David Gratzl, B.Sc.

Simulation of graph-based 3D-microstructure models by means of R-Trees

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Mathematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dr.rer.nat. Matthias Neumann B.Sc. M.Sc.

Institute of Statistics

Graz, February 2026

Abstract

A method for efficiently simulating certain graph-based 3D models is presented. While the exact model we are interested in can be adjusted to describe a wide range of microstructures, its intended use was to describe three-phase microstructures which appear in anodes of solid oxide fuel cells. The model is based on three random graphs in space. Then, these structures are simulated on a 3D grid, where each voxel is assigned to the phase whose underlying graph is closest to it. For larger grids, these nearest neighbour computations can take a long time when using conventional data structures. This is why an exact simulation of the grid was deemed infeasible, when the model was first conceptualized and an approximative approach was used instead. The method described here uses R-trees, which are specialized data structures intended to be used for multi-dimensional spatial search, to significantly cut down on the running time of the computation. In conjunction with several other optimizations that speed up the simulation, a method for the exact computation of the grid is achieved, which is a practically viable option. A comparison of the running times of the different approaches is presented. Since it is now feasible to exactly simulate the 3D grid, the accuracy of the approximative approach was investigated.

Kurzfassung

Es wird ein Verfahren zum Simulieren von bestimmten Graphen-basierten 3D-Modellen vorgestellt. Während das genaue Modell, welches uns interessiert, angepasst werden kann um eine große Palette an Mikrostrukturen zu beschreiben, war es ursprünglich für dreiphasige Mikrostrukturen vorgesehen, welche in Anoden von Festoxidbrennstoffzellen auftreten. Das Modell basiert auf drei zufälligen Graphen im Raum. Die Strukturen werden dann in einem 3D-Zellgitter simuliert, wobei jeder Voxel derjenigen Phase zugeordnet wird, deren zugrundeliegender Graph ihm am nächsten liegt. Diese Nearest-Neighbour Berechnungen können für größere Zellgitter jedoch sehr lange dauern, wenn man mit konventionellen Datenstrukturen arbeitet. Deshalb wurde eine exakte Simulation des Zellgitters als nicht praktikabel gesehen, als das Modell erstmalig konzipiert wurde. Stattdessen wurde ein approximativer Ansatz verwendet. Das hier beschriebene Verfahren verwendet R-Bäume um die Laufzeit der Berechnung stark zu reduzieren. R-Bäume sind eine spezialisierte Datenstruktur, die vor allem bei mehrdimensional Nearest-Neighbour Berechnungen Anwendung finden. In Verbindung mit mehreren Leistungsoptimierungen erhält man ein Verfahren zur exakten Berechnung des Zellgitters, welches effizient genug ist um in der Praxis Verwendung finden kann. Die Laufzeiten der verschiedenen Verfahren werden miteinander verglichen. Da das Zellgitter nun in einer annehmbaren Zeit exakt berechnet werden kann, wurde außerdem die Genauigkeit des approximativen Ansatzes untersucht.

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich beim Erstellen dieser Arbeit unterstützt haben. Allen voran möchte ich mich bei Ass. Prof. Dr. Matthias Neumann bedanken, der nicht nur meine Masterarbeit betreut hat und jederzeit für Fragen offen war, sondern auch darüber hinaus eine große Hilfe für meine weitere akademische Laufbahn war.

Meinen Freunden Julia, Guido und Felix danke ich für den emotionalen Rückhalt über die Dauer meines gesamten Studiums und Georg dafür, dass ich mit meinen Fragen jederzeit zu ihm kommen konnte.

Zuletzt möchte ich mich bei meinen Eltern für die langjährige Unterstützung bedanken, ohne welche mein Studium nicht möglich gewesen wäre.

Contents

List of Figures

List of Tables

1 Introduction

With global energy demands constantly on the rise, new and efficient ways to produce green electricity are always needed to reduce the global share of fossil fuels. This is why solid oxide fuel cells have garnered interest from the energy sector, as they are a reliable and durable source of green energy. These energy cells convert the chemical energy of their fuel directly into electricity with very high efficiencies. While usual combustion engines achieve efficiencies of around 30%, solid oxide fuel cells can reach up to double those values [1]. All these reasons make the cells interesting enough to be researched in depth. However, investigating solid oxide fuel cells is a hard task, since their effectiveness is dependent on the morphology at the micro- and nanoscopic scales. Resolving the morphology at the micro- or nanoscale by 3D-imaging is quite costly and therefore the amount of experimental data is limited.

Thus simulations based on stochastic microstructure modelling are very common. The idea is to fit a parametric stochastic microstructure model to a relatively small set of image data, which is then used to generate digital twins. In this case digital twin refers to virtual microstructures with properties that are similar to those in the observed data [2]. One can then test macroscopic properties of these simulated structures, like its conductivity or its permeability. This allows for an easier study of the relationship between its microstructure and its properties. In some cases it is also possible to find relationships between production parameters and microstructure characteristics with the help of these stochastic models. In certain cases one can also simulate new virtual microstructures, which have not been built in reality, using predictive simulations [3]. This is done by using systematic variation of the model parameters, resulting in a large range of virtual microstructures, which are still realistic, but can also be quite different from the original image data used to build the model [2]. Another advantage of simulating the structures is, that the only restriction on how big the sampling window can be is the available computational power and time. This means that most of the time the simulated structures can be much bigger than the real ones, which allows for an investigation of the local heterogeneity of the microstructure, which may not be possible for smaller observation windows. Therefore, one can create a large database consisting of these virtual microstructures, which would usually not be feasible using only imaging data of real microstructures [3].

For all these reasons it would be great to be able to efficiently simulate the microstructures at hand. Exactly this was done in [4], where a model was constructed and simulated on a discrete voxel grid. However, since an exact simulation of each voxel was not feasible using conventional data structures, as this would take an unreasonably long time, an approximation was used instead. The main goal of my work is to speed up the exact simulation and actually making it a viable option, by using a special type of data structure. Another part of my work was to then compare the two simulation approaches. This was done by not only comparing the exactness of the approximation, where the amount of correctly assigned voxels is considered, but also looking at how this changes different morphological characteristics, which can be computed from the simulated cells.

The method for simulating microstructures which we will describe in this thesis is based on multiple graphs in $\mathbb{R}^3$ and the proximity of its points to these graphs. The idea is, that each graph corresponds to one phase within the structure to be simulated. Within these graph-based models we have quite a lot of freedom to change the resulting microstructure. With these types of models, one usually starts with random points in space. Depending on the structure, we would want these points to be distributed independently of each other or maybe even form clusters. One example of a structure based on clustered points is described in [2], where a completely different type of material is simulated. The random points form the vertices of the graphs, so we still need edges to connect them. There are several rules one can apply on how the vertices are connected, some of which cause the graphs to be connected while others may leave vertices isolated.

A conceptually different type of three-phase microstructure model is described in [3], where the same fuel cells are simulated. However, instead of graphs it uses Gaussian random fields in $\mathbb{R}^3$ to assign the voxels to the corresponding phases. These different types of models have pros and cons. One model might always exhibit certain properties which are desirable, while for the other the relationships between model parameters and characteristics of the microstructure might be easier to understand. Also not every type of model might be appropriate for some structures. This is why choosing the correct type of model is one of the most important parts of such a simulation.

2 Solid Oxide Fuel Cells

2.1 Basic Idea

The following short introduction to solid oxide fuel cells as well as additional information on them can be found in [5].

Like already mentioned, fuel cells are used to generate electricity. They do this by converting chemical energy into electric energy, similarly to a battery. The difference however is, that the fuel is supplied from an external source. The fuel cells which we consider are called solid oxide fuel cells, as all of their components are solid. They operate at high temperatures, usually between 500 and 1000 $°C$.

Due to their construction and high operating temperature, the energy cells have several advantages over usual types of electricity generators, including higher efficiency and reliability, as well as being flexible in terms of the fuel they use as well as the sizes in which they come. Because their by-products are non-polluting, they are also more environmentally friendly. As mentioned in the introduction, they are also exceptionally good at converting chemical to electrical energy, especially when compared to combustion engines. They reach conversion efficiencies of around 50%, which can be further increased to 70% when the heat which is produced in the process is used in combined heat and power applications.

