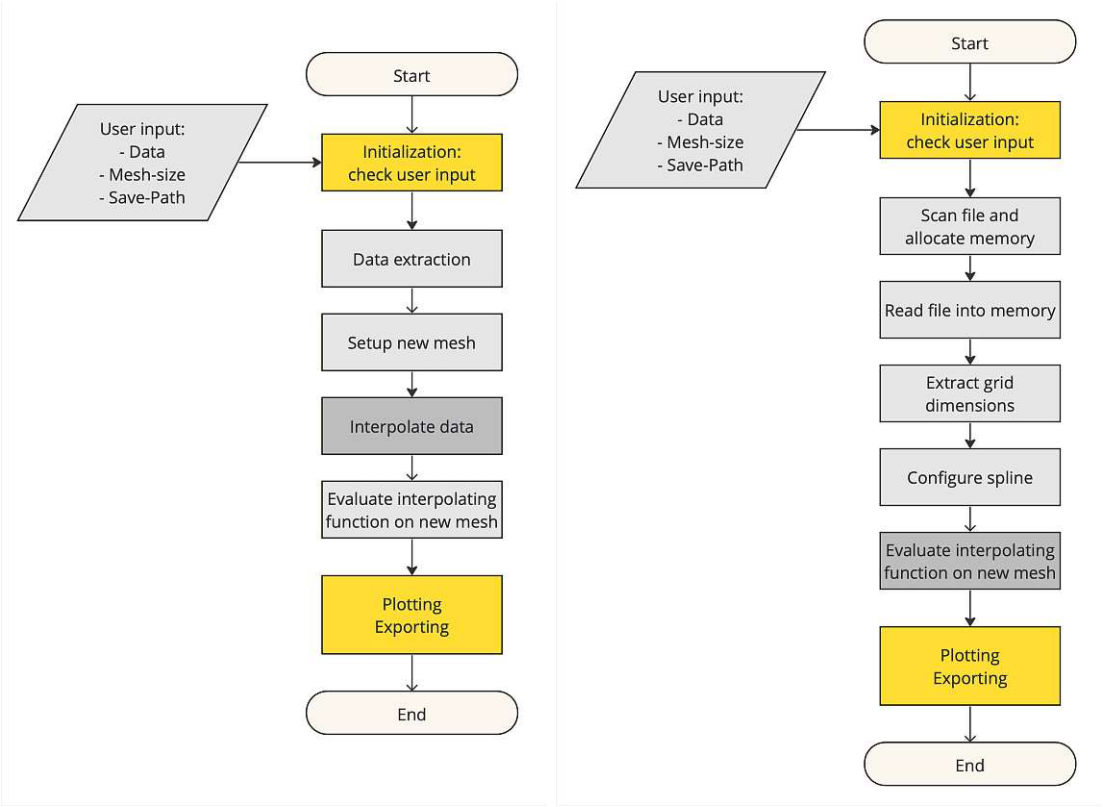
Radial Basis Function	$\phi(\mathbf{x})$
Gaussians	$e^{-(cr)^2}$
Polyharmonic	r^{2k-1}
	$r^{2k}log(r)$
Mulitquadratics	$\sqrt{r^2 + c^2}$

Table 3.1: Classic types of RBFs

3.2 Implementation

Considering the circumstance that the externally provided quantities will, in general, not represent an analytical function and further that we can not make any assumptions on the sampling of the quantity, meaning we cannot assume whether the provided quantity will satisfy any particular conditions as to spacing or density, the choice was made to use spline interpolation techniques and RBFs going forward. Bivariate spline interpolation and RBF interpolation were chosen as they are of considerable interest, in particular, in scattered data fitting, the construction and reconstruction of surfaces [53, 48] and further, since there exist many well-maintained and optimized libraries that have these methods of interpolation implemented. Two choices arise for the implementation. They are (1) integrating an implementation routine into the simulator and (2) a standalone implementation used in a pre-processing step. For the standalone implementation, it was decided to work with *Python* [54] as it is already in use in the simulator's post-processing step. For the implementation focused on integration within Vienna WD itself, it is therefore evident that it was implemented in C [55], as this is the programming language that the Vienna WD simulation kernel is written in. A readily available and well-maintained library for *Python* is the *SciPy* library [56], which offers an interpolate module with all the discussed methods in place. In C, the off-the-shelf available resource for interpolation is the *GSL* library [57]. However, for bivariate interpolation, *GSL* only offers the possibility of spline interpolations of linear and cubic order. Therefore, these will be the only methods compared to the *Python* implementation. In the following, both approaches will be presented, with a discussion of the implications on the workflow of each approach and the findings presented later in Chapter 5. The workflow of each implementation is visualized in Fig. 3.1 as a flowchart and will be used in the following to present the algorithms and explain their workings. A flowchart is a type of diagram that represents a workflow or process [58], in this case, a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. In Fig. 3.1, trapezoidal nodes describe input from the user that is necessary when invoking the program, and yellow nodes describe the sections where the program interacts with the outside through input from the user or output to the operating system. Grey nodes

describe the different stages within the program, where the interpolation step is highlighted in a darker shade.



(a) Python (b) C

Figure 3.1: Flowcharts describing the program algorithm for the implementation of different interpolation routines in *Python* (a) and *C* (b).

3.2.1 Interpolation using *SciPy* in *Python*

As previously mentioned in Section 3.2, two different types of interpolation within the *Python SciPy* library are looked into. These are the previously mentioned spline interpolations where the *SciPy* library offers the `RectBivariateSpline` method for interpolating an arbitrary quantity on a rectangular mesh. The other interpolation method investigated is the RBF interpolation, where *SciPy* provides the `RBFInterpolator` method.

The implementation of both methods will be presented collectively in the following, while only the key differences of the specific method used for the interpolation itself will be pointed out. The implementation of the interpolation routine was done as standalone *Python* script `interpolation.py` and further as *Python* notebook [59] `interpolation.ipynb`. Since both implementations are identical in their functionality, only the standalone script will be discussed in the following. The script is divided into three parts:

1. The first part is concerned with the import of the necessary modules and data.
2. The second part is concerned with the interpolation of the quantity using *SciPy* routines from `scipy.interpolate`.
3. The third part is concerned with plotting and exporting the interpolated quantity.

Import of modules and data

The import of modules and user data is visualized in Fig. 3.1a as the yellow node at the top of the flowchart. The modules imported are shown in Listing 3.1:

- `os`: for extraction of file arguments
- `sys`: for the handling of command line arguments
- `numpy`: for the handling of arrays
- `scipy.interpolate`: for the interpolation routines
- `matplotlib.pyplot`: for the plotting of the interpolated quantity

```
1  import os
2  import sys
3  import numpy as np
4  from scipy.interpolate import RectBivariateSpline as RBS
5  from scipy.interpolate import RBFInterpolator as RBF
6  import matplotlib.pyplot as plt
```

Listing 3.1: Import of necessary modules

The first module to be imported is `os`. It is used to discern whether the command line argument for the input file has the correct extension. `sys` is used to provide the functionality of using command line arguments within the *Python* program. *NumPy* [60] included as `numpy`, provides essential numerical computing capabilities to *Python*. Specifically, it is an array object capable of handling multiple dimensions alongside a plethora of associated objects. Complementing this, *NumPy* boasts an extensive collection of functions tailored for swift operations on arrays encompassing mathematical, logical, sorting, selection, input/output operations, and beyond. The sub-package `scipy.interpolate` is part of the *SciPy* library. *SciPy* is a comprehensive collection of mathematical algorithms and convenience functions built on top of *NumPy*, adding high-level commands and classes for data manipulation and visualization. Specifically, `scipy.interoplate` provides us with interpolation classes, functions, and the accompanying evaluation methods. `matplotlib.pylot` provides the visualization interface from *Matplotlib* [61].

After importing all necessary libraries and packages, the script proceeds to declare all the variables needed for the interpolation routine, depending on the command line arguments. This is done using the length of `sys.argv` and the command line arguments themselves, as shown in Listing 3.2. The command line arguments are:

- `sys.argv[1]` `data`: Data to be interpolated
- `sys.argv[2]` `working path`: Path to write results to
- `sys.argv[3]` `mesh_size`: Mesh size for the evaluation

```
1 print(f"USING {sys.argv[1]} WITH MESH SIZE {sys.argv[3]}
2         WRITING TO {sys.argv[2]}")
3 data = np.genfromtxt(sys.argv[1], delimiter=' ',
4                       skip_header=1)
5 working_path = sys.argv[2]
6 mesh_size_x = float(sys.argv[3])
7 mesh_size_y = float(sys.argv[3])
```

Listing 3.2: Data import from command line arguments

The data to be interpolated is read from the file given in `sys.argv[1]` using `numpy.genfromtxt` and stored in the variable `data`. This is represented in Fig. 3.1a by the grey, rectangular node after the yellow initialization node. Data is assumed to arrive as a CSV, as used in the simulator. This is checked, and if the provided file is not of the right type, the program raises an error and returns as shown in Listing 3.3.

```
1 if file_extension.lower() != '.csv':
2     print("Error: The file is not a CSV file.")
3     sys.exit(1)
```

Listing 3.3: File extension check

The working path is given in `sys.argv[2]` and is used to write the interpolated quantity to the filesystem. To construct the evaluation grid, the mesh size is given in `sys.argv[3]` and is used to create the evaluation grid for the interpolation routine. Though the mesh size is given as a single value, the interpolation routine could support different mesh sizes in the x and y directions. Therefore, the mesh size is split into two values `mesh_size_x` and `mesh_size_y`, which are then used to create an evaluation grid. This is done to future-proof the routine for the case that different mesh sizes in the x and y directions are needed at a later point in the development of ViennaWD.

Interpolation

The second part of the script is concerned with the interpolation using the *SciPy* routine `RectBivariateSpline` or the *SciPy* routine `RBFInterpolator` provided in the `scipy.interpolate` package. To this end, the routines require the grid points or support knots as well as the values of the quantity considered at those grid points. Therefore, the data is later split into the x and y values of the quantity and the quantity values themselves according to the respective requirements of the routines `RectBivariateSpline` and `RBFInterpolator`.

The data points of the interpolation data are stored in the variable `data`, and the evaluation grid is created using the mesh size given in `sys.argv[3]`. Therefore, the x and y values are scanned for their minimum and maximum values, and the evaluation grids are created using the mesh sizes, represented in Fig. 3.1a by the

second light grey, rectangular node. This is done via the `numpy.arange` function, which creates an array of values from a given start value to a given end value with a given step size, which in our case is the mesh size. The new data points are then stored in the variables `Xnew` and `Ynew` and are used to further create the evaluation mesh grids via `numpy.meshgrid`, where `numpy.meshgrid` returns a list of coordinate matrices from coordinate vectors. This is shown in Listing 3.4, where further the mesh grid is then saved as `xnew` and `ynew`.

```
x_min, x_max = data[:,0][0], data[:,0][-1]
y_min, y_max = data[:,1][0], data[:,1][-1]

Xnew = np.arange(x_min, x_max+mesh_size_x, step=mesh_size_x)
Ynew = np.arange(y_min, y_max+mesh_size_y, step=mesh_size_y)
xnew, ynew = np.meshgrid(Xnew, Ynew)
```

Listing 3.4: Evaluation grid setup

The interpolating function is then constructed using the `RectBivariateSpline` and `RBFInterpolator` routines from the package `scipy.interpolate` respectively. At this stage, the invocation of the two functions, visualized by the dark grey node in Fig. 3.1a, differs slightly.

- `scipy.interpolate.RectBivariateSpline` requires the unique x and y values of the original grid in strictly ascending order as 1-D arrays' of size n_x and n_y, respectively, to set up the support knots for the interpolating function and the function values are to be supplied as a 2D array of function values at those grid points of size (n_x, n_y), where n_x, n_y is the number of gridpoints in each dimension

- `RBFInterpolator`, on the other hand, requires the grid points to be passed as a 2D array with dimensions $(n_x \cdot n_y, 2)$. Therefore, the function values must also be supplied in the same manner as a 1-D array of size $(n_x \cdot n_y, 1)$.

For the spline interpolation, the order of the underlying spline space can be chosen at this point, and for the RBF-Interpolator, the underlying RBFs $\phi(\mathbf{x})$. The interpolating function is then stored in the variable `interpf` to avoid unnecessary

recalculation of the interpolation function when evaluation at different intervals is necessary.

For the evaluation, the interpolating function `interpf` is called with the evaluation mesh grids `xnew` and `ynew` as arguments for the spline routine `RBS` and with `new_grid` for the RBF routine `RBF`, as the interpolating function again requires different invocations. This process step is visualized as the last grey node in Fig. 3.1a. The function call and evaluation for both the spline and RBF methods are shown in Listing 3.5 and 3.6, respectively.

```
1     interpf = RBS(x, y, z, kx=order, ky=order)
2     interp_data = interpf(xnew, ynew, grid=False)
```

Listing 3.5: Interpolation and Evaluation of the spline method

```
1     interpf = RBF(XY, Z, kernel=order, epsilon=0.5)
2     interp_data = interpf(new_grid)
```

Listing 3.6: Interpolation and Evaluation of the RBF method

Plotting and exporting

The third part of the script is concerned with plotting and exporting the interpolated quantity. A new filename for the output is constructed using the working path given in `sys.argv[2]`, the filename of the input data given in `sys.argv[1]`, and the mesh size given in `sys.argv[3]`. The interpolated data is then saved to the new file using `numpy.savetxt` as shown in Listing 3.7.

```
1 file = sys.argv[1].split("/")[-1].split(".")[0]
2 save_file = working_path + file +
3             "_intp_RBF_{}_{}.csv".format(order, mesh_size_x)
4 np.savetxt(save_file,
5            np.concatenate((xnew.reshape(xnew.size, 1),
6                            ynew.reshape(xnew.size, 1),
7            interp_data.reshape(xnew.size, 1)), axis=1),
8            fmt = '%1.4e', delimiter=' ', newline='\n',
9            header='x y z', comments='')
```

Listing 3.7: Data export to file system

Finally, the interpolated quantity is plotted using `matplotlib.pyplot` and saved to the working path using `matplotlib.pyplot.savefig` as shown in Listing 3.8. As this process requires outside communication with the filesystem, it is visualized as the bottom yellow node in Fig. 3.1a. This is done via the use of a boolean variable `plot`, which is set to `False` by default. It would allow the use of another command line argument if deemed necessary. Since the feature of plotting the input and output data is, however, only of use for debugging reasons and is not needed for the interpolation itself in a production environment, the feature was not implemented.

```
1  if plot:
2      save_plot_output = working_path + file +
3          "_output_RBF_{}_{}.png".format(order, mesh_size_x)
4      fig = plt.figure(figsize = [12,12])
5      ax = plt.axes(projection='3d')
6      ax.plot_surface(xnew, ynew, interp_data, cmap='viridis')
7      ax.set_xlabel('x')
8      ax.set_ylabel('y')
9      ax.set_zlabel('z')
10     fig.savefig(save_plot_output)
```

Listing 3.8: Visualization of data

3.2.2 C - GSL

For the interpolation using *C*, the *GSL* subroutine `gsl/gsl_interp2d.h` and `gsl/gsl_spline2d.h` from the *GSL* library was used. The routine uses an instance of bicubic or bilinear splines (`gsl_interp2d_bicubic`, `gsl_interp2d_bilinear`) together with the number of support knots in each dimension to create an interpolation object consisting of an underlying spline supported on these knots.

The interpolation routine was implemented as a standalone *C* script `interpolation.c` and, further, as a feature to be used within the simulator. Since both implementations are identical in their functionality, only the standalone script will be discussed in the following. Figure 3.1b shows the program flow for the standalone implementation in *C*, with a short note on the simulator implementation at the end of

the chapter. The script is divided into three parts:

1. The first part is concerned with the import of the necessary modules.
2. The second part consists of the different functions used in the main routine.
3. The main routine itself.

Import of modules

The first part of the script is concerned with the import of the necessary modules, as shown in Listing 3.9. The modules imported are:

- `stdio.h`: for the handling of command line arguments
- `stdlib.h`: for the handling of arrays
- `gsl/math.h`: for the interpolation routine
- `gsl/gsl_interp2d.h`: for the interpolation routine
- `gsl/gsl_spline2d.h`: for the interpolation routine

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <gsl/gsl_math.h>
4 #include <gsl/gsl_interp2d.h>
5 #include <gsl/gsl_spline2d.h>
```

Listing 3.9: Inclusion of necessary headers

`stdio.h` provides the necessary functionality to handle the input provided when invoking the program. `stlib.h` provides the required functionalities for dynamic memory allocation. Specifically, the function `malloc` and the data type `size_t`. `gsl/math.h` is needed to calculate the new grid structure. `gsl/gsl_interp2d.h` and `gsl/gsl_spline2d.h` are the header files from the *GSL* library that are needed to perform the interpolation task. `gsl/gsl_spline.h` provides the basis spline object that is used within `gsl/gsl_interp2d.h` to set up and calculate the interpolating spline function.

Functions

The second part of the program is concerned with the different functions used for the main routine. The functions are:

- `read_csv(x_array, y_array, z_array, filename, scan, size)`
- `countDistinct(array, N)`
- `configure_spline(x_array, y_array_y, z_array, n_x, n_y, spline)`

Since within *C* for dynamic memory allocation, defined as a procedure in which the size of a data structure is changed during the runtime, necessitates a call to either `malloc` or `calloc`, the function `read_csv`, shown in Listing 3.10, is called several times. The function is called with pointers to the containers for the x, y and z values (`xp`, `yp`, `zp`), the `filename` as provided via command line input, an integer `scan` used to discern whether to read or to scan the file, as well as an integer pointer `size` for use if the function is called to scan a file.

The initial call to the function scans the file for the number of lines and stores it in the variable `size`. This is done to allocate the memory for the arrays `x`, `y`, and `z`, which is done in the main routine and accomplished via the boolean variable `scan` that either initiates a scan of the file and writes the resulting number of lines to `size` or reads the content and writes it to the appropriate container. After reallocating the memory associated with the pointers `x`, `y`, and `z`, the `read_csv` function is called again to read the data from the file and store it in the respective arrays.

```
1  void read_csv(double* xp, double* yp, double* zp,
2                char* filename, int scan, int* size)
3  {
4      float nodePositionX, nodePositionY, quantity;
5      FILE* fp = fopen(filename, "r");
6      int line = 0;
7
8      if (!fp) printf("Can't open file %s\n", filename);
9
10     int retval = fscanf( fp, "%*[^\n]\n" );
```

```
11      if(retval == -1) printf("[ERROR]\n" );
12
13      if(scan){
14        while ( fscanf(fp, "%f %f %f\n", &nodePositionX,
15                       &nodePositionY, &quantity) != EOF )
16          {line++;}
17          *size = line;
18      }
19      else{
20        while ( fscanf(fp, "%f %f %f\n", &nodePositionX,
21                       &nodePositionY, &quantity) != EOF )
22        {
23          xp[line] = nodePositionX;
24          yp[line] = nodePositionY;
25          zp[line] = quantity;
26          line++;
27        }
28        fclose(fp);
29      }
30  }
```

Listing 3.10: Function to read or scan CSV from file

Since the number of distinct values in the arrays `x` and `y`, which are the number of support knots in each direction, cannot be known in advance, it is necessary to calculate them separately. The function `countDistinct` (Listing 3.11) is used to count the number of distinct values x and y in the arrays `x` and `y` and return the respective number of unique entries for later use in the main routine. The routine is called with the array `arr` and the size of the array `n` to be inspected.

```
1 size_t countDistinct(double* arr, size_t n)
2 {
3     int res = 1;
4     for (int i = 1; i < n; i++) {
5         int j = 0;
6         for (j = 0; j < i; j++)
```

```
7               if (arr[i] == arr[j])
8                   break;
9           if (i == j)
10              res++;
11      }
12      return res;
13  }
```

Listing 3.11: Function to count number of distinct values in an array

The function `configure_spline` (Listing 3.12) configures the spline object used for the interpolation routine. To this end, the function takes the arrays `x`, `y`, and `z` as well as the number of support knots in the x and y direction (`nx`, `ny`) and a spline object `spline` as arguments. With `gsl_spline2d_set`, the structure of the interpolating polynomial on the support knots is created, and each support knot is assigned a value from the array `z`. With `gsl_spline2d_init`, the spline object is then initialized to the given values of the support knots, i.e., their x and y values in the original grid.

```
1  void configure_spline( double* x, double* y, double* z,
2                         int nx, int ny, gsl_spline2d *spline )
3  {
4    size_t i, j;
5    double *z_smth = malloc(nx * ny * sizeof(double));
6    double *xvals = malloc(nx * sizeof(double));
7    double *yvals = malloc(ny * sizeof(double));
8
9    for (i = 0; i < nx; i++) {
10     for (j = 0; j < ny; j++) {
11       gsl_spline2d_set(spline, z_smth, i, j, z[j*nx + i]);
12     }
13   }
14
15   j = 0;
16   for (i = 0; i < nx*ny; i++) {
17     if (i%ny == 0)
```

```
18      {
19        xvals[j] = x[i];
20        j++;
21      }
22    }
23    for (i = 0; i < ny; i++)
24    {
25      yvals[i] = y[i];
26    }
27    gsl_spline2d_init(spline, xvals, yvals, z, nx, ny);
28  }
```

Listing 3.12: Function to configure spline object

Main routine

The program requires the following arguments to be supplied when invoking it. These are the quantities to be interpolated, supplied as a *CSV* file with three columns, where the first two columns describe the grid point and the third column the value of the quantity at each grid point. The first two columns, meaning the x and y values of the grid, are required to be in strictly ascending order. This is necessary because the *GSL* routines for the interpolation require the provided grid points to be in strictly ascending order. Therefore, this constraint on the input is imposed to avoid a costly sorting algorithm within the implementation. The second user input required is the new mesh size to generate the evaluation grid on which the interpolating function shall be evaluated. The third user input required is the path to which the interpolated quantity is written.

Via the variables `inputFile`, `working_path`, and `mesh_size_x/y`, the user input when calling the program is saved. The main routine then begins to initialize all necessary objects and allocate all necessary memory (Listing 3.13). The required external input to the program is given by the grey trapezoidal node in Fig. 3.1b, while the initialization is represented by the yellow node.

```
1   const gsl_interp2d_type *T = gsl_interp2d_bilinear/
     gsl_interp2d_bicubic;
2   char* inputFile = argv[1];
```

```
3   char* working_path = argv[2];
4   double mesh_size_x = atof(argv[3]);
5   double mesh_size_y = atof(argv[3]);
6
7   int Nx, Ny;
8   int i, j;
9   int nx, ny;
10
11  const int PATH_LENGTH = 256;
12
13  int scan = 1;
14  int* size = malloc(sizeof(int));
15
16  double range_x;
17  double range_y;
18
19  FILE   *filePntr;
20  char   filenameINTP[PATH_LENGTH];
21
22  double* xa = malloc(sizeof(double));
23  double* ya = malloc(sizeof(double));
24  double* za = malloc(sizeof(double));
```

Listing 3.13: Initialization from user provided arguments and declaration of necessary variables

The arrays `xa` and `ya` will contain the x and y values of the original grid, and the array `za` will contain the value of the externally provided quantity to be interpolated at those points. As mentioned previously, via the use of the boolean variable `scan`, the function `read_csv` is called twice, with the appropriate memory allocation in between. This is represented in Fig. 3.1b by the first two grey nodes after the yellow initialization node. The data is then read from the file and stored in the arrays `xa`, `ya`, and `za`, which are in a further scan used to count the number of distinct values in the arrays `xa` and `ya` as shown in Listing 3.14.

```
read_csv(xa, ya, za, inputFile, scan, size);
xa = realloc(xa, *size * sizeof(double));
ya = realloc(ya, *size * sizeof(double));
za = realloc(za, *size * sizeof(double));

if (xa == NULL || ya == NULL || za == NULL)
  { printf("Error reallocating memory\n"); }

scan = 0;
read_csv(xa, ya, za, inputFile, scan, size);
```

Listing 3.14: Reallocation of data array memory as ascertained by `read_csv`

The number of distinct values in the arrays `xa` and `ya` is then stored in the variables `nx` and `ny`. This is done with the previously mentioned function `countDistinct` and is represented in Fig. 3.1b as a grey node. This step allows us to extract our grid from the input data. With this information, the spline object can now be set up (Listing 3.15). A `gsl_spline2d` instance `spline` is declared using the `gsl_spline2d_alloc` function. It is called with the desired type of spline T and the number of support knots in the x and y directions. Further, two accelerator objects are created using the `gsl_interp_accel_alloc` function.

```
nx = countDistinct(xa, *size); /* x grid points */
ny = countDistinct(ya, *size); /* y grid points */

gsl_spline2d *spline = gsl_spline2d_alloc(T, nx, ny);
gsl_interp_accel *xacc = gsl_interp_accel_alloc();
gsl_interp_accel *yacc = gsl_interp_accel_alloc();

configure_spline(xa, ya, za, nx, ny, spline);
```

Listing 3.15: Determination of grid structure and initialization of spline and accelerator objects

The method with which the spline object is initialized is either `gsl_interp2d_bilinear` for piecewise linear interpolation or `gsl_interp2d_bicubic` for the classic cubic spline interpolation method. The spline object is configured using the `configure_spline`

function as shown in Listing 3.15, line 8. It is called with the containers containing the grid data (`x`, `y`) and the value at each grid point (`z`), as well as with the previously calculated number of support knots in each direction (`nx`, `ny`) and the created spline instance `spline`. Again, this represents the grey node in Fig. 3.1b directly above the dark grey interpolation node.

The number of grid points in each direction for the evaluation grid, `Nx`, `Ny`, is calculated using the mesh size provided via user input through the variables `mesh_size_x`, `mesh_size_y` as shown in Listing 3.16.

```
1  range_x = xa[*size-1] - xa[0];
2  range_y = ya[*size-1] - ya[0];
3  Nx = floor(range_x / mesh_size_x) + 1;
4  Ny = floor(range_y / mesh_size_y) + 1;
```

Listing 3.16: Calculation of evaluation grid dimensions

The interpolated quantity is evaluated on the evaluation grid using `gsl_spline2d_eval`. The evaluation points are constructed in a nested loop using the previously calculated dimensions `Nx`, `Ny` as shown in Listing 3.17. The function is called with the spline object, the x and y values of the evaluation grid, and the accelerator objects as arguments. In Fig. 3.1b, this is represented by the dark grey node above the bottom yellow node where the interpolated data is written to a file using `fprintf`.

```
1  for (i = 0; i < Nx; ++i)
2  {
3    double yj = ya[0] + (ya[nx-1] - ya[0]) * i /  (Ny-1);
4    for (j = 0; j < Ny; ++j)
5    {
6      double xi = xa[0] + (xa[*size-1] - xa[0]) * j / (Nx-1);
7      double zij = gsl_spline2d_eval(spline, yj, xi, xacc, yacc
       );
8      fprintf(filePntr, "%e %e %e\n", xi, yj,  zij);
9    }
10   fprintf(filePntr, "\n");
11 }
```

Listing 3.17: Evaluation of interpolating spline

Integration into ViennaWD

To incorporate the functionality into ViennaWD, the decision was made to include a variable in the input *LUA* file, `interpolate`, to discern whether the interpolation routine shall be called or not. The *LUA* file is used to communicate all relevant simulation parameters, e.g., mesh size, size of the simulation domain, boundary conditions, etc., to the simulator and is passed as a command line argument when the simulation program is invoked. The additional variable internally is added as an additional integer member to the *C* struct `geometry_t`, which is a composite data type that defines a physically grouped list of variables under one name in a block of memory. Specifically, in this case, the struct `geometry_t` groups the necessary variables to describe the device geometry as shown in Listing 3.18.

```
1  {    char    is2D;
2       char    bcType[4];
3       unsigned short int Kmax[3];
4       unsigned short int nNodes[2];
5       unsigned short int nNodes_global[2];
6       unsigned short int domain_overlap;
7       unsigned short int kDimensions;
8       unsigned short int annihilationSpatialScaling;
9       double meshSize;
10      double cellVol;
11      double deviceWidth;
12      double Lx[2];
13      double Ly[2];
14      double Lx_global[2];
15      double Ly_global[2];
16      double Lcoh;
17      double Lcutoff;
18      double delK;
19      double hbarDelkM;
20      unsigned short int eff_mass_profile;
21      int interp;} geometry_t;
```

Listing 3.18: Geometry struct used in ViennaWD

Depending on this variable, the interpolation is triggered. The interpolation result is then again written to the filesystem so that the interpolated quantity is available for inspection after the simulation has been completed. In this context, the main routine from the standalone implementation has been substituted by the function `interpolate` shown in Listing 3.19. `interpolate` has the same functionality as the standalone implementation with the important distinction that it is not provided with command line input but rather with the following parameters. `inputFile`, which is the variable that has the quantities filename stored, `working_path` for writing the interpolated quantity to the filesystem, and `geometry`, which is the struct mentioned before, that holds the information about the required mesh size as a member `meshSize`.