A single cell in the simplest type of fuel cell consists of a porous anode and cathode as well as a dense electrolyte between them. Usually the electrolytes are oxygen ion (O^{2-}) conductors. In general a solid oxide fuel cell works by using hydrogen (H^2) as fuel, which is oxidized at the anode and reducing oxygen (O_2) at the cathode into O^{2-}. These oxygen ions then diffuse through the electrolyte to the anode where they react with hydrogen ions H^+ to produce water. This reaction sets free electrons which flow from the anode to the cathode to generate power. Since these cells run on external fuel, this process can continue as long as hydrogen and oxygen ions are supplied.

The chemical reaction which takes place is a redox reaction, which is described in the following equations:

$$\text{Anode reaction:} \quad 2H_2 \rightarrow 4H^+ + 4e^-$$

$$\text{Cathode reaction:} \quad O_2 + 4e^- \rightarrow 2O^{2-}$$

$$\text{Overall reaction:} \quad 2H_2 + O_2 \rightarrow 2H_2O$$

If another type of fuel, like carbon monoxide or hydrocarbons, such as methane, is used, then these reactions will look slightly different, but the idea stays the same.

2.2 Overview of anode

The anode is used to oxidise the fuel as well as for transporting away the electrons from where the reaction takes place to some collector outside the cell. The material which is used for the anode should also allow the fuel gas to diffuse to the reaction site as well as being able to transport away the reaction products from there.

The material for the anode needs to fulfill several requirements with regards to its structure and its electrical and electrochemical properties. Since the anode operates under extreme conditions, the material should be chemically stable and not degrade too easily. Since the electricity is transported out of the cell through this material, it should be an excellent conductor for electricity. To transport the fuel to the reaction sites and the by-products away from it, it should have a continuous porous structure, that allows for rapid transport. Ideally the material should also be cheap and also allow for an easy fabrication of the cells.

The reaction sites, which we were referencing earlier, are called the triple phase boundary. These regions are all places inside the cell where the oxygen ion conductor and the porous structure inside the anode meet with the electron conducting phase. The fuel, which in our case is H_2, diffuses through the pores of the electricity conducting material and oxidises somewhere within this triple phase boundary. There it reacts with O^{2-} to produce water and electricity. Since all three phases meet here, the electricity can then also be transported

out of the cell. For the purposes of our simulation we will only consider the fuel cells in terms of these three phases.

To maximize the efficiency of the SOFC, not only does the size or rather length of the triple phase boundary need to be as large as possible, but its parts also have to be connected to the respective sources of fuel and oxygen ions as well as to the electricity collector to be what is called electrochemically active. This means part of the triple phase boundary becomes inactive, if oxygen ions or fuel cannot be transported there, as well as if the electrons cannot be removed from the reaction site.

However, this is not the only important characteristic for the efficiency of the fuel cells. There are several morphological features of the microstructure, which might not directly influence the electricity production, but instead have a large impact on the transport properties of the cell. These features all describe in some way how efficient the transport of the oxygen ions, fuel and electrons through the respective phases is [6]. An example of this would be the so-called constrictivity, which measures the strength of bottlenecks inside the microstructure, through which transport is restricted, or the mean geodesic tortuosity, which indicates the average length of a path through a phase [4].

A good candidate for the anode material is Nickel, as it displays the highest activity for the hydrogen oxidation reaction among similar metals. It is also a lot cheaper than a lot of alternatives, like copper or noble metals like platinum. Despite this, using nickel alone as the anode material is not advisable, as pores might get closed, because of nickel's low melting temperature. This could cause the triple phase boundary to become inactive or get lost entirely.

Its thermal expansion coefficient is a lot higher than that of yttria stabilised zirconia or YSZ for short. YSZ is often used as the electrolyte material in solid oxide fuel cells and will also be used in the energy cells which we will consider later. Using only nickel would thus lead to a quicker degradation of the cells and consequently to a loss of efficiency. While degradation is something which always has to be factored in, we will only simulate energy cells which are in pristine condition.

An alternative is to use a composite material, which consists of a mixture of electronic and ionic conductors. Usually cermets are used for this and among those Ni-YSZ cermet does a good job of fulfilling most requirements which the anode should have. While it does come with several disadvantages, like its low redox stability and the tendency of nickel coarsening after extended use, it is still widely used for anodes.

The preparation of Ni-YSZ anodes involves several steps. First, nickel-oxide and YSZ are blended into a homogeneous mixture. These anodes are then heated to high temperatures and lastly the nickel-oxide and YSZ mixture is reduced to a Ni-YSZ mixture. This reduction happens either during the operation of the energy cells or externally at the same temperatures at which the cells operate, so at 600-1000°C. During the reduction of the nickel oxide pores form within it, as it loses about 40% of its volume in the process. This means that using more NiO also means an increase in pores.

3 Point Processes

3.1 Basic Idea

Random point patterns appear in many scientific areas like medicine, biology or even forestry, as their simplicity makes them quite versatile. They can be used to represent objects by themselves, but can also be built upon when considering more complex models. An example of the latter can be found in material sciences, where they form the basis of graph based models like the one in [4].

In case of our model, we are looking for a specific type of point pattern, as not every one of them might have all the properties, which we are looking for. A random point pattern could for example form clusters, which may be too extreme for what one would perceive as random. This means ideally, our points would be somewhat spread out evenly. However, we do not want the points to be spread out too evenly, as this sort of regularity also is not what we want for our pattern. Lastly, the points should also not be restricted by any sort of minimum distance between them, as random points may be arbitrarily close to each other. This is not to say that all point patterns have to follow these criteria to be useful, as the model used in [2] is build on a pattern where the points form clusters.

Detecting and quantifying these types of effects is exactly the goal of point statistics. It describes what are called point processes, the most important of which is the Poisson point process. It is a point pattern, which exhibits none of the above mentioned properties like clustering, meaning it is truly a completely random pattern. This makes it the ideal candidate for statistical tests, as they can serve as a null hypothesis when testing for interaction between points. If, for example, one would want to know if certain patterns in a given point process are the product of any underlying interaction between them or if they can be attributed to expected fluctuations in the model, then such a hypothesis test would be sensible.

The following definitions and properties of points processes, as well as further information can be found on p.35ff of [7].

3.2 General point processes

Before going into detail about specific types of point processes, we should first answer what a point process actually is.

Mathematically speaking, a point process on $\mathbb{R}^d$ is a random variable with values in the measurable space $[\mathbb{N}, \mathcal{N}]$. Here $\mathbb{N}$ denotes the family of all sequences ϕ of points in $\mathbb{R}^d$, which satisfy the following conditions:

1. The sequences ϕ are locally finite, meaning that for any bounded Borel set B, the amount of points in the intersection of B and ϕ is finite, i.e $|B \cap \phi| < \infty$.
2. Also the sequences should be simple, which is fulfilled if no two points of ϕ coincide.

Since one way to look at point processes is to consider them as a random sequence or set of points, they are often written as $\{x_1, x_2, ..\}$ or $\{x_n\}$ instead. A second way to look at them is as a random counting measure, in which case we write $\phi(B)$ for the random number of points of ϕ which lie in the Borel set B.

The σ-algebra $\mathcal{N}$ is the smallest σ-algebra on $\mathbb{N}$ which makes all mappings $\phi \mapsto \phi(B)$ measurable, where B are all bounded Borel sets. This σ-algebra contains the so-called configuration sets, usually denoted by Y. These sets describe some point sequence property, for example the set $Y = \{\phi : \phi(B) = 0\}$, which is the set of all point sequences with no point in a given Borel set B.

With this we get that formally a point process ϕ is a measurable mapping $\phi : [\Omega, \mathcal{A}, \mathbb{P}] \to [\mathbb{N}, \mathcal{N}]$, where $[\Omega, \mathcal{A}, \mathbb{P}]$ is a probability space. This mapping generates a distribution on $[\mathbb{N}, \mathcal{N}]$, which we call the distribution P of ϕ.

This distribution is determined by the probabilities

$$P(Y) = \mathbb{P}(\phi \in Y) = \mathbb{P}(\{\omega \in \Omega : \phi(\omega) \in Y\}),$$

for $Y \in \mathcal{N}$. By $\phi \in Y$ we denote that the process ϕ has some property which defines Y. In the case of our example configuration set from before, this property would be that ϕ contains no point in B.

Also important are the finite-dimensional distributions, which are probabilities of the form

$$\mathbb{P}(\phi(B_1) = n_1, \ldots, \phi(B_k) = n_k),$$

where the B_i are Borel sets and the n_i are natural numbers including 0. The term is the probability of the event, that ϕ has n_i points in set B_i for $i = 1, \ldots, k$. The importance of these probabilities is, that the distribution P of ϕ is uniquely determined by the system of all these values for all $k \geq 1, n_1, ..., n_k \geq 0, B_1, ..., B_k \in \mathcal{B}(\mathbb{R}^d)$..

In fact, the distribution of P can even be described by the even smaller system which only includes the void-probabilities. These are given by

$$\begin{aligned}\rho_B &= P(\{\phi \in \mathbb{N} : \phi(B) = 0\}) \\ &= \mathbb{P}(\phi(B) = 0) \\ &= \mathbb{P}(\phi \cap B = \emptyset),\end{aligned}$$

for Borel sets B. From this definition we see that ρ_B simply describes the probability that the set B contains no points from ϕ. As mentioned, P is uniquely determined by the system $\{\rho_K : \text{K is compact set}\}$. This means that if two processes are defined by the same void-probabilities, then they are also the same.

Also important are the concepts of stationarity and isotropy. A point process ϕ is stationary if it is invariant under translation. By this we mean that the processes $\phi = \{x_n\}$ and $\phi + x = \{x_n + x\}$ have the same distribution for all $x \in \mathbb{R}^d$. The idea behind isotropy is similar, but instead of invariance under translation it describes invariance under rotation. If a process is both stationary and isotropic it is called motion-invariant, as it does not matter where the process is shifted to or if it is rotated, it will always have the same distribution.

The last definition we will look at is intensity. For a stationary point process the intensity λ is defined as the mean number of points in a Borel set with measure 1. So for example, in any d-dimensional unit cube, we expect to see λ points.

A more exact name for point processes would be "random point field", as the word process usually describes something which changes over time. The origin of this name is from when researches first looked into the theory. Instead of considering random points in space, they instead looked at random events in time.

3.3 Poisson point processes

For our use, we are mostly interested in the homogeneous Poisson point process. A point process ϕ is said to be a homogeneous Poisson point process, if its point counts follow a Poisson distribution, meaning that the random number of points of ϕ in a bounded Borel set B is Poisson distributed with parameter $\mu(B) = \lambda\nu_d(B)$, and also that its points are independently scattered.

Here, the constant λ is the aforementioned intensity of the process and $\nu_d(B)$ is the d-dimensional Lebesgue measure of B. And thus we get

$$\mathbb{P}(\phi(B) = m) = \frac{\mu(B)^m}{m!} \exp(-\mu(B)) \qquad m = 0, 1, 2, \ldots \tag{1}$$

When scaled to a Borel set B with arbitrary measure $\nu_d(B)$ one would thus expect to see $\lambda\nu_d(B)$ points, which is exactly the mean of the above mentioned Poisson distribution with parameter $\mu(B) = \lambda\nu_d(B)$.

For the second property, the term independent scattering means, that the number of points of ϕ in k disjoint Borel sets $B_1, \ldots, B_k$ form k independent random variables for any k. Therefore independent scattering means that

$$\mathbb{P}[\phi(B_1) = n_1, \ldots, \phi(B_k) = n_k] = \mathbb{P}[\phi(B_1) = n_1] \ldots \mathbb{P}[\phi(B_k) = n_k].$$

As mentioned, the void-probabilities can also be quite helpful when comparing different point processes. In the case of the homogeneous Poisson point process we get this probability from Equation (1) with $m = 0$ and $\mu(B) = \lambda\nu_d(B)$, which results in the void probability

$$\rho_B = \exp(-\lambda\nu_d(B)).$$

The constant λ is the characteristic parameter of a homogeneous Poisson point process, meaning that the homogeneous Poisson point process is completely defined by its intensity λ and the two properties in its definition. Note that the homogeneous Poisson point process is both stationary and isotropic.

3.4 Binomial point processes

Next we want to look at a related process, the binomial point process. To this end we first consider a single point x, which is uniformly distributed in a compact set $W \subseteq \mathbb{R}^d$. Then for every Borel set $A \subset W$ the probability that x falls in A is given by

$$\mathbb{P}(x \in A) = \frac{\nu_d(A)}{\nu_d(W)}. \tag{2}$$

This is just the relative d-dimensional volume of A in relation to W.

A binomial point process on a compact set $W \subset \mathbb{R}^d$ is the result of independently and uniformly distributing $n \geq 1$ random points over W. From the independence of the points and Equation (2) we get the probability

$$\begin{aligned}\mathbb{P}(x_1 \in A_1, \ldots, x_n \in A_n) &= \mathbb{P}(x_1 \in A_1) \ldots \mathbb{P}(x_n \in A_n) \\ &= \frac{\nu_d(A_1) \ldots \nu_d(A_n)}{\nu_d(W)^n},\end{aligned}$$

for Borel subsets A_i of W. We will denote such a process by $\phi_{W^{(n)}}$.

Like the Poisson point process, the binomial point process also gets its name from the distribution of $\phi_{W^{(n)}}(A)$, where A is a Borel subset of W. It follows that for a binomial point process, the number of points in A is Binomial distributed with $n = \phi_{W^{(n)}}(W)$ and $p(A) = \nu_d(A)/\nu_d(W)$. Since the mean of a binomial random variable is given by np, the mean number of points per unit volume, denoted by λ, is given by

$$\lambda = n\frac{\nu_d(U)}{\nu_d(W)} = \frac{n}{\nu_d(W)}, \tag{3}$$

where U is a Borel set with unit volume. Here, λ is again called the intensity of the process. The expected amount of points in any Borel subset A of W is thus

$$\mathbb{E}(\phi_{W^{(n)}}(A)) = \lambda\nu_d(A). \tag{4}$$

The probability for any point to not lie in a given compact subset A is the same as the probability of the same point lying in its complement A^c, which is

$$\frac{\nu_d(A^c)}{\nu_d(W)} = \frac{\nu_d(W) - \nu_d(A)}{\nu_d(W)}.$$

Since the void probability for a binomial point process is the probability for n independently distributed points to not lie in A, we can simply multiply these n values to get

$$\rho_A = \mathbb{P}(\phi_{W^{(n)}}(A) = 0) = \frac{(\nu_d(W) - \nu_d(A))^n}{\nu_d(W)^n}. \tag{5}$$

At the beginning of this chapter, a relation between the binomial and Poisson point processes was mentioned. They are related by the same theorem which also connects their respective distributions, the Poisson limit theorem. A version of it states the following.

Theorem 3.1 (Poisson Limit Theorem)**.** *Let $X(n, A) \sim Binom(n, p(A))$. If $n \to \infty$ and $p(A) \to 0$ such that $np(A)$ stays fixed, then $X(n, A)$ converges in distribution to a random variable which is $Poisson(\lambda\nu_d(A))$ distributed.*

If W converges to $\mathbb{R}^d$ and we let n go to ∞ such that $\lambda = n/\nu_d(W)$ stays the same, then the random variable $\phi_{W^{(n)}}(A)$ fulfills the requirements of the theorem for every fixed and bounded Borel set A. Thus, if a limiting point process ϕ exists, $\phi(A) \sim Poisson(\lambda\mu_d(A))$ should hold for every bounded Borel set A.

This limiting process thus fulfills the first property of the Poisson point process. For the second property, we need to show, that it is also independently scattered. To this end, we consider the finite dimensional distributions for a binomial point process. Given disjoint Borel sets A_i with $\cup A_i = W$ and for all $n_1, \ldots, n_k \geq 1, k \geq 2$ with $n_1 + \ldots + n_k = n$, the finite dimensional distributions are given by the following multinomial probabilities:

$$\mathbb{P}[\phi_{W^{(n)}}(A_1) = n_1, \ldots, \phi_{W^{(n)}}(A_k) = n_k] = \frac{n!}{n_1! \ldots n_k!} \frac{\nu_d(A_1)^{n_1} \ldots \nu_d(A_k)^{n_k}}{\nu_d(W)^n}$$

From this it is implied, that the above mentioned limiting process is independently scattered, i.e that the random variables $\phi(A_1), ..., \phi(A_k)$ are independent if the Borel set A_i are disjoint.