```
1 void interpolate( const char *inputFile,
2                   const char *working_path,
3                   const geometry_t *geometry )
```

Listing 3.19: Interpolation function called in the setup process

The routine is called in the setup process of the simulator. The distributor process is responsible for setting up all the necessary data structures before populating them with the input data and then broadcasting them to the receiver processes. Therefore, when the distributor process initially initializes the electric potential, the routine is triggered depending on the aforementioned newly introduced variable `interpolate` before the program continues as normal to read the now interpolated quantity from the file system.

4 Effective Mass

Serving as a fundamental concept in condensed matter physics, semiconductor physics, and materials science, the effective mass encapsulates the behavior of charge carriers within a material [62, 63]. Its meticulous consideration within simulations facilitates a deeper understanding of complex phenomena, enabling researchers and engineers to make informed decisions and predictions regarding the performance and behavior of materials, devices, and systems. Modern nanoelectronic devices, such as gate-all-around field effect transistors (GAAFETs), consist of many different materials. This is because different properties are needed for each specific part of such devices. With semiconducting 2D materials currently in contention to potentially outperform silicon [64] based field-effect transistors (FETs), with dimensions scaled down to a few atomic layers, these new nanoelectronic devices have to operate at high electric fields and therefore require suitable insulators. At such minuscule scales, defects are inevitable at the interfaces between the materials. However, such interfacial defects substantially increase the leakage currents through the gate insulators and thus degrade the desired performance gain of such designs [6]. Figure 4.1 shows the effects of interfacial defects as scattering centers and how they can severely degrade the mobility and substantially increase the leakage currents through hexagonal boron nitride (hBN) insulator layers. To study these and other effects, a modern simulator must be able to simulate the transport of charge carriers through different types of materials. The mobility of charge carriers, electrons, and holes in these different materials is characterized by their effective mass [65]. In many semiconductors (Ge, Si, $GaAs$, ...), the band structure for ideal lattices, without crystal defects or impurities, can be locally expressed as

$$E(\mathbf{k}) = E_0 + \frac{\partial^2 E}{\partial \mathbf{k}^2} \tag{4.1}$$

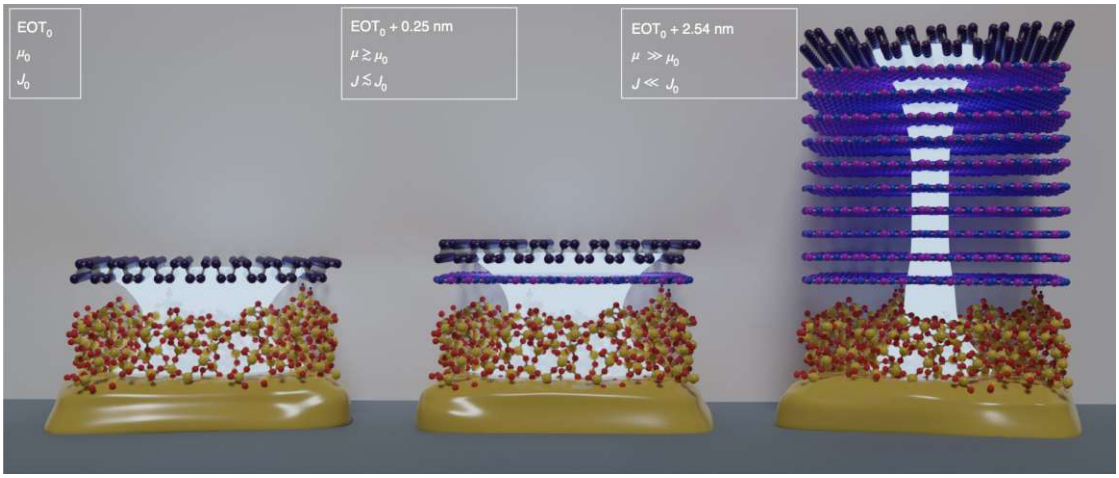


Figure 4.1: Interface of black phosphorus (P, purple) with SiO2 (Si, yellow; O, red), illustrating the effect of hexagonal boron nitride (hbN) interlayers to sufficiently suppress Coulomb scattering and remote phonon scattering in the underlying oxide to assure high mobilities. [6]. Reprinted with permission from Knobloch et al., Nature Electronics 4.2 (2021), pp. 98–108, Copyright 2021 Springer Nature.

Thus, in these simple cases, the effective mass is given by the curvature of the band structure at the local extremum [65]:

$$m^* = \hbar^2 \frac{1}{\frac{\partial^2 E}{\partial \mathbf{k}^2}} \tag{4.2}$$

The description of charge carriers with their respective effective masses m^* allows them to be treated with the Wigner equation, introduced in Eq. (2.13), describing ballistic carrier transport. In this simple assumption, the effective mass is taken to be isotropic over each material. This allows for the implementation of the effective mass into the simulator on a grid-based approach.

In Fig. 4.2, again, a flowchart for ViennaWD is presented where dark grey nodes highlight the process steps in which adaptations have been made to incorporate the effective mass into the simulator.

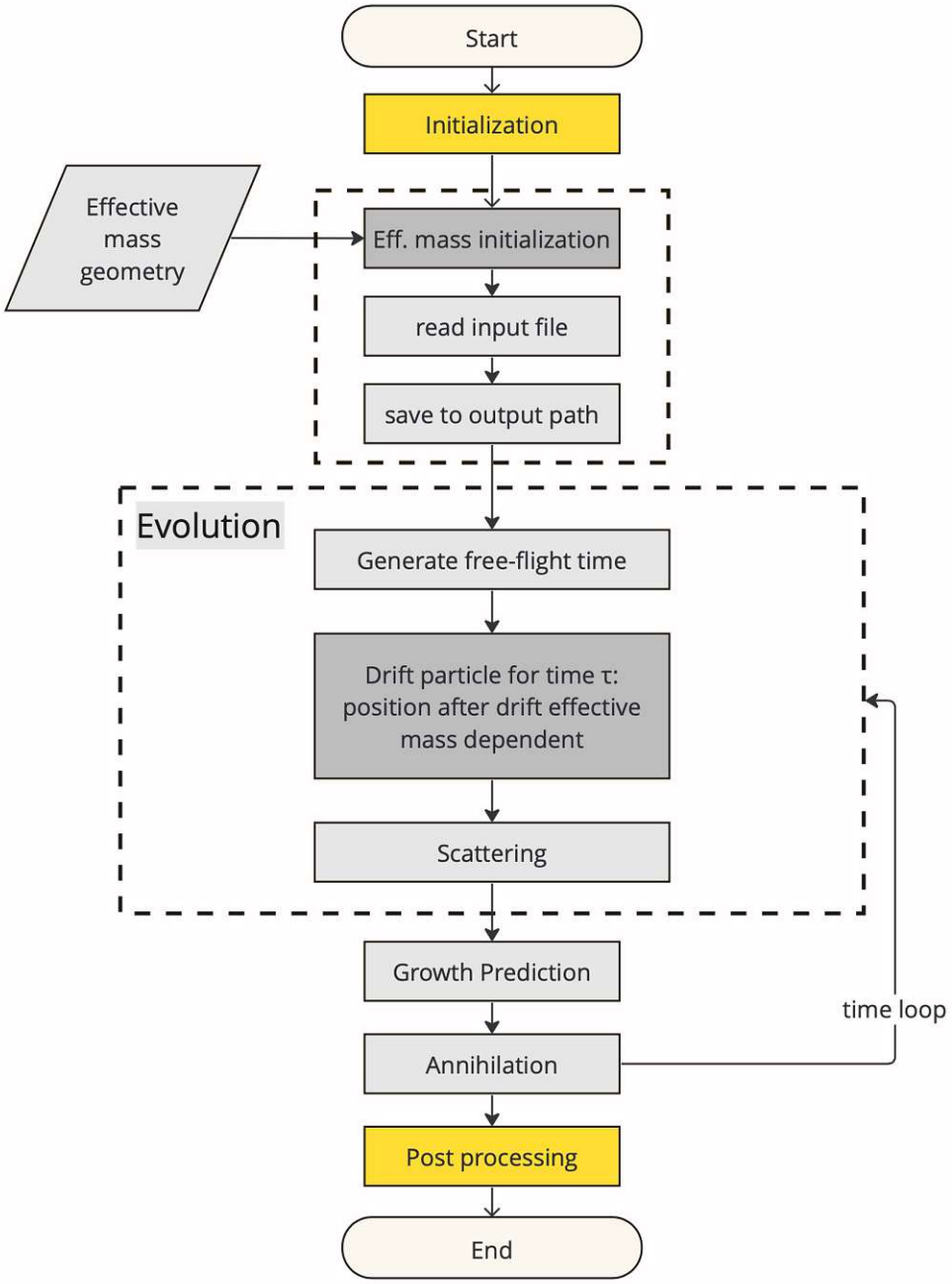


Figure 4.2: Flowchart describing the additions and effects of the effective mass functionality into Vienna WD.

4.1 Implementation

The following data structures were amended to support the functionality in the simulator. The struct `physical_quantities` combines quantities defined at each mesh node, such as, for example, the potential, the effective mass, but also the density and the current. `physical_quantities` were augmented by the member `effective_mass` to provide the appropriate storage container. The struct `geometry` was augmented with an identifier `eff_mass_profile` to either trigger the effective mass routine or set it to default at each grid point. This is visualized in Fig. 4.2 in the first dark grey node at the top of the flowchart. The simulator had to be extended with routines that were implemented in a standalone module `eff_mass.c`, that introduces the following functions:

- Initialization function: `iiiEffMassProfile`
- Extracting function: `readEffMass`
- Reset function: `resetEffMass`
- Global allocation function: `globalEffMassAlloc`

Initialization function (`iiiEffMassProfile`) is called by the distributor process of the main routine, and the function header is shown in Listing 4.1. It takes as parameters the structures `physical_quantities`, which are needed to later store the information from the input file in the according container, `geometry` with the information whether an effective mass profile was loaded or not and for further use in the save function. Further parameters are the `inputFile` that holds the information where the effective mass will be read from the file system and working path, again needed for the save function.

```
1 void iiiEffMassProfile( phys_quant_t *phys_quants,
2                         geometry_t *geometry,
3                         const char *inputFile,
4                         const char *working_path )
```

Listing 4.1: Initialization function for the point-based effective mass

The above-mentioned save function for the effective mass (`saveEffMass`) functionality is added to the respective module `save_funcs.c` and handles the output, in this case of the effective mass, to the filesystem.

Extracting function (`readEffMass`) reads the effective mass from `inputFile`. The function first goes on to check whether the file can be opened and then further if the file contains information. If both are true, the function then proceeds to read line by line from `inputFile`, and the data is saved to the member `effective_mass` of the struct `physical_quantities`. The function header is shown in Listing 4.2.

```
1 void readEffMass( const char *inputFile,
2                   const geometry_t * geometry,
3                   phys_quant_t *phys_quants )
```

Listing 4.2: Extraction function used to read the data from file into the appropriate member in struct `geometry`

Reset function (`resetEffMass`) resets the effective mass to the default value. The function header is shown in Listing 4.3.

```
1 void resetEffMass( const geometry_t * geometry,
2                    phys_quant_t *phys_quants )
```

Listing 4.3: Reset function

Global allocation function (`globalEffMassAlloc`) is a function that allocates memory for the global effective mass necessary when running the program in a parallel fashion to distribute the user input to the receiver processes, and the function header is shown in Listing 4.4.

```
1 void globalEffMassAlloc( double ***global_effMass,
2                          unsigned short int data_rows,
3                          unsigned short int data_cols,
4                          char destroy )
```

Listing 4.4: Global allocation for distributed memory calculations

In addition to the considerations within the simulator itself, several peripheral tools were introduced and altered. To construct the geometries of different material structures characterized by their effective mass, a *Python* notebook was developed that allows the user to easily and rapidly prototype new geometries and directly visualize them within the notebook, thus providing an easy-to-use tool in the setup of new simulations. Further, the post-processing step was augmented to support the newly introduced functionality.

4.2 Implications

As already introduced in Section 2.1.1, the effective mass enters the Wigner equation as presented earlier in Eq. (2.7). Within ViennaWD, this affects the propagation of the particles that sample the initial wavefunction and is shown in Fig. 4.2 in the second highlighted node and in Listing 4.5. At each time step, the particle drifts for duration τ before either the particle is scattered or the end of the time step is reached. In this drift phase, the new particle position is calculated depending on the particle momentum at the start of that timestep.