A Poisson point process is thus the limit of a binomial one. We can however also go in the other direction by conditioning a Poisson point process. This means that we restrict the process ϕ to a compact set under the condition $\phi(W) = n$. To show that the result is a binomial point process in W with n point we compare the void probabilities. We have

$$\begin{aligned}
\mathbb{P}(\phi(K) = 0 | \phi(W) = n) &= \frac{\mathbb{P}(\phi(K) = 0, \phi(W) = n)}{\mathbb{P}(\phi(W) = n)} \\
&= \frac{\mathbb{P}(\phi(K) = 0)\mathbb{P}(\phi(W \backslash K) = n)}{\mathbb{P}(\phi(W) = n)} \\
&= \frac{(\nu_d(W) - \nu_d(K))^n}{\nu_d(W)^n},
\end{aligned}$$

where K is a compact subset of W. The first equality follows from the well known Bayes' theorem and the second one from the fact, that all n points have to lie in $W \backslash K$ since none may lie in K. Since this coincides with the void probabilities obtained for the binomial point process in Equality (5), we know that a conditioning of the Poisson point process of this sort yields a binomial one.

3.5 Simulation of a Poisson point process

When modelling the microstructure of the fuel cell anodes, which are defined in the infinite space $\mathbb{R}^3$, we only generate realisations of it on compact sets instead. In particular, we will restrict ourselves to compact 3-dimensional cuboids. This means that if we want our points inside it to follow a Poisson point distribution we only have to simulate a binomial point process with a fixed number of points n as we just showed.

According to property 1 of the Poisson point process this number n should follow a Poisson distribution with parameter $\mu(B) = \lambda\nu_3(W)$, where λ will be a model parameter and $\nu_3(W)$ is the volume of the cuboid W. So as a first step we simply simulate n as a Poisson distributed random variable, which the statistical software package R [8] is easily capable of.

Since we simulate a binomial point process, the points are all uniformly distributed. In our case this is a rather simple task, since the compact space is a cuboid. Since the process is invariant under rotation, we can assume that it is aligned with all 3 axes and call the respective lower and upper bounds in each direction l_x, u_x, l_y, u_y, l_z and u_z. Starting with the x-axis, we generate a sequence of n independent values $x_1, \ldots, x_n$ where each is uniformly distributed in $[l_x, u_x]$. Doing the same for the y- and z-axis leads to the sequences $y_1, \ldots, y_n$ and $z_1, \ldots, z_n$. Then each of the n points $P_i = \{x_i, y_i, z_i\}$ is uniformly distributed and independent of each other.

In a more general setting, our compact Borel set K might not be a cuboid and not even 3-dimensional. In this case let d be its dimension and $\nu_d(K)$ its d-dimensional Borel-measure. Instead of simulating the points over K, we instead use a cuboid, which contains K. Any cuboid may be used, however if we want to minimize our runtime, we choose it to be as small as possible. As before we can assume that it is aligned with all d

axes and we call the bounds l_i and u_i for $i = 1, \ldots, d$. To calculate the number of points needed, we again generate a random number n from a Poisson distribution with parameter $\lambda\nu_d(K)$. For each point we first simulate each coordinate independently as before and then check if it is actually in K. If it is not in K, we simulate the point again. For this reason we wanted the bounding cuboid to be as small as possible, since this means that fewer points will land in the area outside of K. This is called the rejection-method, as any points not in K will be rejected.

4 Beta-Skeletons

4.1 Why use more than points?

We now have a way of constructing a bunch of points in $\mathbb{R}^3$, but we still need a way to fill the entire compact space. An idea would be to construct a Poisson point process for each of the three phases of the energy cell and then assign each point in space to the phase which is closest. This however would cause some problems. One problem would be, that the only parameters we have at the moment are λ_1, λ_2 and λ_3, which are the parameters for each of the three point processes. This means we won't have a lot of control over the simulation, since we can only influence the amount of points and by extension the relative amount of space each of the phases occupies.

The bigger problem however is, that we will not be able to ensure that the phases are completely connected, since each part of the phase needs to have a connection to its base in order to produce electricity. A situation as in Figure 1 may arise.

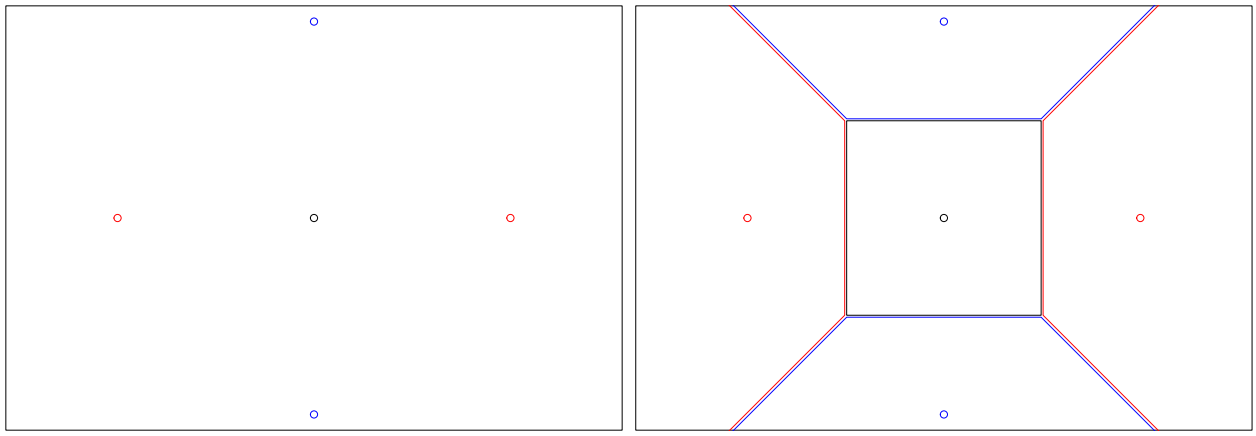

Figure 1: Example of not connected phase

While this example is in two dimensions, one can easily imagine the 3-dimensional case, where the black phase in the middle would be completely blocked off from any other part of the black phase. Since a simulated cell would have thousands of points for each phase, it is very likely for such a case to occur.

Instead of defining the simulated phases by their distance to the nearest point, one could also consider the distances to edges connecting points from the same phase. As long as the distance to an edge is 0 if a point in space lies directly on it, then phases which are defined this way would be completely connected, as long as the respective graphs consisting of points and edges are also connected. This holds as long as points and edges from different phases never occupy the same space, since in that case one would have to choose which phase the intersecting point belonged to. However, since the points and edges are continuously distributed in space and have 0 volume, the probability of such an event is 0.

4.2 Cycle-based beta-skeleton

There are many types of connected graphs, whose edge sets can be considered as rules on how to connect a given vertex set. But not all of them are useful for our purpose. For example, they might not have any possibility for parametrization, which as mentioned might come in handy when simulating the cells.

The following rule for connecting points takes in a single parameter b and returns a graph, which is called a beta-skeleton. These definitions work in any dimension, as they are only dependent on d-dimensional balls. For simulating the cells, only the 3-dimensional case is relevant, but for an easier visualisation we will restrict ourselves to two dimensions.

There are two possibilities for defining a beta-skeleton. For the first one we have the following rule for

connecting two points x and y, which one can read about in [9].
We connect x and y if there exists no third point z such that the angle between the lines xz and zy is greater than θ, where

$$\theta = \begin{cases} \sin^{-1}(\frac{1}{b}), & b \geq 1 \\ 180° - \sin^{-1}(b), & b < 1. \end{cases}$$

The space where no third point may be is called critical region $A_b(x, y)$ and its form greatly depends on b.

For $b = 1$ the angle θ is exactly 90°, therefore the critical region is a circle with x and y lying on opposing sides. The circle thus has diameter $|x - y|$ and its midpoint is $(x + y)/2$. For higher dimensions we would instead have a d-dimensional ball with the same diameter and midpoint as the circle. The resulting beta-skeleton is also called Gabriel graph [9], which has many applications like constructing a minimum spanning tree.

For $b > 1$ the angle θ is smaller than 90°, which means that any third point has to be farther away than before. This can be visualised by a circle with points x and y still on its edge but no longer at opposing ends, in which case the line segment $\overline{xy}$ is called a chord of the circle. There are however an infinite amount of such circles, which is why we need another distinguishing feature. For this we will choose the diameter, which we can easily calculate by considering the third point which lays opposite to either x or y on the circle. Without loss of generality we use x and call the opposite point z. These two split the circle in half, therefore y lays on a semicircle with edge points x and z. By the famous Thales's theorem the three points form a right triangle, where the right angle is the one laying at point y. Since we want the angle at z to be θ, we can use basic trigonometry to obtain d.

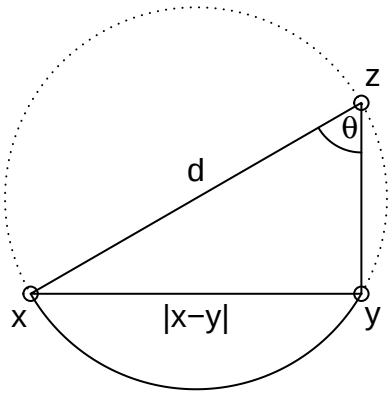

Figure 2: Right triangle inside circle bounded by x and y

Using the definition $\sin(\theta) = \text{opposite/hypotenuse}$ we get that $\theta = \sin^{-1}(|x - y|/d)$. Because also $\theta = \sin^{-1}(1/b)$ and since the arcsine function is injective we get

$$\frac{|x - y|}{d} = \frac{1}{b} \implies d = b|x - y|.$$

Thus the diameter of the circle is b multiplied by the length of the chord.

In two dimensions there are two circles which can be represented like this. The points should lie outside the dotted line in Figure 2, since that is where we get the angle θ. This makes up the larger part of the circle, which is why the critical region is the union of the two circles.

In higher dimensions there are not just two balls which can be described by the chord and diameter in this way, but an infinite amount. The critical region is thus the union of infinitely many d-dimensional balls for $d > 2$.

If $b < 1$, we get a θ between 90° and 180°, which yields a critical region that is the intersection of 2 balls. The case where $b = 1$ can therefore be seen as the transition between the other two cases, since a single ball

is the union as well as the intersection of two identical balls. The intersection of the boundaries of the two balls is again a circle with x and y as poles.

Since $\sin^{-1}(x) \to 0$ for $x \to 0$ it follows that

$$\theta \to \begin{cases} 0, & b \to \infty \\ \pi, & b \to 0. \end{cases}$$

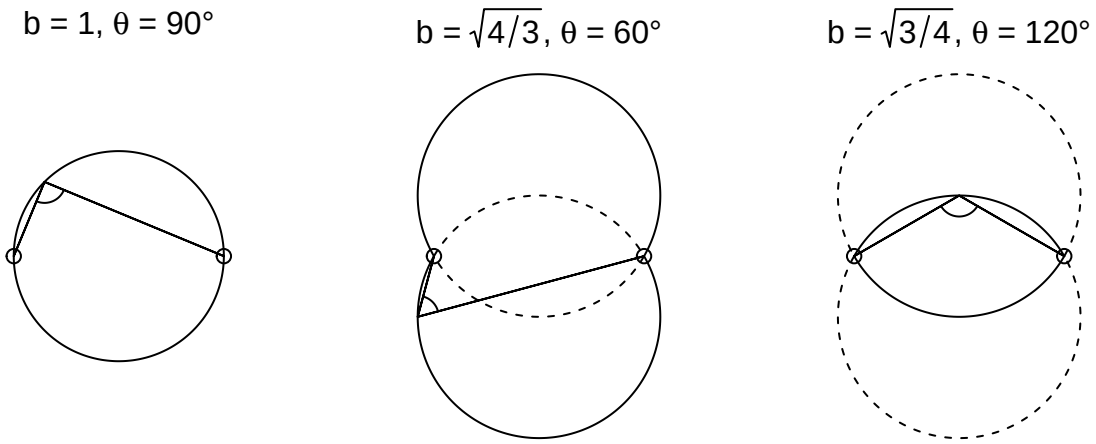

Figure 3: Critical regions for circle-based Beta-skeleton in 2D

Additionally, because a smaller angle means a larger critical region and vice versa as can be seen in Figure 3, the skeleton loses edges the larger b becomes. This can be seen in Figure 4, where we have three beta-skeletons of the same Poisson point process for different values of b. For the smaller value of b the skeleton has so many edges that they start to intersect whereas for the larger b we have so few, that the graph is not even connected. For the intermediate value of $b = 1$ we get a skeleton which results in a graph which is connected as well as planar. The connectedness holds for any Poisson point process, not just in this example, as we will see later for the other definition of a beta-skeleton.

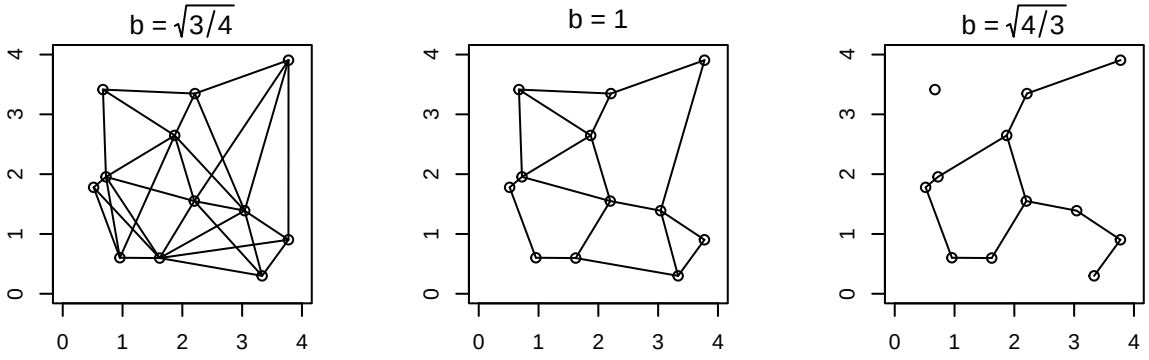

Figure 4: Circle-based Beta-skeletons of 2-dimensional Poisson-Point-Process

In fact $b = 1$ is the largest value for which connectedness can be guaranteed. For a counterexample we consider $b = 1 + \varepsilon$ for $\varepsilon > 0$ in two dimensions. Then the critical region will be the union of two circles, which will cover some area above and below either of the endpoints. While this area will be very small for small values of ε, it will never be zero. So if we consider a point process which contains any two points x and y and some third point z in this small region, then neither point will be connected to y. As can be seen in Figure 5,

the third point is just inside the critical region of x and y, so the two won't be connected. Also y and z won't be connected, as x is clearly in their critical region as well. Therefore only x and z will be connected, leaving y isolated.

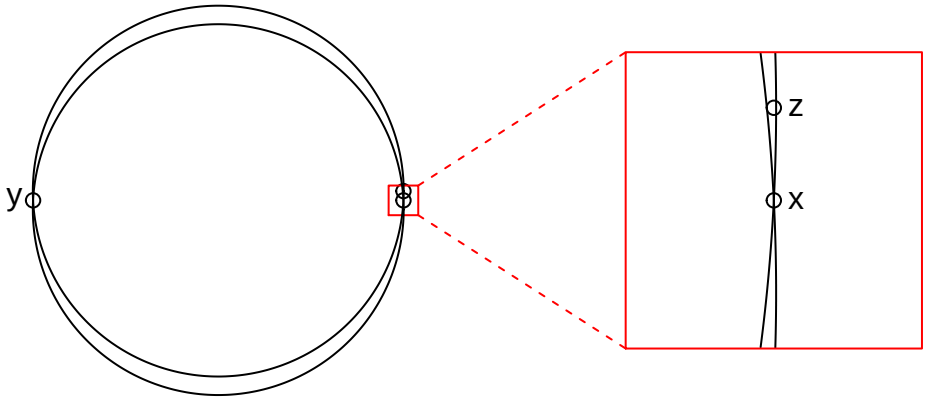

Figure 5: Counterexample for connectedness if b>1

Since we want the beta-skeleton to be connected, to ensure that each phase is completely connected, we should not choose b to be larger than 1. It would however be preferable to have a wider range of usable values for our parameter b, since one might want fewer edges without losing connectedness. This is however not possible for this definition of the beta-skeleton.