```
1    particle_position_x = C * particle_momentum_x /
2                          effMass_factor[i,j] * tau
3    particle_position_y = C * particle_momentum_y /
4                          effMass_factor[i,j] * tau
```

Listing 4.5: Calculation of new particle postition depending on local effective mass

Here, `effMass_factor` is the effective mass at the current grid point in terms of m_c, the charge carrier rest mass, `particle_momentum` the particle momentum in x and y direction respectively for the particle, and C a constant containing the necessary unit conversions. Here, the effective mass is a multiplier that enters the equation inversely. In the above equation shown in listing 4.5, a higher effective mass effectively reduces the momentum of the particle, and therefore, the position that the particle arrives at after the drift phase is nearer to its origin than for a particle that at its grid point observes a lower effective mass.

5 Evaluation

First, two very different test functions are introduced, which are representative of arbitrary quantities that might be encountered in a simulation workflow with ViennaWD. Such quantities are, for example, the electric potential that was recorded externally as an example of a relatively smooth and slowly varying quantity. A smooth test function will be introduced to validate the interpolation routines investigated on such data. However, noncontinuous quantities such as modeled electric potentials, effective mass profiles, and descriptions of geometries might also be encountered. A simple step function to evaluate the implemented interpolation routines on such data will be introduced as well. Further, a measure for the interpolation error is introduced, and using the introduced measure, it is discussed visually and quantitatively whether the interpolation methods investigated hold up to the different challenges faced in a representative workflow.

Second, ViennaWD simulations of a single minimum uncertainty wave packet traversing different materials using the implemented position-dependent effective mass routines will be shown. Different geometries will be investigated to validate the implementation and explain the effect a spatially varying effective mass has on such a wave packet.

5.1 Interpolation

Two very different sets of data points were chosen to discern the applicability of the interpolation to data that might be encountered within a typical TCAD simulation workflow. A step function consisting of two distinct plateaus such as might be encountered when interpolating material structures such as an effective mass profile. Further, a smooth function consisting of several Gaussian peaks and

troughs was constructed to test the applicability of the interpolation routines on smooth data. Such data might be encountered when trying to fit an experimentally measured electric potential onto the grid necessary to verify physical quantities with ViennaWD. For both choices of the test function, the implementations were tested for different amounts of grid refinement, particularly for 10, 4, 2, and 4/3 times denser grids than the original.

To introduce some metric to compare the different interpolation techniques against each other, the following measure is introduced:

$$\|A\|_1 = \frac{\sum_{i,j} |a_{i,j}|}{\|A\|} \tag{5.1}$$

Here $a_{i,j}$ are taken to be the pointwise differences between the analytical values for the test quantity at gridpoint i, j and $\|A\|$ is the size of A, meaning the number of grid points. This measure ensures that the calculated sum of pointwise errors is appropriately weighed with the number of data points, therefore allowing for a comparison of the interpolation approaches on different grid refinements.

5.1.1 Step

In Fig. 5.1 (a), (c), and (e) show the evaluation of the interpolating spline function on two times denser grids for the *Python* implementation, for (a) bilinear, (c) bicubic, (e) biquintic interpolation order respectively. (b), (d) shows the evaluation of the interpolating spline function using the *C-GSL* implementation for (b) bilinear and (d) bicubic order, respectively. (f) shows the step function that was used as input for the interpolation routines. Figure 5.2 shows the evaluation of the interpolating RBF function on a two times denser grid for (a) the linear order polynomial basis function and (b) the cubic order polynomial basis function. (c) for the quintic order polynomial basis function. (d) for the Gaussian basis function. All interpolations were performed for the step function as in Fig. 5.1 (f). In Fig. 5.1, oscillations in the evaluated interpolation functions for orders of the interpolating polynomial higher than $k = 1$ are observed. These oscillations stem from the fact that polynomials of higher order have the inherent condition that the higher derivatives of the interpolating polynomial have to be piecewise continuous. As seen in Fig. 5.1 (a), (b), the interpolation using first-order polynomials does

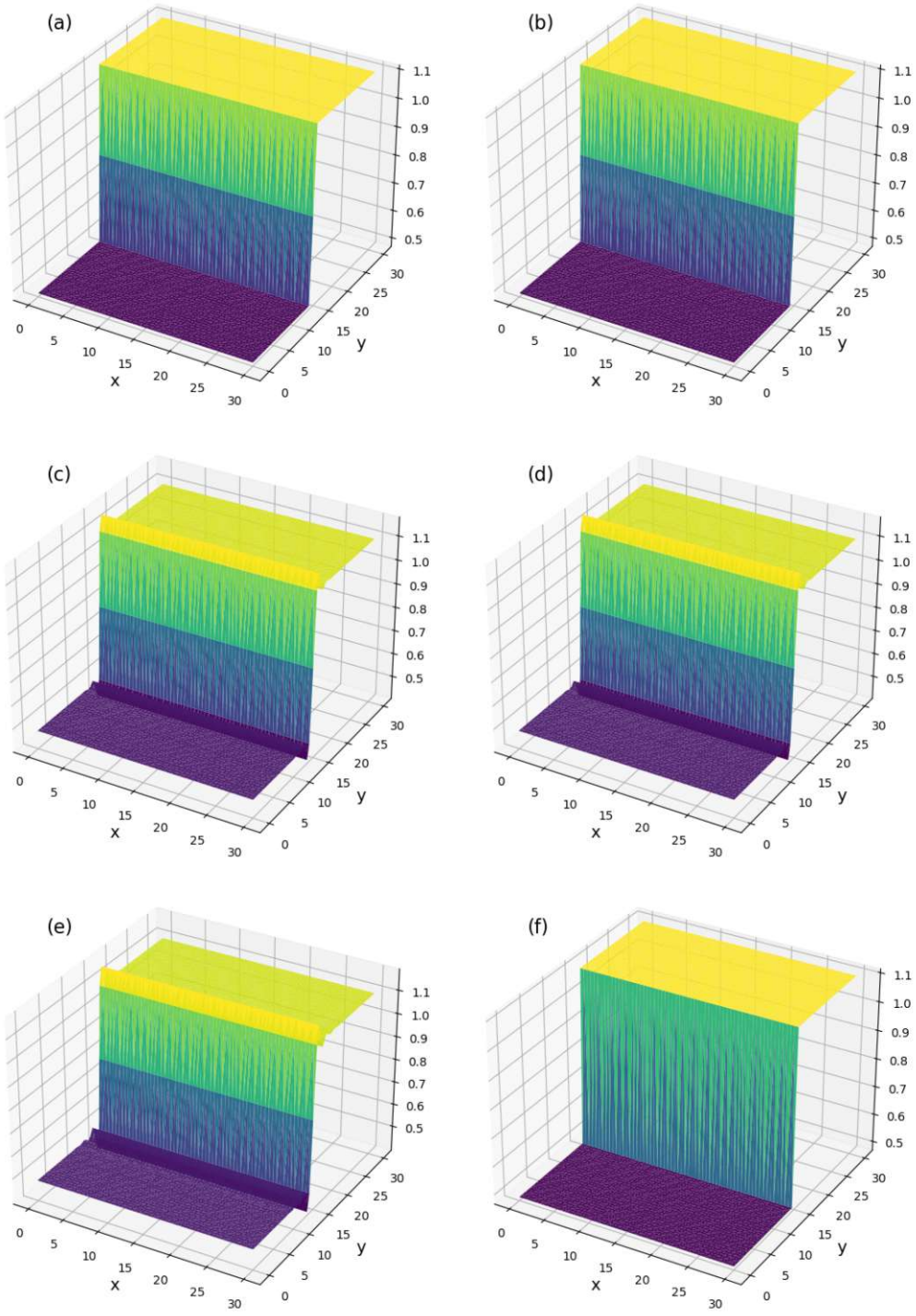


Figure 5.1: Interpolation results for *Python* and *GSL* routines for a step function.

not experience this behavior. These are both linear methods. Though no oscillations are observed for these very simple interpolation routines, it can be noticed

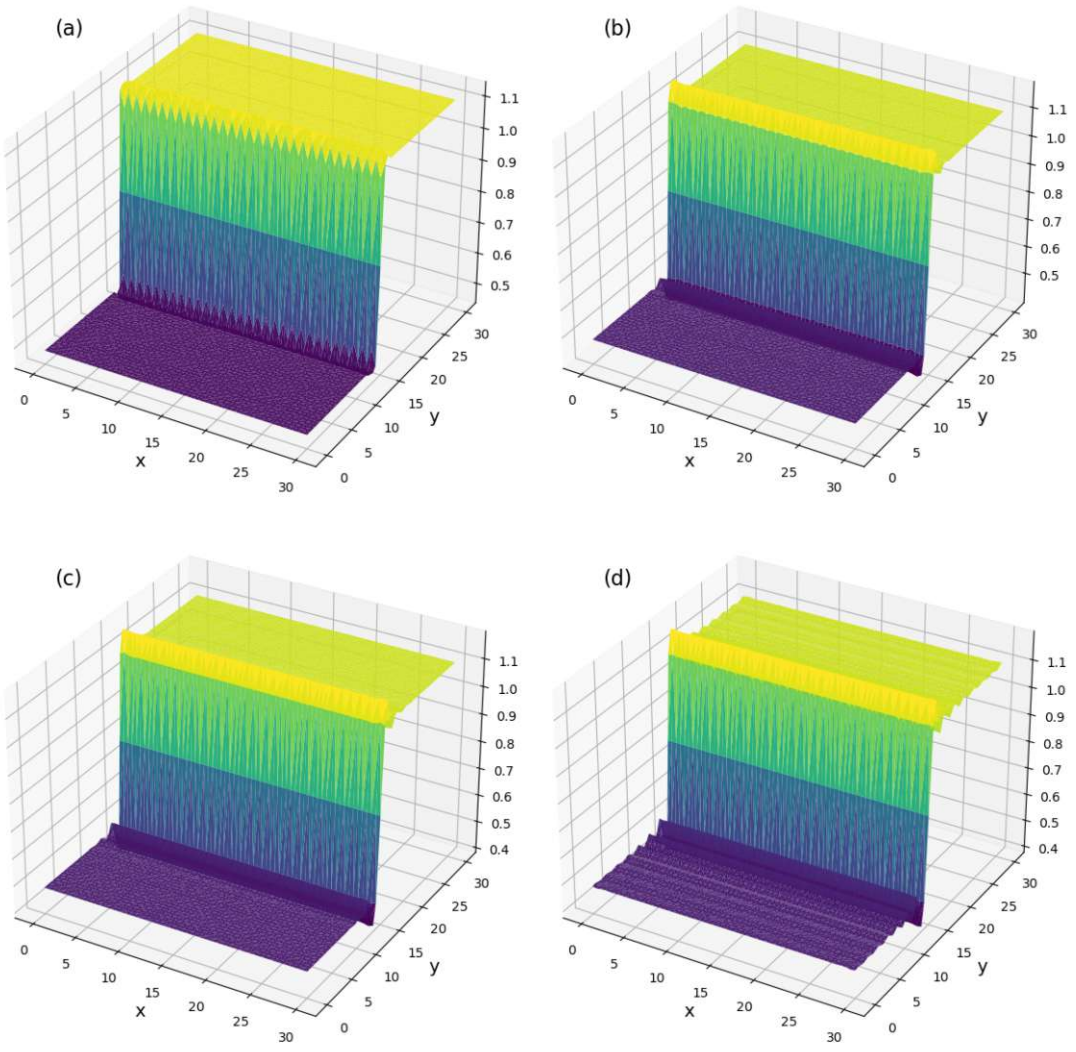


Figure 5.2: Interpolation results for the *Python* RBF implementation for a step function.

that between the grid points along the x direction closest to the step, instead of a vertical surface, a linear behavior between the two values of the plateaus of the quantity is introduced.

For the third-order polynomial interpolations, which are Fig. 5.1 (c) and Fig. 5.1 (d), minor oscillations of the interpolated quantity around the grid points closest to the step are visible. Compared to the aforementioned relatively slow rise in the

quantity for the linear interpolation methods, a steeper rise, reconstructing the original step better, is generated at the expense of introduced oscillations. A further increase in polynomial order for the interpolation shows even more oscillations present in the interpolated quantity without meaningful gains to the reconstruction of the step.

Though RBFs are said to limit these oscillations [46], they can still be observed in Fig. 5.2 for each of the four basis functions evaluated. The three RBF interpolations with polynomial kernel Fig. 5.2 (a-c), do all reconstruct the step rather successfully. However, oscillations are introduced not only perpendicular to the step function but also in the axis along the edge. This can be attributed to the use of RBFs, which are symmetric about the grid point on which they are located and are known to experience the "Runge phenomenon" [46]. These edge oscillations reduce visibly for higher order polynomial RBFs as seen in Fig. 5.2 (b) and Fig. 5.2 (c). However, the third and fifth-order polynomial RBFs produce noticeably more oscillations in the interpolated quantity perpendicular to the step.

Further, a Gaussian kernel was evaluated for the RBF implementation as introduced in Table 3.1. This was tested for a variety of different values of the constant c present in the Gaussian kernel, with a value of $c = 0.5$ producing some of the best results. Again, oscillations were reduced directly at the edge of the step; however, the oscillations perpendicular to the step increased noticeably. This phenomenon is, for example, tackled in [66], but will not be delved into here and will be discussed at the end of the chapter in Section 5.1.4.

5.1.2 Smooth

Figure 5.3 (a), (c), and (e) show the evaluation of the interpolating spline function on six times denser grids for the *Python* implementation, for (a) bilinear, (c) bicubic, (e) biquintic interpolation order respectively. (b), (d) shows the evaluation of the interpolating spline function using the *C-GSL* implementation for (b) bilinear and (d) bicubic order, respectively. (f) shows the smooth test function that was used as input for the interpolation routines. Figure 5.4 shows the evaluation of the interpolating RBF function on a six times denser grid for (a) the linear order polynomial basis function, (b) the cubic order polynomial basis function, (c) for the

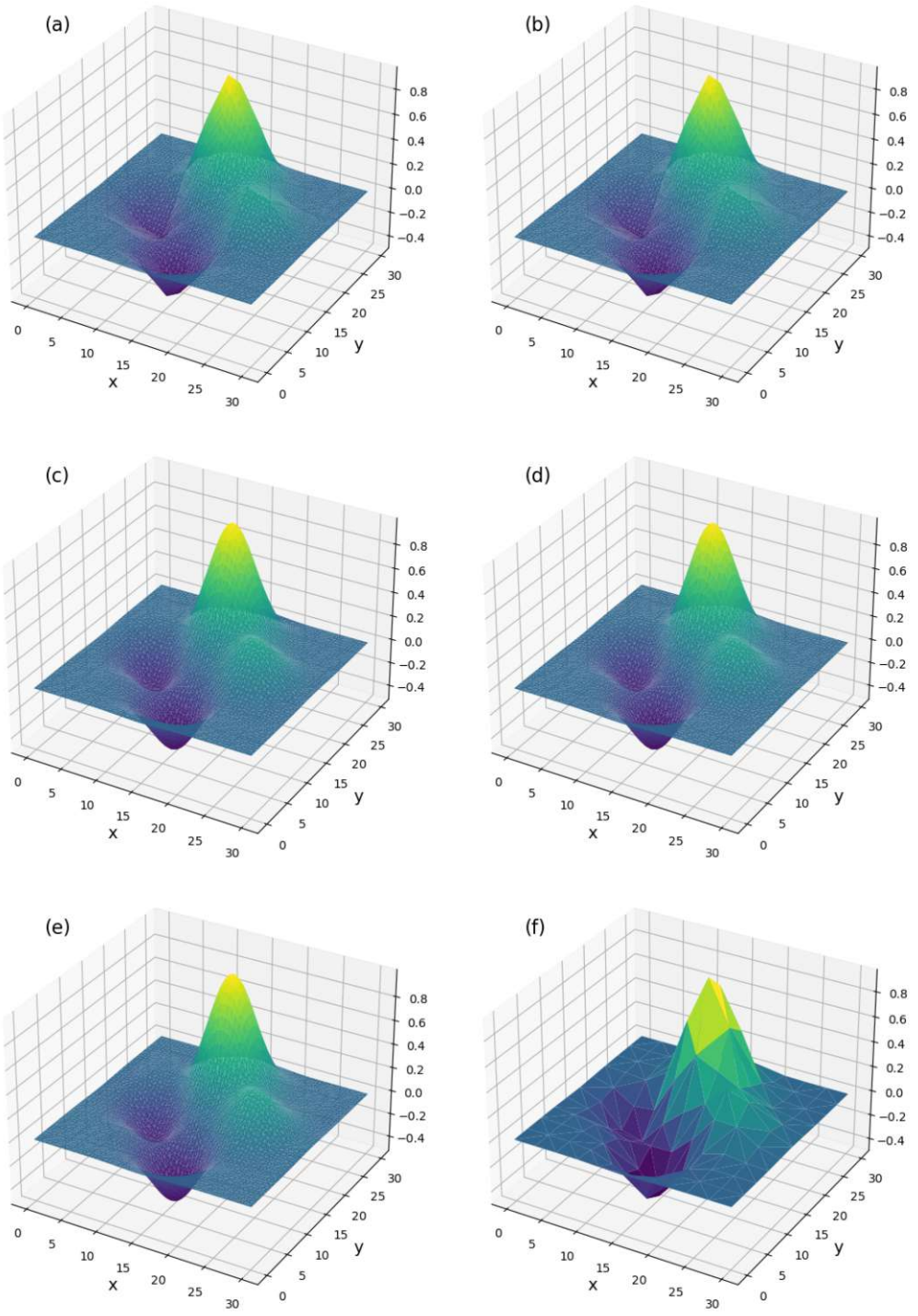


Figure 5.3: Interpolation results for *Python* and *GSL* routines for a smooth function.

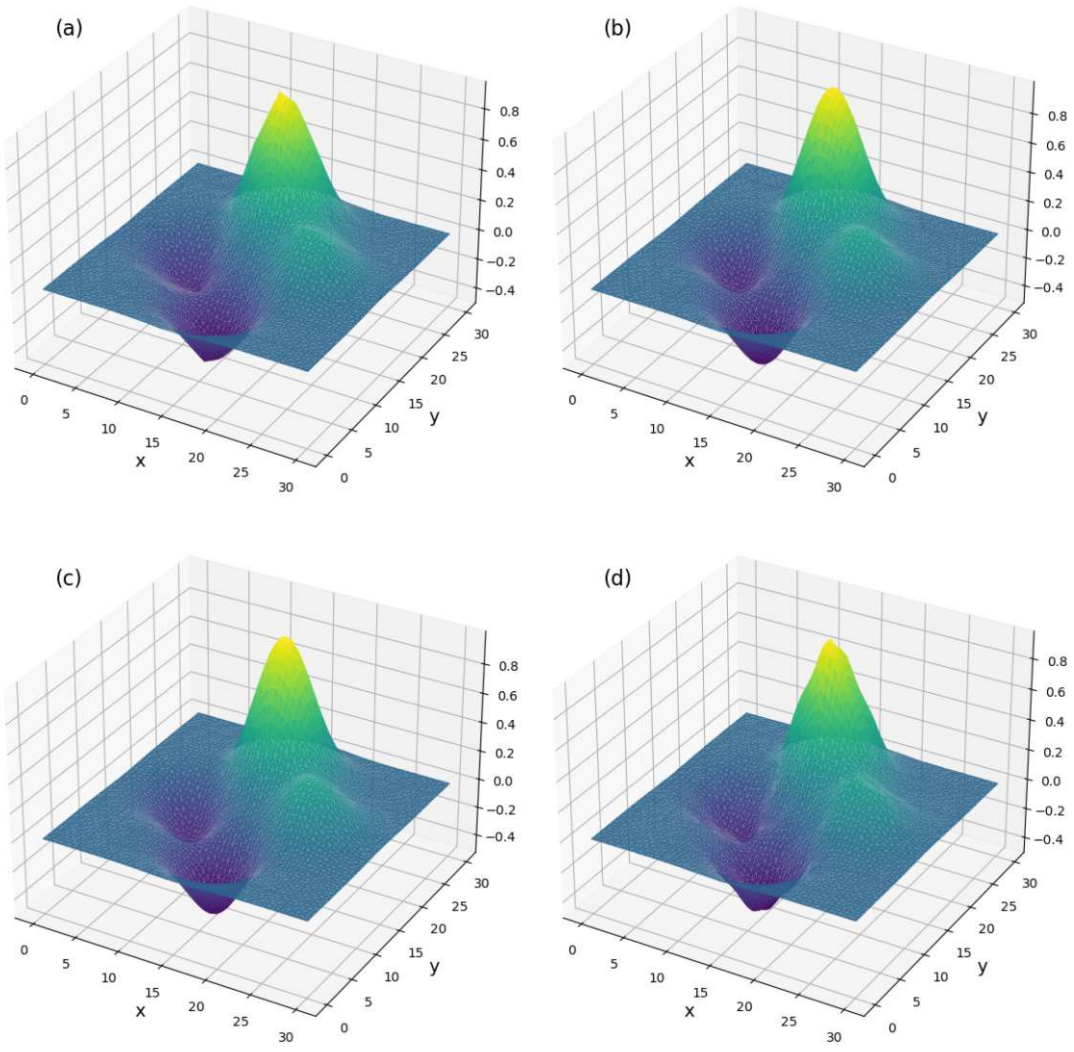


Figure 5.4: Interpolation results for the *Python* RBF implementation for a smooth function.

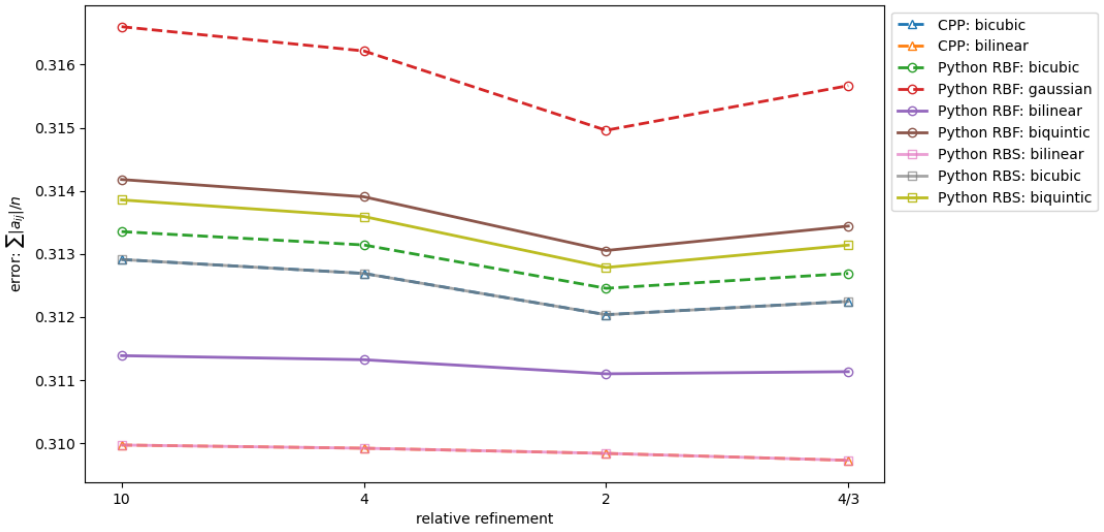
quintic order polynomial basis function, and (d) for the Gaussian basis function. All interpolations were performed for the smooth test function as in Fig. 5.3(f). The smooth test function was evaluated on two different sets of data points: once using the previously used 30 grid points per direction and again using only ten grid points per direction. Visualized in Fig. 5.3 and Fig. 5.4 is the interpolation on the sparser dataset, as the differences in interpolation methods are more visible.

In Fig. 5.3, all interpolation methods implemented visually perform the interpolation as expected. The previously observed oscillations do not arise for any of the spline interpolation methods, regardless of their underlying polynomial order. Further, there also aren't any oscillations visible for the implementation of the RBF interpolation Fig. 5.4. Therefore, both interpolation methods are viable for interpolating an arbitrary quantity that is sufficiently smooth on the domain.

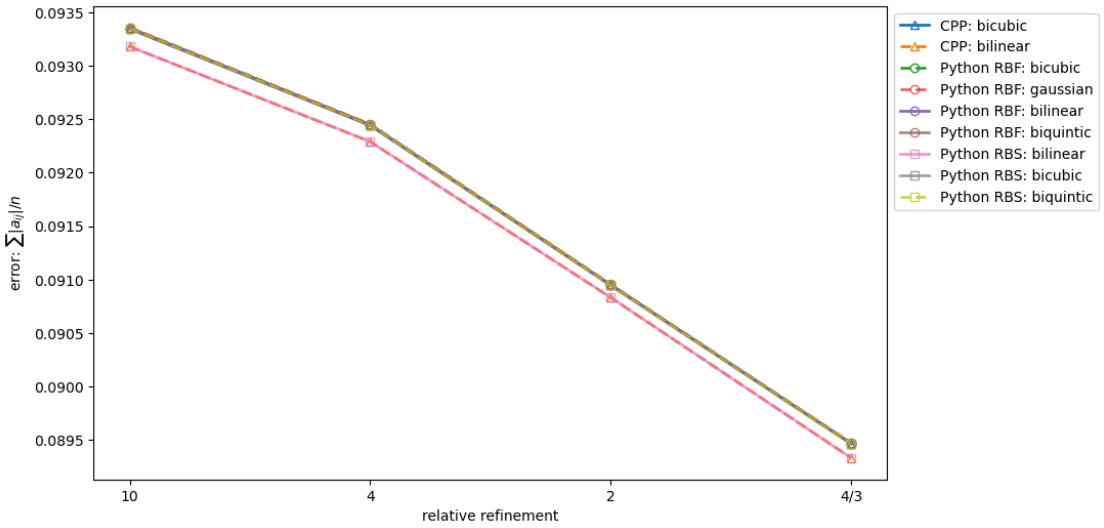
Though all methods now reproduce the test function without oscillations, there is still a visible difference in reconstruction quality. In both the spline and RBF reconstructions, the linear kernels Fig. 5.3 (a), (b) and Fig. 5.4 (a) interpolate the sampled conglomerate of Gaussian functions Fig. 5.3 (f) visibly rough. Since the test function is sampled sparsely, the extrema are not on top of the analytical extrema, and the linear kernels, as they do not contain information about the surrounding shape of the test function, are therefore not able to reconstruct the analytical extrema very well. In contrast to the linear kernels the cubic kernels Fig. 5.3 (c), (d), Fig. 5.4 (b) and the quintic kernels Fig. 5.3 (e), Fig. 5.4 (c) reconstruct the analytical result very well. In Fig. 5.4 (d), the Gaussian kernel again is not able to reconstruct the analytical function very well, as the method, in a sense, too accurately samples the test function and therefore reproduces the input very well without capturing the overall behavior of the data.

5.1.3 Error Analysis

In Fig. 5.5 for both the smooth and the step test function, the error measure introduced in Eq. (5.1) is shown for different mesh refinements. In Fig. 5.5a, the error measure is relatively large when compared to the error measures observed in Fig. 5.5b. For the discontinuous test function, also referred to as the step function, the different interpolation methods and the different polynomial orders for those methods vary significantly. The visual impression made above that the method of RBFs with a Gaussian kernel produces the most oscillations is represented here again, as it can be clearly seen that the method (Python RBF: gaussian) performs slightly worse in the error measure compared to the other kernels. The two methods using underlying quintic polynomials, namely the biquintile spline interpolation and the RBF interpolation with the quintic polynomial kernel that visually also



(a) Step function



(b) Smooth function

Figure 5.5: Error plots for both the step test function in (a) and the smooth test function in (b). The errors shown are calculated as explained in Eq. (5.1).

did not perform very well do also not measure well in their respective error (Python RBF/RBS: quintic). All three routines can be compared for the methods based on third-order polynomials, as this is also a supported method in the *GSL* library.

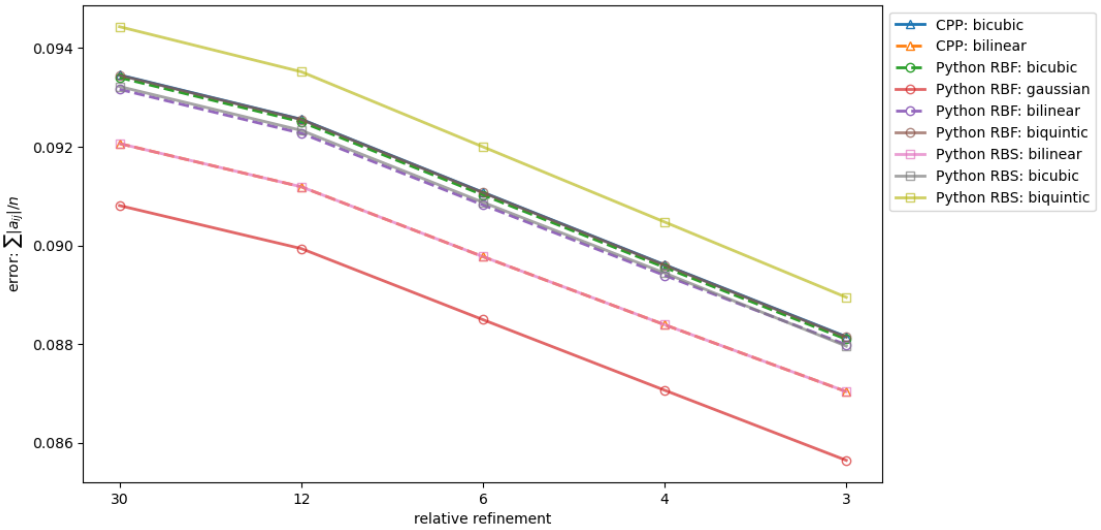


Figure 5.6: Error plot for the sparsely sampled smooth test function.

These methods (Python RBF/RBS: cubic and CPP: bicubic) in the error measure perform better, supporting the previous visual observation. The RBF with the underlying linear kernel is the best measuring method, and the two bilinear spline implementations for *C* and *Python* again perform identically.

For the smooth test function, the error measure for all interpolation methods is significantly lower than for the discontinuous step function. This supports the visual observation made previously. Further, again, the two linear spline interpolation methods in *C* and *Python* perform slightly better than the other interpolation methods investigated. For the smooth test function, the interpolation was further tested on a much sparser initial grid structure, where the number of grid points in each dimension was reduced threefold. In Fig. 5.6, a clear difference is now visible in the performance of the different interpolation methods. Again, the higher-order polynomial kernel for RBF interpolation, as well as the higher-order spline interpolation, show a higher error for all mesh refinements than the methods with underlying linear kernels or polynomials. However, in contrast to the measurements before, the Gaussian kernel for the RBF method with the same constant $c = 0.5$ now outperforms the linear kernels.

5.1.4 Findings

From a measurement point of view, the classical cubic spline and the piecewise linear spline performed the best when applied to both the discontinuous and the smooth test functions. Further, no evidence has been found that higher order interpolation or RBF interpolation describes rapidly varying quantities with any meaningful improvements, if even when compared to the other mentioned methods. The Gaussian kernel for the RBF showed great applicability when the initial data was relatively sparse, reconstructing the smooth test function most accurately. However, when the domain is sampled more often, the Gaussian method falls behind other kernels and methods used, both visually and qualitatively. Further, consistent results of the method cannot be guaranteed without tuning the parameter c to the specific problem.

Though the visualization of the interpolated data has been implemented in the *Python* script as well, one cannot overlook the simplicity with which the tuning of interpolation parameters can be achieved using *Python* Notebooks. However, one clear advantage to using the interpolation routine within the simulator itself is that the mesh size used within the simulator has to match the mesh size used to evaluate the interpolating spline. With the integrated implementation, this is ensured trivially, as the parameter is taken directly from the underlying *Lua* file used in the setup of the simulation. When using the stand-alone implementations in a pre-processing workflow, the operator currently has to manually make sure that these two parameters are indeed the same to avoid unnecessary re-evaluations of the interpolation. Depending on the type of use, the stand-alone or the integrated implementations might be preferred, with a strong argument for the stand-alone implementation being that, especially in the early stages of setting up new simulations, the interpolation might not be needed at all since the simulator will already receive all relevant quantities in the correct format and therefore the design as a lean simulator with only the core features integrated might be preferred.

Therefore, the conclusion can be drawn that for the variety of different quantities that might be encountered, linear interpolation methods are sufficient for the reconstruction of an externally provided quantity that is sufficiently smooth. Further, they avoid oscillations when encountering discontinuities, therefore sampling

the parts of the domain next to said discontinuities ideally, with the only drawback being the reduced reproduction capability of steep gradients. Both implementations offer an acceptable degree of user-friendliness. From a user interaction point of view, it may come down to taste and design principles, whether one or the other is preferred.

5.2 Effective Mass

To show the influence of different effective masses on the transport properties of the material, proof of concept simulations were performed with the implemented techniques.