4.3 Lune-based beta-skeleton

For the fuel cells we will use a different version, which is not based on circles but on lunes. One can find the following definitions in [4]. Again we will have a critical region for each pair of points, which are only connected by an edge in the skeleton if no other point is inside this critical region. However this time the region is defined a bit different.

Given two points x and y and the parameter $b \geq 1$ we first define the two midpoints $m_1 = (b/2)x + (1-(b/2))y$ and $m_2 = (b/2)y + (1-(b/2))x$. For $b = 1$ the midpoints coincide exactly halfway between x and y whereas for $b = 2$ they coincide with x and y respectively. For b values larger than 2, the midpoints no longer lie between x and y.

For the critical region, we then define circles around the midpoints. The radii are given by the distances from the midpoints to x and y. To be more specific, the radius for the circle around m_1 is $|m_1 - y|$ and the one for the circle around m_2 is $|m_2 - x|$. With all these definitions we can define the critical region as the intersection of the two circles, so

$$A_b(x, y) = B(m_1, |m_1 - y|) \cap B(m_2, |m_2 - x|).$$

Replacing m_1 and m_2 by their definition, we get that the radii are the same, since

$$\left.\begin{aligned} |m_1 - y| &= \left|\frac{b}{2}x + \left(1 - \frac{b}{2}\right)y - y\right| = \left|\frac{b}{2}x - \frac{b}{2}y\right| \\ |m_2 - x| &= \left|\frac{b}{2}y + \left(1 - \frac{b}{2}\right)x - x\right| = \left|\frac{b}{2}y - \frac{b}{2}x\right| \end{aligned}\right\} = \frac{b}{2}|x - y|. \tag{6}$$

The critical regions for different values of b can be seen in Figure 6.

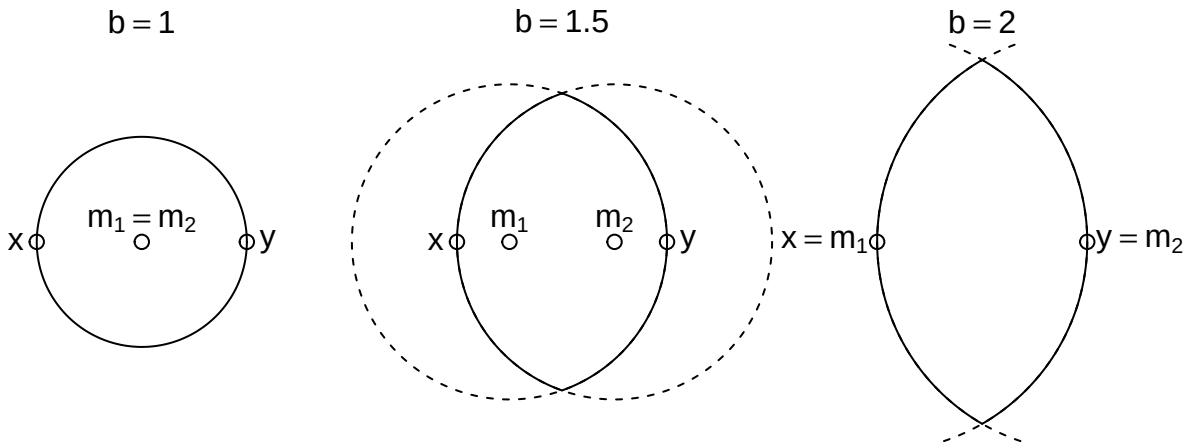

Figure 6: Critical regions for lunes-based Beta-skeleton in 2D

We can see that for $b = 1$ the two definitions of the beta-skeleton coincide, resulting in the afore mentioned Gabriel-graph.

As b gets bigger, so do the radii of the circles, resulting in a larger critical region. In Figure 6 we can see that as the circles get larger, they only grow in the direction away from the respective edge point x or y. Because of this every smaller circle is contained in every respective larger circle. This means that for $b_1 < b_2$, we get that $A_{b_1}(x, y) \subset A_{b_2}(x, y)$ holds for the respective critical regions. So just like for the circle-based skeleton the number of edges becomes smaller as b gets larger.

One should however note that the critical regions grow much slower in comparison, as can be seen from the circle-based definition for $b = \sqrt{4/3}$, which results in $\theta = 60°$. This can be directly compared with the lunes-based critical region for $b = 2$, as we can see that x and y form an equilateral triangle with either of the intersections of the the circles. This also results in an angle of 60° and therefore $A_2(x, y)$ is entirely contained within the circle-based critical region for $b = \sqrt{4/3}$. This can be seen in Figure 7 , where each graph shows the "height" of the critical regions for different values of b for both definitions. In this case height is the farthest distance inside the region from the line segment $\overline{xy}$ in relation to $|x - y|$.

These heights can be calculated by simple trigonometry, resulting in h_c for the circle-based skeleton and h_l for the lunes-based one.

$$h_c = \begin{cases} \frac{b}{2} + \frac{\sqrt{b^2-1}}{4}, b \geq 1 \\ \frac{b}{2} - \frac{\sqrt{b^2-1}}{4}, b < 1 \end{cases} \qquad h_l = \sqrt{\frac{2b-1}{4}}$$

In Figure 7 we see, that the definition of h_l extends to values below 1. According to Equation (6), the radius of each circle is $(b/2)|x - y|$, and so, for $b < 1$, it is less than half the distance between the two points. This means that for those smaller values the critical region can no longer contain either of the two points. In 2 dimensions this can result in non planar graphs and in any case, it will result in more edges. Both of these may not be a problem, since for our cells we work in 3 dimensions, where planarity is not a concern, and a skeleton with more edges could also be a better fit. But since we mostly work with values of b around 2 and because most literature assumes $b \geq 1$, we will do the same.

The definition of the critical region again only depends on circles and thus also holds in higher dimensions by replacing the circles with the respective d-dimensional balls with the same radii and midpoints. However, in contrast to the other definition, there are always only 2 balls instead of uncountably many for $d \geq 3$, because there are always only 2 midpoints. The relative neighbourhood graph is defined on a set of vertices, where two points x and y are connected by an edge in the graph, if no third point z is closer to both points than

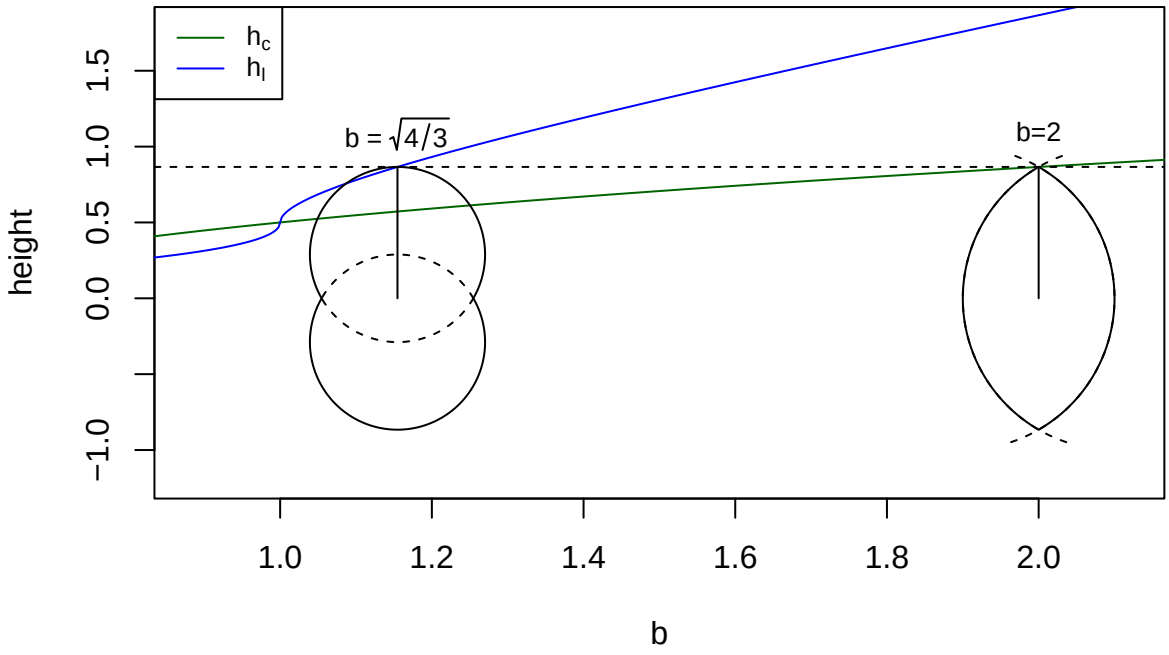

Figure 7: Comparing heights of critical regions for both definitions of beta-skeleton

they are to each other. For $b = 2$ the edges of the critical region all have distance $|x - y|$ to either x or y and thus the two definitions are the same.