All ViennaWD 2D simulations were done with a single wave packet traversing representative geometries and were initialized as a minimum-uncertainty Gaussian distribution with initial momentum along the center line of the simulation box in y-direction, with a FWHM of $7,065nm$. The simulations were performed with a time-step of $0.1fs$ and a grid size of $0.5nm$.

5.2.1 Contact

This simulation was done with the effective mass as shown above in Fig. 5.7 with a value of 0.48 for Molybdenum di-sulfide (MoS_2) [67] and 1.1 for Gold (Au) [68]. In Fig. 5.8 the electron density of a single wave-packet traversing a MoS_2 - Au contact at $20fs$, $35fs$, $55fs$, and $90fs$ simulation time is shown. The simulation was done without the potential to show the influence of the effective mass on the wave packet. Therefore, a broadening of the wave packet due to the change in effective mass as the Gaussian wave packet reaches the (MoS_2) - Au contact can be observed Fig. 5.8 (t=55 fs). The wave packet broadens due to the change in effective mass and the resulting change in its group velocity. Since Au has a higher effective mass than MoS_2, the momentum of the wave packet is reduced in its direction of motion.

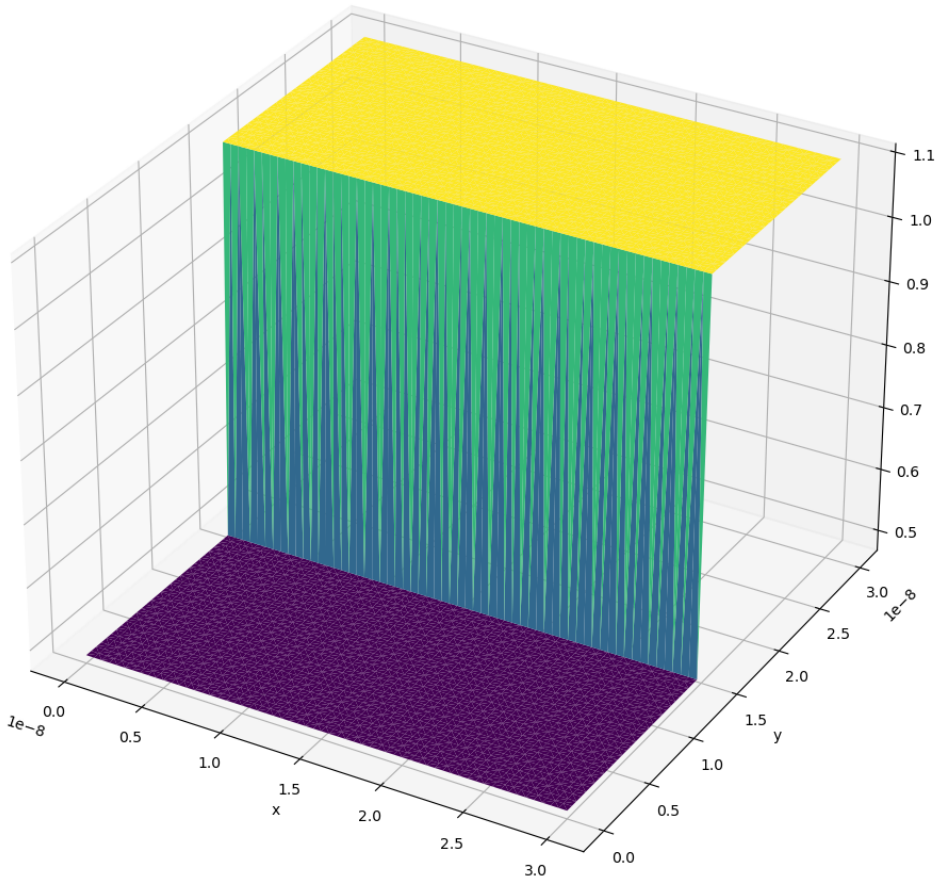


Figure 5.7: Effective mass for the MoS_2-Au contact.

5.2.2 Barrier

In Fig. 5.9, a Au layer was sandwiched between a MoS_2 and Gallium Arsenide ($GaAs$) layer, where the effective mass of $GaAs$ is taken as 0.067 [69]. Again, in Fig. 5.10, the electron density is shown at different representative timesteps. In Fig. 5.10 (t=30 fs), the wave packet starts entering the simulation domain symmetrically in the MoS_2 layer. In Fig. 5.10 (t=70 fs), the wave packet then broadens as seen previously in Fig. 5.8 (t=55 fs) when entering the Au region of the simulation domain. However, the wave packet then gets stretched back into a nearly symmetrical shape again. Further, the elongated wave packet that was previously relatively concentrated now broadens significantly within the $GaAs$ region.

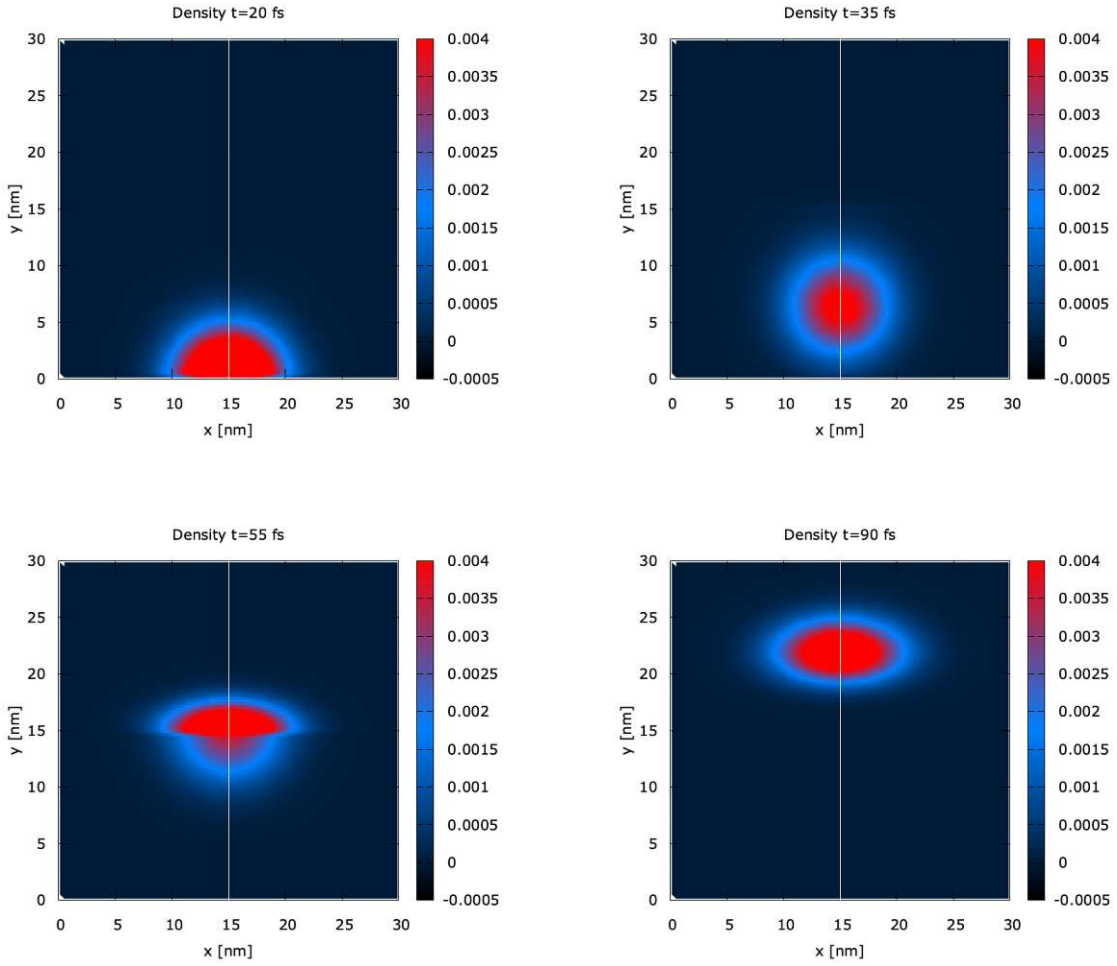


Figure 5.8: Electron density of a single wave-packet traversing a MoS_2-Au contact at $20fs$, $35fs$, $55fs$ and $90fs$ simulation time.

5.2.3 Intricate

In Fig. 5.11, an artificial benchmark case with several different materials is shown. The values for each distinct region are, however, no longer directly correlated to actual materials but rather are within a range of effective masses encountered in nanoelectronic devices. The rationale behind this particular artificial setup is loosely linked to the fact that modern nanoelectronic devices are built of various intricate shapes and geometries. This benchmark case is thus a testament to this

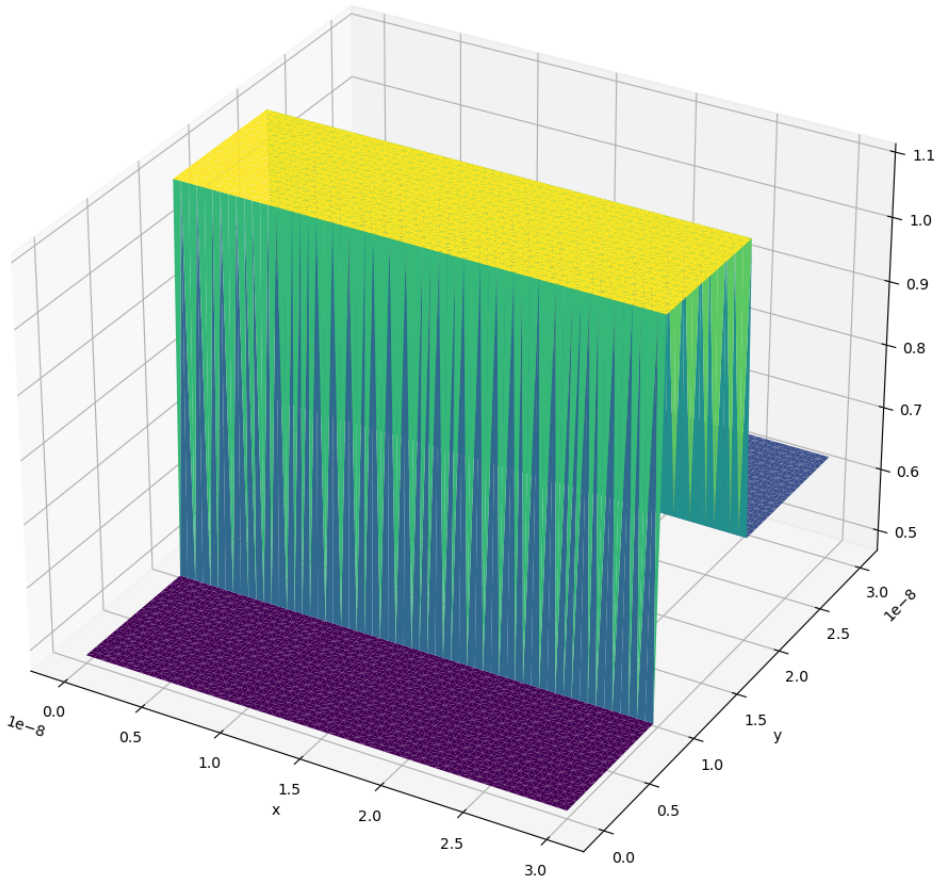


Figure 5.9: Effective mass for the Au barrier.

development. These geometries include intricate patterns of dopant distributions, gate structures, and interconnects on semiconductor substrates, as well as isolation regions. Figure 5.12 shows the evolution of the minimum uncertainty wave packet through this complex geometry. In Fig. 5.12 (t=50 fs), the initial symmetrical wave packet can already be seen wrapping around the trapezoidal region to the right of the simulation domain. However, when we consult Fig. 5.11, we can see that this is a region of very low effective mass, meaning that the mobility of the charge carriers in this region is a magnitude higher than in the surrounding regions. Therefore, the phenomenon observed is actually that part of the wave packet that reaches this region first rapidly crosses it and reconstitutes at the interface with the region diagonally through the geometry Fig. 5.12 (t=75 fs). In Fig. 5.12 (t=65 fs), at a

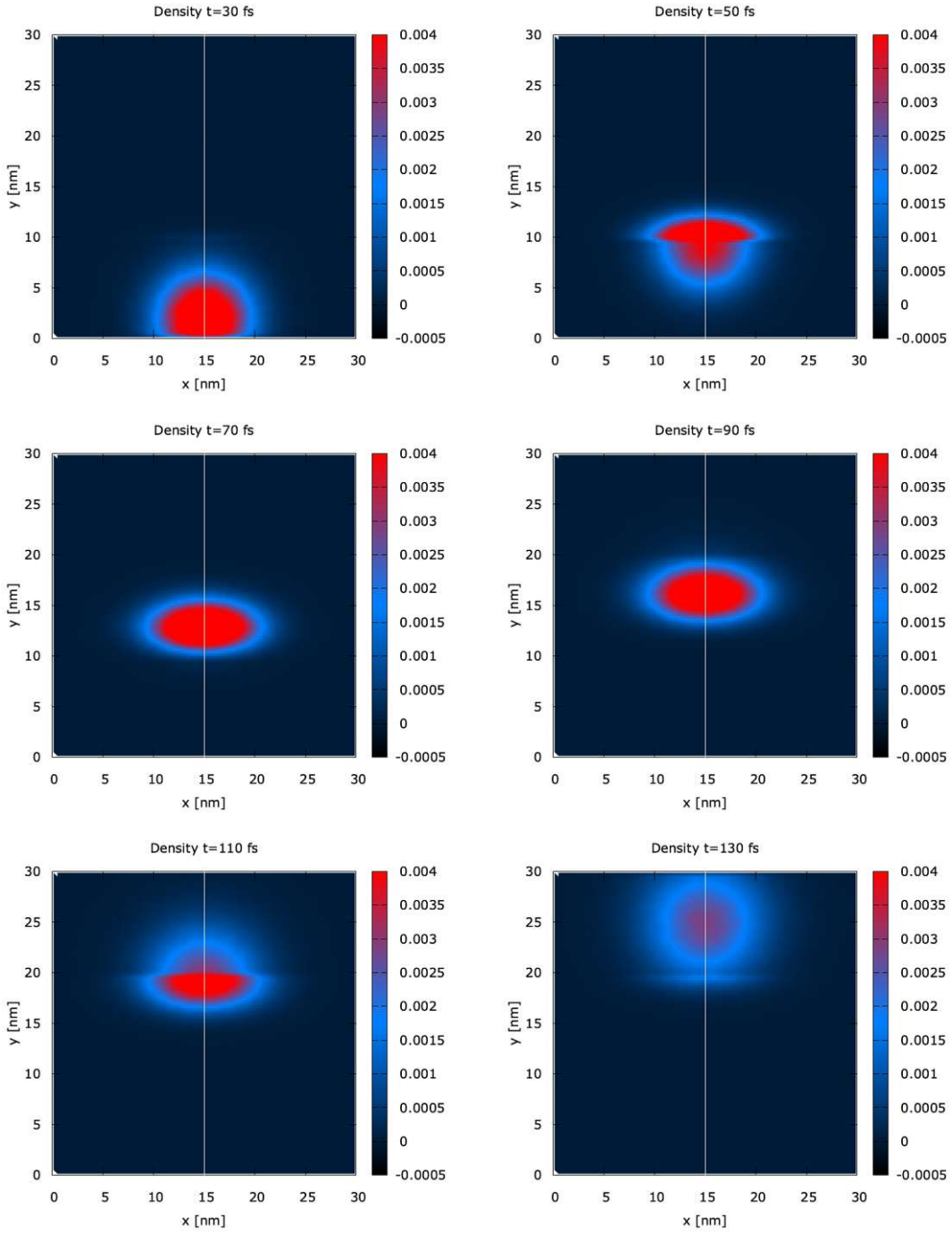


Figure 5.10: Electron density of a single wave-packet traversing a MoS_2-Au-$GaAs$ contact at $30fs$, $50fs$, $70fs$, $90fs$, $110fs$ and $130fs$ simulation time.

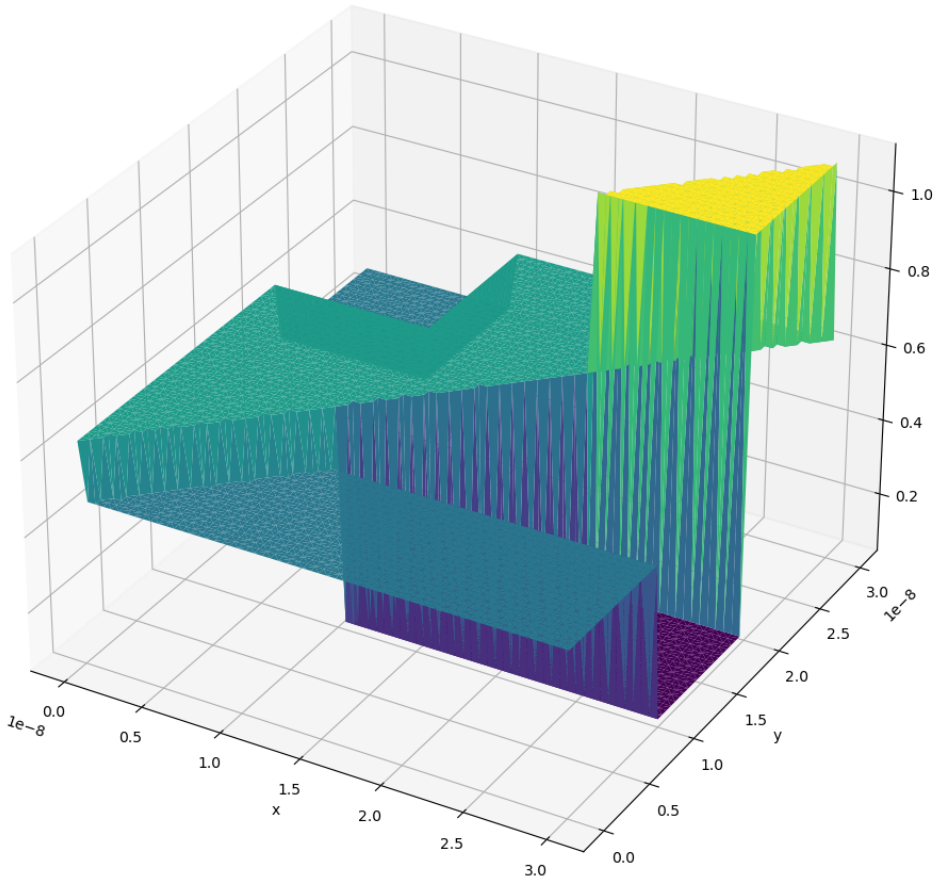


Figure 5.11: Effective mass for artificial benchmark case.

timestep between those two observations, it is clearly visible that part of the wave function is located within the highly conductive region. In Fig. 5.12 (t=85 fs), the wave packet as it has fully crossed over into the middle of the simulation domain can be seen and now appears as a spun version. This appearance can be linked to the different amounts of time that parts of the wave packet have spent within the high mobility region of the simulation domain. Finally, in Fig. 5.12 (t=140 fs), the wave packet reaches the end of the simulation domain with a region of higher charge carrier mobility to the right and a region of low mobility to the left. The final wave packet is, therefore, a highly distorted representation of the initial symmetrical wave packet.

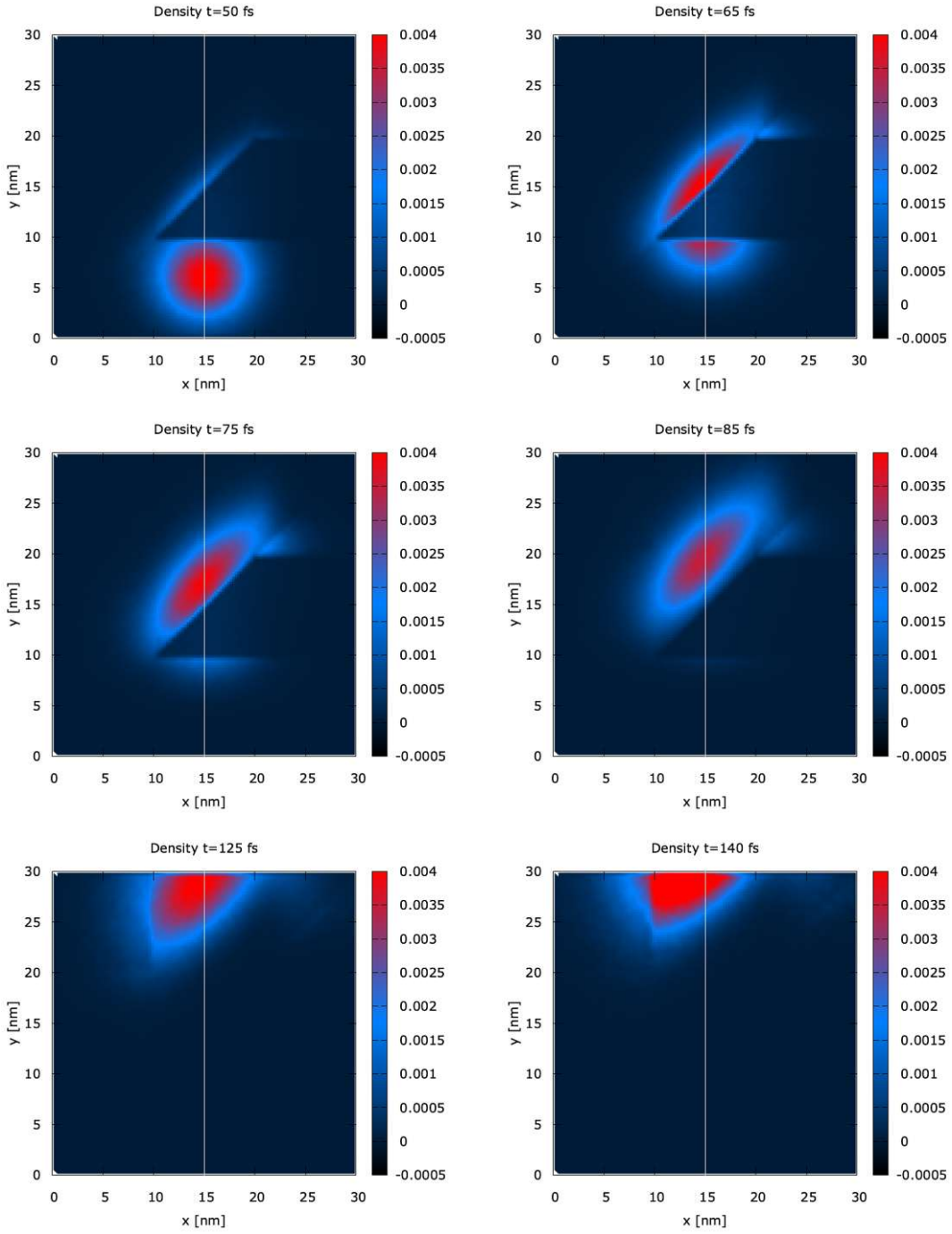


Figure 5.12: Electron density of a single wave-packet traversing intricate patterns of different materials as shown in Fig. 5.11 at $50fs$, $65fs$, $75fs$, $85fs$, $125fs$ and $140fs$ simulation time.

6 Summary

Finally, an overview of the two key contributions to ViennaWD, namely (1) the addition of an interpolation mechanism that can be used within the simulator and also as a standalone pre-processing step to map an arbitrary quantity onto the required grid structure of ViennaWD and (2) the inclusion of the spatially varying effective mass into the framework of the simulator is given. The findings that can be taken away from the results presented in Chapter 5 will be discussed in the following, with further implementation ideas and recommendations regarding the future development and possible applications of the new capabilities will be discussed.

Interpolation

In the previous section Section 5.1, evaluations of different interpolation routines were compared using an appropriate measure. Using linear spline interpolation provides us with the most optimal result for discontinuous interpolation tasks. The linear spline interpolation cannot produce any oscillations in the regions next to the discontinuity, and therefore, the surrounding regions are rendered continuous as in the original geometry. Thus, the lack of reconstruction of the step representing the discontinuity is a worthwhile trade-off for the gained reconstruction of the other features. In light of the observation that the linear spline interpolation in both the *GSL* and *Python* implementation performs best in the measure used, the decision to use linear spline interpolation as the default interpolation method is further supported. Further emphasis is added as the linear spline interpolation performs equally as well as the other interpolation methods when measured for the smooth test quantity. However, this measure depends significantly on the sampling of the original quantity. The number of grid points sampling the original quantity was chosen to represent the test function relatively precisely, which is also expected

from any experimentally measured quantities or previously modeled data on a slightly different grid. RBFs did not result in any meaningful improvements to the error measure used. On the contrary, some choices for the RBF kernel performed worse than the comparable basis functions for splines. However, the point can be made that when the experimental quantity in consideration is not sampled at a regular grid or only a minimal number of scattered data points is available, using RBFs is more or less without option. Therefore, the conclusion is that the use of a higher-order basis function for both the RBF and the spline interpolation methods is not necessary for the expected quantities that the interpolation routine will be applied to at the time of writing and that the use of linear spline interpolation is sufficient. Further extensions to this approach might include studying different interpolation methods, such as the Akima interpolation, or even machine learning approaches [49].

Effective Mass