Next we take a look at the resulting beta-skeletons. For this we will use the same Poisson point process as before.

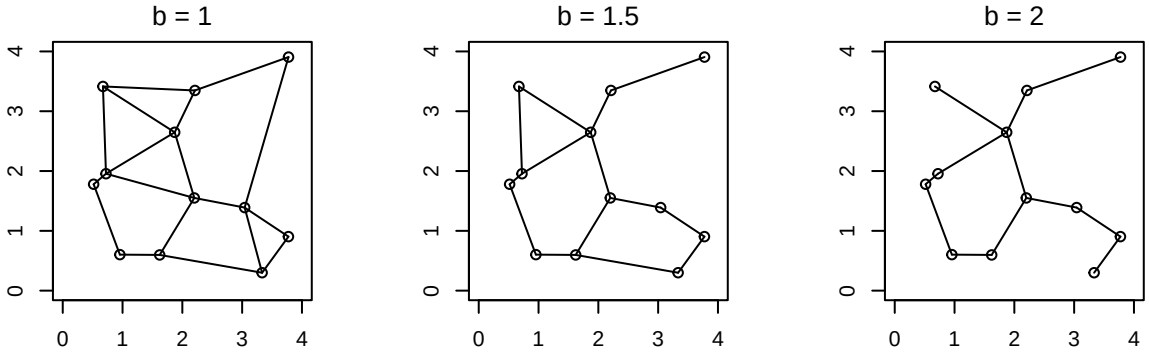

Figure 8: Lunes-based Beta-skeletons of 2-dimensional Poisson-Point-Process

The graph loses edges at a much smaller rate than for the circle-based definition. Even though the value of b went up by a lot more, the amount of edges did not decrease that much. This checks out with our findings regarding the height of the critical regions as well as the fact that regions with the same height are much smaller in the lunes-based skeleton. We can even see that for $b = 2$ the graph is still connected, which is in fact the largest value of b for which this can be guaranteed.

In practical use cases like our simulation, one might even be able to use values of $b > 2$, since, while it does

not guarantee connectivity of the skeleton, such a large value of b also does not exclude connectivity. In addition to this, while we can only be certain that all phases are completely connected if the respective skeletons are, it is also possible that gaps in the skeletons can be bridged by the phases if no other skeleton is too close, as can be seen in Figure 9.

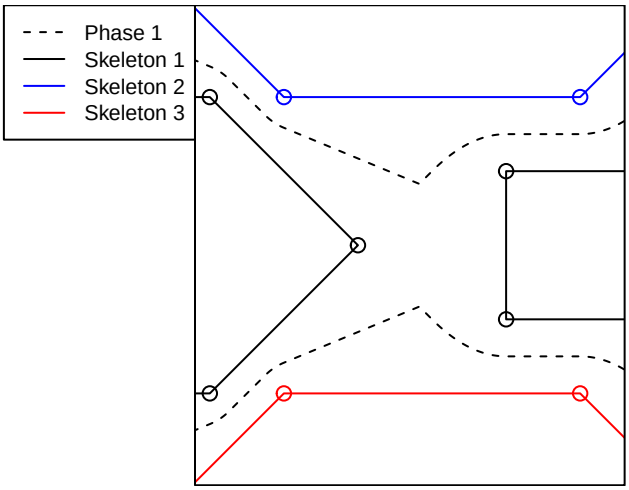

Figure 9: Bridging gaps in beta-skeletons for b>2

4.4 Additional Parameter

With the introduction of beta-skeletons we now have a lot more control over the shape as we can see in Figure 9. Using only points meant that all borders were straight lines, but now we can even create curved borders. Our current distance function is simply the minimum Euclidean distance from any given point x to the beta-skeleton B, which we will denote by $d(x, B)$.

Next we introduce another parameter $\gamma \geq 1$, which controls how much the phase centers around the vertices of the skeleton. To this end we define a second distance function

$$d'_\gamma(x, B) := \min(\gamma d(x, B), d(x, V(B))),$$

where $d(x, V(B))$ is the Euclidean distance from x to the nearest vertex of B. This means that if x would be closer to an edge than a vertex, this distance would be multiplied by γ and then compared with the distance from x to the nearest vertex. If x is already closer to a vertex than to an edge, then $d(x, B) = d(x, V(B))$ and since $\gamma \geq 1$ the minimum of the two is always $d(x, V(B))$.

A point therefore has to be much closer to an edge to be considered part of the respective phase than it has to be for a vertex. To visualize this, we look at the contour lines of the distances to a skeleton for different values of γ.

Each contour line corresponds to the same distance in all three figures. As we can see, the lines stay at the same distance around the vertices for all values of γ, but get closer to the edges as it gets larger. For $\gamma = 1$ the new distance function d'_γ is the same as the old one, which is why the contour lines do not bulge around the vertices in this case.

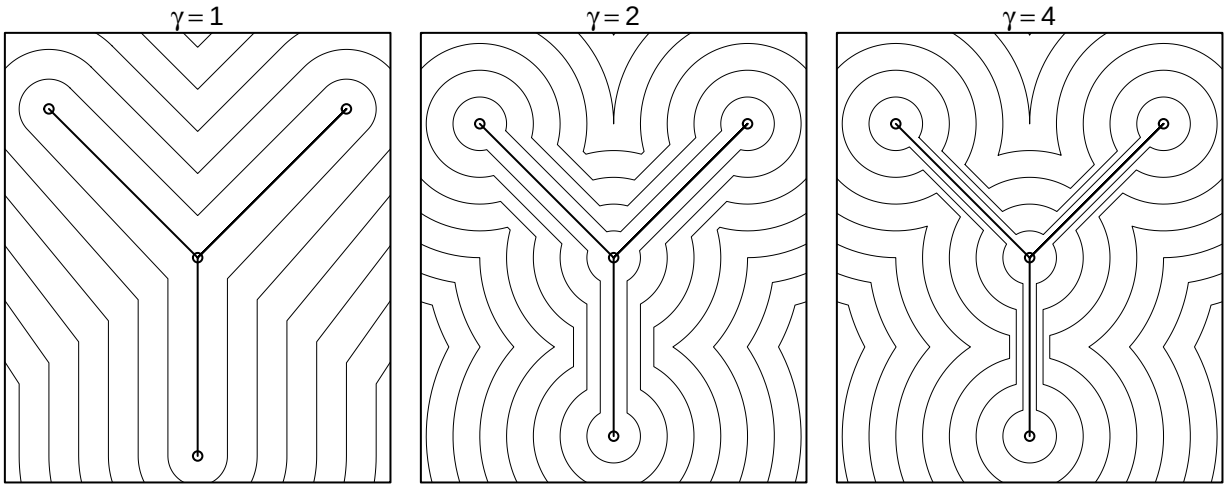

Figure 10: Contour lines for different values of γ

While this change may not be too dramatic when compared to introducing the beta-skeleton, it still gives us another lever to influence the way the phases will look at the end.

5 Random sets

Since the simulation starts off with three random sets of points, the resulting structure, or rather each of the simulated phases, can be considered a random set, as it fills some amount of space inside the cell which is completely dependent on a random process. Since this description of a random set is very handwavy, we next consider a more rigorous definition, which can be found in [7]. There one can also find the definitions and methods of computation for the following sub-chapters.

There are many ways to define a random set. Usually it is done by working with a system of test sets and then analysing their intersections with the random set.

Instead of considering how large these intersections are, one is usually only interested in whether it is empty or not. This form of measurement is closely related to the so called hitting σ-algebra $\mathcal{F}$.