Implementing the effective mass (Chapter 4) into ViennaWD resulted in the simulations presented in Section 5.2. For the example of three different geometries that might be encountered in modern-day nanoelectronic structures, simulations of a minimum-uncertainty wave packet were performed. These simulations showed that not only simplistic changes from one material to another, as presented in Section 5.2.1, but also very intricate geometries, such as presented in Section 5.2.3, can be modeled. Not only was the implementation of the effective mass into ViennaWD successful, as can be verified with the proof of concept simulations mentioned before, but an easy-to-use *Python* program now exists to set up such geometries. The simulations showed that the shape of a wave packet traversing such a domain can be manipulated by constructing different geometries. In further research, this mechanism could be used in conjunction with an applied electric potential to study ever more complex physical phenomena. In particular, introducing different materials into ViennaWD simulations could help study leakage currents through insulating layers in currently researched field effect devices such as GAAFETs and FinFETs.

Bibliography

[1] D.E. Stevenson and R.M. Panoff. "Experiences in building the Clemson Computational Sciences Program". In: *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. 1990, pp. 366–375. DOI: 10.1109/SUPERC.1990.130043.

[2] Dragica Vasileska and Stephen M. Goodnick. "Computational electronics". In: *Materials Science and Engineering: R: Reports* 38.5 (2002), pp. 181–236. ISSN: 0927-796X. DOI: https://doi.org/10.1016/S0927-796X(02)00039-6. URL: https://www.sciencedirect.com/science/article/pii/S0927796X02000396.

[3] David K Ferry et al. "A review of quantum transport in field-effect transistors". In: *Semiconductor Science and Technology* 37.4 (Feb. 2022), p. 043001. DOI: 10.1088/1361-6641/ac4405. URL: https://dx.doi.org/10.1088/1361-6641/ac4405.

[4] J.P. Colinge et al. "Silicon-on-insulator 'gate-all-around device'". In: *International Technical Digest on Electron Devices*. 1990, pp. 595–598. DOI: 10.1109/IEDM.1990.237128.

[5] Jin Yong Oh et al. "Demonstration of gate-all-around FETs based on suspended CVD-grown silicon nanowires". In: *2013 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2013, pp. 1–2. DOI: 10.1109/S3S.2013.6716567.

[6] Theresia Knobloch et al. "The performance limits of hexagonal boron nitride as an insulator for scaled CMOS devices based on two-dimensional materials". In: *Nature Electronics* 4.2 (2021), pp. 98–108.

[7] J.S. Tsai, Y. Nakamura, and Yu. Pashkin. "The first solid state qubit". In: *58th DRC. Device Research Conference. Conference Digest (Cat. No.00TH8526)*. 2000, pp. 93–94. DOI: 10.1109/DRC.2000.877104.

[8] David K Ferry, Xavier Oriols, and Josef Weinbub. *Quantum Transport in Semiconductor Devices*. IOP Publishing, 2023. DOI: 10.1088/978-0-7503-5237-6. URL: https://dx.doi.org/10.1088/978-0-7503-5237-6.

[9] Josef Weinbub, Paul Ellinghaus, and Mihail Nedjalkov. "Domain decomposition strategies for the two-dimensional Wigner Monte Carlo Method". In: *Journal of Computational Electronics* 14.4 (2015), pp. 922–929. DOI: 10.1007/s10825-015-0730-0. URL: https://doi.org/10.1007/s10825-015-0730-0.

[10] M. Nedjalkov et al. "Wigner Function Approach". In: *Nano-Electronic Devices: Semiclassical and Quantum Transport Modeling*. Ed. by Dragica Vasileska and Stephen M. Goodnick. New York, NY: Springer New York, 2011, pp. 289–358. DOI: 10.1007/978-1-4419-8840-9_5. URL: https://doi.org/10.1007/978-1-4419-8840-9_5.

[11] J. Weinbub and D. K. Ferry. "Recent advances in Wigner function approaches". In: *Applied Physics Reviews* 5.4 (2018), p. 041104. DOI: 10.1063/1.5046663. URL: https://doi.org/10.1063/1.5046663.

[12] David K Ferry and Mihail Nedjalkov. *The Wigner Function in Science and Technology*. IOP Publishing, 2018. DOI: 10.1088/978-0-7503-1671-2. URL: https://dx.doi.org/10.1088/978-0-7503-1671-2.

[13] Mauro Ballicchia, Josef Weinbub, and Mihail Nedjalkov. "Electron evolution around a repulsive dopant in a quantum wire: coherence effects". In: *Nanoscale* 10 (48 2018), pp. 23037–23049. DOI: 10.1039/C8NR06933F.

[14] Josef Weinbub, Mauro Ballicchia, and Mihail Nedjalkov. "Gate-controlled electron quantum interference logic". In: *Nanoscale* 14 (2022), pp. 13520–13525. DOI: 10.1039/D2NR04423D.

[15] C. Tavernier et al. “TCAD modeling challenges for 14nm FullyDepleted SOI technology performance assessment”. In: *2015 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. 2015, pp. 4–7. DOI: 10.1109/SISPAD.2015.7292244.

[16] Jian Ping Sun et al. “Resonant tunneling diodes: models and properties”. In: *Proceedings of the IEEE* 86.4 (1998), pp. 641–660. DOI: 10.1109/5.663541.

[17] L. L. Chang, L. Esaki, and R. Tsu. “Resonant tunneling in semiconductor double barriers”. In: *Applied Physics Letters* 24.12 (June 1974), pp. 593–595. ISSN: 0003-6951. DOI: 10.1063/1.1655067. eprint: https://pubs.aip.org/aip/apl/article-pdf/24/12/593/18429549/593_1_online.pdf. URL: https://doi.org/10.1063/1.1655067.

[18] Christopher Bäuerle et al. “Coherent control of single electrons: a review of current progress”. In: *Reports on Progress in Physics* 81.5 (2018), p. 056503. DOI: 10.1088/1361-6633/aaa98a.

[19] Craig S. Lent and David J. Kirkner. “The quantum transmitting boundary method”. In: *Journal of Applied Physics* 67.10 (May 1990), pp. 6353–6359. ISSN: 0021-8979. DOI: 10.1063/1.345156. eprint: https://pubs.aip.org/aip/jap/article-pdf/67/10/6353/18634383/6353_1_online.pdf. URL: https://doi.org/10.1063/1.345156.

[20] Mahdi Pourfath. *The non-equilibrium Green’s function method for nanoscale device simulation*. Vol. 3. Springer, 2014.

[21] M. Nedjalkov et al. “Physical scales in the Wigner–Boltzmann equation”. In: *Annals of Physics* 328 (2013), pp. 220–237. ISSN: 0003-4916. DOI: https://doi.org/10.1016/j.aop.2012.10.001. URL: https://www.sciencedirect.com/science/article/pii/S0003491612001558.

[22] Paul Ellinghaus. “Two-dimensional Wigner Monte Carlo simulation for time-resolved quantum transport with scattering”. PhD thesis. Wien, 2016. DOI: https://doi.org/10.34726/hss.2016.35764.

[23] J. Weinbub and D. K. Ferry. "Recent advances in Wigner function approaches". In: *Applied Physics Reviews* 5.4 (Oct. 2018), p. 041104. ISSN: 1931-9401. DOI: 10.1063/1.5046663. eprint: https://pubs.aip.org/aip/apr/article-pdf/doi/10.1063/1.5046663/13187645/041104_1_online.pdf. URL: https://doi.org/10.1063/1.5046663.

[24] Ansgar Jüngel. *Transport equations for semiconductors*. eng. Lecture notes in physics. Berlin [u.a.]: Springer, 2009. ISBN: 3540895256. DOI: https://doi.org/10.1007/978-3-540-89526-8.

[25] F. Poupaud. "About Boltzmann Equations for Transport Modeling in Semiconductors". In: *Simulation of Semiconductor Devices and Processes*. Ed. by Siegfried Selberherr, Hannes Stippel, and Ernst Strasser. Vienna: Springer Vienna, 1993, pp. 17–20. ISBN: 978-3-7091-6657-4.

[26] Craig S. Lent and David J. Kirkner. "The quantum transmitting boundary method". In: *Journal of Applied Physics* 67.10 (May 1990), pp. 6353–6359. ISSN: 0021-8979. DOI: 10.1063/1.345156. eprint: https://pubs.aip.org/aip/jap/article-pdf/67/10/6353/8010618/6353_1_online.pdf. URL: https://doi.org/10.1063/1.345156.

[27] Robert Kosik. "Numerical challenges on the road to NanoTCAD". PhD thesis. 2004.

[28] Fausto Rossi and Tilmann Kuhn. "Theory of ultrafast phenomena in photoexcited semiconductors". In: *Rev. Mod. Phys.* 74 (3 Aug. 2002), pp. 895–950. DOI: 10.1103/RevModPhys.74.895. URL: https://link.aps.org/doi/10.1103/RevModPhys.74.895.

[29] E. Wigner. "On the Quantum Correction For Thermodynamic Equilibrium". In: *Phys. Rev.* 40 (5 June 1932), pp. 749–759. DOI: 10.1103/PhysRev.40.749. URL: https://link.aps.org/doi/10.1103/PhysRev.40.749.

[30] Dietrich Leibfried, Tilman Pfau, and Christopher Monroe. "Shadows and Mirrors: Reconstructing Quantum States of Atom Motion". In: *Physics Today* 51.4 (Apr. 1998), pp. 22–28. ISSN: 0031-9228. DOI: 10.1063/1.882256. eprint: https://pubs.aip.org/physicstoday/article-pdf/51/4/22/8312867/22_1_online.pdf. URL: https://doi.org/10.1063/1.882256.

[31] William R. Frensley. “Boundary conditions for open quantum systems driven far from equilibrium”. In: *Rev. Mod. Phys.* 62 (3 July 1990), pp. 745–791. DOI: 10.1103/RevModPhys.62.745. URL: https://link.aps.org/doi/10.1103/RevModPhys.62.745.

[32] A. D’Amico et al. *From Nanostructures to Nanosensing Applications: Proceedings of the International School of Physics "Enrico Fermi", Varenna on Lake Como, Villa Monastero, 20-30 July 2004*. From Nanostructures to Nanosensing Applications: Proceedings of the International School of Physics "Enrico Fermi", Varenna on Lake Como, Villa Monastero, 20-30 July 2004. IOS Press, 2005. ISBN: 9781586035273. URL: https://books.google.at/books?id=PeXjB0eyP7sC.

[33] M. Nedjalkov et al. “Unified particle approach to Wigner-Boltzmann transport in small semiconductor devices”. In: *Phys. Rev. B* 70 (11 Sept. 2004), p. 115319. DOI: 10.1103/PhysRevB.70.115319. URL: https://link.aps.org/doi/10.1103/PhysRevB.70.115319.

[34] Damien Querlioz and Philippe Dollfus. “The Wigner Monte Carlo Method for Nanoelectronic Devices: A Particle Description of Quantum Transport and Decoherence”. In: *The Wigner Monte Carlo Method for Nanoelectronic Devices: A Particle Description of Quantum Transport and Decoherence* (Mar. 2013). DOI: 10.1002/9781118618479.

[35] V. Sverdlov et al. “Current transport models for nanoscale semiconductor devices”. In: *Materials Science and Engineering: R: Reports* 58.6 (2008), pp. 228–270. ISSN: 0927-796X. DOI: https://doi.org/10.1016/j.mser.2007.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0927796X07001088.

[36] J. Weinbub and D. K. Ferry. “Recent advances in Wigner function approaches”. In: *Applied Physics Reviews* 5.4 (Oct. 2018), p. 041104. ISSN: 1931-9401. DOI: 10.1063/1.5046663. eprint: https://pubs.aip.org/aip/apr/article-pdf/doi/10.1063/1.5046663/13187645/041104_1_online.pdf. URL: https://doi.org/10.1063/1.5046663.

[37] David K Ferry and Mihail Nedjalkov. *The Wigner Function in Science and Technology*. 2053-2563. IOP Publishing, 2018. ISBN: 978-0-7503-1671-2. DOI: 10.1088/978-0-7503-1671-2. URL: https://dx.doi.org/10.1088/978-0-7503-1671-2.

[38] Gurov Todor Dimov Ivan. " Monte Carlo Algorithm for Solving Integral Equations with Polynomial Non-Linearity. Parallel Implementation". In: (2000). ISSN: 0204-9805.

[39] ViennaWD – Wigner Ensemble Monte Carlo Simulator. URL: https://viennawd.sourceforge.%20net..

[40] "In: The Salishan Conference on High Speed Computing". In: 2011. URL: http://www.lanl.gov/conferences/salishan/.

[41] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. "Chapter 2 - HPC Architecture 1: Systems and Technologies". In: *High Performance Computing*. Ed. by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. Boston: Morgan Kaufmann, 2018, pp. 43–82. ISBN: 978-0-12-420158-3. DOI: https://doi.org/10.1016/B978-0-12-420158-3.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780124201583000022.

[42] MPI: A Message-Passing Interface Standard, Version 4.0. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[43] M. Anderson T. Sterling and M. Brodowicz. *High Performance Computing: Modern Systems and Practices*. Elsevier, 2018. ISBN: 978-0-12-420158-3.

[44] Günther Nürnberger. *Approximation by spline functions*. eng. Berlin [u.a.]: Springer, 1989. ISBN: 3540516182.

[45] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. "Polynomial Interpolation". In: *Numerical Mathematics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 333–377. ISBN: 978-3-540-49809-4. DOI: 10.1007/978-3-540-49809-4_8. URL: https://doi.org/10.1007/978-3-540-49809-4_8.

[46] Bengt Fornberg and Julia Zuev. "The Runge phenomenon and spatially variable shape parameters in RBF interpolation". In: *Computers Mathematics with Applications* 54.3 (2007), pp. 379–398. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2007.01.028. URL: https://www.sciencedirect.com/science/article/pii/S0898122107002210.

[47] M. J. D. Powell. *Approximation Theory and Methods*. Cambridge University Press, 1981.

[48] Robert Schaback. "Creating Surfaces from Scattered Data Using Radial Basis Functions". In: 1995.

[49] Jin Li et al. "Application of machine learning methods to spatial interpolation of environmental variables". In: *Environmental Modelling Software* 26.12 (2011), pp. 1647–1659. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2011.07.004. URL: https://www.sciencedirect.com/science/article/pii/S1364815211001654.

[50] Damiana Lazzaro and Laura Bacchelli Montefusco. "Radial basis functions for the multivariate interpolation of large scattered data sets". In: *Journal of Computational and Applied Mathematics* 140 (2002), pp. 521–536. URL: https://api.semanticscholar.org/CorpusID:53383888.

[51] Rolland L. Hardy. "Multiquadric equations of topography and other irregular surfaces". In: *Journal of Geophysical Research (1896-1977)* 76.8 (1971), pp. 1905–1915. DOI: https://doi.org/10.1029/JB076i008p01905. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JB076i008p01905. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB076i008p01905.

[52] Richard Franke. "Scattered data interpolation: tests of some methods". In: *Mathematics of Computation* 38 (1982), pp. 181–200. URL: https://api.semanticscholar.org/CorpusID:8290519.

[53] G. Nürnberger and F. Zeilfelder. "Developments in bivariate spline interpolation". In: *Journal of Computational and Applied Mathematics* 121.1 (2000), pp. 125–152. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-

0427(00)00346-0. URL: https://www.sciencedirect.com/science/article/pii/S0377042700003460.

[54] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[55] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[56] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[57] Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[58] *Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*. Standard. Geneva, CH: International Organization for Standardization, Mar. 1985.

[59] Brian E. Granger and Fernando Pérez. "Jupyter: Thinking and Storytelling With Code and Data". In: *Computing in Science Engineering* 23.2 (2021), pp. 7–14. DOI: 10.1109/MCSE.2021.3059263.

[60] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[61] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[62] Massimo V. Fischetti et al. "Semiclassical and Quantum Electronic Transport in Nanometer-Scale Structures: Empirical Pseudopotential Band Structure, Monte Carlo Simulations and Pauli Master Equation". In: *Nano-Electronic Devices: Semiclassical and Quantum Transport Modeling*. Ed. by Dragica Vasileska and Stephen M. Goodnick. New York, NY: Springer New York, 2011, pp. 183–247. ISBN: 978-1-4419-8840-9. DOI: 10.1007/978-1-4419-8840-9_3. URL: https://doi.org/10.1007/978-1-4419-8840-9_3.

[63] Cristina Medina-Bailon et al. “Impact of the Effective Mass on the Mobility in Si Nanowire Transistors”. In: *2018 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. 2018, pp. 297–300. DOI: 10.1109/SISPAD.2018.8551630.

[64] Deji Akinwande et al. “Graphene and two-dimensional materials for silicon technology”. In: *Nature* 573.7775 (2019), pp. 507–518.

[65] Otfried Madelung. “Fundamentals”. In: *Introduction to Solid-State Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 1–16. ISBN: 978-3-642-61885-7. DOI: 10.1007/978-3-642-61885-7_1. URL: https://doi.org/10.1007/978-3-642-61885-7_1.

[66] John P. Boyd and Fei Xu. “Divergence (Runge Phenomenon) for least-squares polynomial approximation on an equispaced grid and Mock–Chebyshev subset interpolation”. In: *Applied Mathematics and Computation* 210.1 (2009), pp. 158–168. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2008.12.087. URL: https://www.sciencedirect.com/science/article/pii/S0096300308009867.

[67] Kristen Kaasbjerg, Kristian Thygesen, and Karsten Jacobsen. “Phonon-Limited Mobility in n-Type Single-Layer MoS2 from First Principles”. In: *Physical Review B (Condensed Matter and Materials Physics)* 85 (Mar. 2012), p. 115317. DOI: 10.1103/PhysRevB.85.115317.

[68] N.W. Ashcroft and N.D. Mermin. *Solid State Physics*. HRW international editions. Holt, Rinehart and Winston, 1976. ISBN: 9780030839931. URL: https://books.google.at/books?id=oXIfAQAAMAAJ.

[69] P. Lawaetz. “Valence-Band Parameters in Cubic Semiconductors”. In: *Phys. Rev. B* 4 (10 Nov. 1971), pp. 3460–3467. DOI: 10.1103/PhysRevB.4.3460. URL: https://link.aps.org/doi/10.1103/PhysRevB.4.3460.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Ort, Datum

Unterschrift

Name