A random closed set Ξ, or simply random set as we will call it, is a random variable and thus its domain has to be a measurable space, which in this case is given by $(\mathbb{F}, \mathcal{F})$. Here $\mathbb{F}$ is the family of all closed subsets of $\mathbb{R}^d$ and $\mathcal{F}$ is the smallest σ-algebra of subsets of $\mathbb{F}$ containing all hitting sets. A hitting set is defined as

$$\mathbb{F}_K = \{F \in \mathbb{F} : F \cap K \neq \emptyset\} \quad \text{for} \quad K \in \mathbb{K},$$

where $\mathbb{K}$ is the system of the before mentioned test sets, which in this case are all compact subsets of $\mathbb{R}^d$. To summarize, a random set Ξ is a random variable $\Xi : \Omega \to \mathbb{F}$ where $(\Omega, \mathcal{A}, \mathbb{P})$ is a probability space and $(\mathbb{F}, \mathcal{F})$ is the target measurable space. One can even define a distribution P on $(\mathbb{F}, \mathcal{F})$, which is called the distribution of Ξ, where $P(A) = \mathbb{P}(\Xi \in A)$ for each $A \in \mathcal{F}$.

A closed set Ξ is called stationary if Ξ and all translated sets $\Xi + x$ for $x \in \mathbb{R}^d$ have the same distribution. Similarly, Ξ is called isotropic if the distribution of Ξ is invariant under rotations around the origin and it is called motion invariant if it is both stationary and isotropic.

6 Characteristics

The stochastic model which we will describe depends on several parameters, that tell us how to build up the simulated cells. While we can choose these parameters almost freely, we want to pick them in such a way, that the simulated cells behave statistically similar to their real counterparts. Thus, in order to quantify the discrepancy between model realisations and image data, we require meaningful descriptors. We call these descriptors microstructure characteristics. In the following we present their definitions and methods for estimating them based on 3D image data.

Since the structure is considered as the union of three random sets, the microstructure characteristics are introduced as properties of random closed sets here. However, because we discretized the random sets into voxel grids, we not only want to know how we could compute these different properties in continuous space, but also how we can extract them from discretized data.

As for the characteristics themselves, most of them give some information as to how efficient the transport through the different phases can be. For example, the characteristics introduced in Sections 6.1 and 6.2 give some measure of size for the phases and the one introduced in Section 6.3 describes the length of the shortest paths through the phases. The actual transport properties like conductivity of the two metal phases can then be computed by numerical methods.

6.1 Volume fraction

The first of these characteristics is also the simplest one to compute. It is called the volume fraction and for a stationary random set Ξ it is defined as the mean volume of the union of Ξ and the unit cube $C = [0,1]^d$, so

$$p = \mathbb{E}(\nu_d(\Xi \cap C)),$$

where ν_d is again the d-dimensional Lebesgue measure.

It is one of the most important characteristics of a random set and can be compared to the intensity λ of a point process, as both describe how much of each we can expect per unit volume. In contrast to the intensity, the volume fraction can clearly not be larger than 1, as that is the measure of the unit cube.

For stationary Ξ it holds that

$$p = \mathbb{P}(o \in \Xi) = \mathbb{P}(x \in \Xi), \tag{7}$$

for each $x \in \mathbb{R}^d$, where o is the origin. This follows from

$$\begin{aligned} p = \mathbb{E}(\nu_d(\Xi \cap C)) &= \mathbb{E}\left(\int_{\mathbb{R}^d} \mathbb{1}_{\Xi \cap C}(x)dx\right) = \mathbb{E}\left(\int_C \mathbb{1}_\Xi(x)dx\right) = \int_C \mathbb{E}(\mathbb{1}_\Xi(x))dx \\ &= \int_C \mathbb{P}(x \in \Xi)dx = \int_C \mathbb{P}(o \in \Xi)dx = \mathbb{P}(o \in \Xi)\int_C dx = \mathbb{P}(o \in \Xi) = \mathbb{P}(x \in \Xi) \end{aligned}$$

for each $x \in \mathbb{R}^d$.

All steps where we replace the integral variable x by the origin follow from the stationarity of Ξ. Integrating over the unit cube gives us a factor of 1, which is why we can ignore the integral in the last step.

When simulating the energy cell we are not given the true value of p for any of the phases. We could try to calculate the exact values of our simulated cell, since in theory we know all vertices and edges of the beta-skeletons which completely determine the phase. Since we only look at part of the cell, namely the part inside the observation window W, this would result in the empirical volume fraction

$$\hat{p}_V = \frac{\nu_d(\Xi \cap W)}{\nu_d(W)}.$$

In practise, however, this is not a feasible option, since this would be quite computationally costly.

However, this computation can be made much faster. Instead of accurately representing Ξ we instead use grid points which are evenly distributed inside of W. This means that instead of computing the volume fractions inside the window exactly, we simply count the points belonging to each phase and compute their relative amount. Moreover, for 3D image data, we have only information on the voxel grid. This gives us the estimators

$$\hat{p}_i = \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{\Xi_i}(x_k) \quad \text{for} \quad i = 1, 2, 3,$$

where Ξ_i is the set corresponding to phase i and $x_1, .., x_n$ are the grid points inside W.

This estimator is unbiased since

$$\mathbb{E}(\hat{p}) = \frac{1}{n}\sum_{k=1}^{n} \mathbb{E}(\mathbb{1}_{\Xi}(x_k)) = \frac{1}{n}\sum_{k=1}^{n} \mathbb{P}(x_k \in \Xi) = \frac{1}{n} np = p.$$

We use stationarity of the random set to replace $\mathbb{P}(x_k \in \Xi)$ by p for all k. Since for our energy cell the location of the Ξ_i is entirely dependent on the stationary point processes which were used to construct them, the corresponding closed random sets themselves are also stationary.

Under certain conditions on the random set, which can be found on p.231 of [7], we can approximate the variance of this estimator by

$$\sigma_p^2 \approx \frac{p(1-p)}{n}.$$

The variance therefore gets smaller as n gets larger. This is consistent with what we would expect, since this implies that using a finer grid, which better approximates the actual phase, would give us a better estimator.

Alternatively, one could use a larger window while keeping the distance between the grid points the same to get a larger n. In our example this has the benefit of sampling points which are less likely to be correlated, since points which are closer to each other are more likely to be from the same phase simply because of their proximity.

6.2 Contact distribution function

Next we look at another important characteristic, the contact distribution function, which can be considered as some sort of size measurement for the random set. In general it is defined using a Borel set B with positive measure and the property $r_1 B \subset r_2 B$ for all $0 \leq r_1 < r_2$. Then, for a stationary random set Ξ with volume fraction $0 < p < 1$, the contact distribution function is given by

$$H_B(r) = 1 - \mathbb{P}(o \notin \Xi \oplus r\check{B} | o \notin \Xi), \tag{8}$$

or equivalently

$$H_B(r) = 1 - \frac{\mathbb{P}(o \notin \Xi \oplus r\check{B})}{1-p}, \tag{9}$$

for $r \geq 0$. Here $\check{B}$ is the reflection of B defined as

$$\check{B} = -B = \{-x : x \in B\},$$

and $\Xi \oplus r\check{B}$ is the Minkowski sum of Ξ and $r\check{B}$ given by

$$\Xi \oplus r\check{B} = \{x + y : x \in \Xi, y \in r\check{B}\}.$$

The equality of the two definitions follows from the definition of conditional probability, since

$$\mathbb{P}(o \notin \Xi \oplus r\check{B} | o \notin \Xi) = \frac{\mathbb{P}((o \notin \Xi \oplus r\check{B}) \cap (o \notin \Xi))}{\mathbb{P}(o \notin \Xi)}.$$

The way B is defined implies that $o \in B$ and therefore also $o \in \check{B}$. It thus holds that $\Xi \subset \Xi \oplus r\check{B}$ which means that $o \in \Xi$ implies $o \in \Xi \oplus r\check{B}$. By logical negation we get the implication

$$o \notin \Xi \oplus r\check{B} \implies o \notin \Xi,$$

which simplifies the numerator to the required probability $\mathbb{P}(o \notin \Xi \oplus r\check{B})$. For the denominator we simply use Equation (7), which implies

$$\mathbb{P}(o \notin \Xi) = 1 - \mathbb{P}(o \in \Xi) = 1 - p,$$

and thus shows that both definitions are equal.

For compact and convex B, which contains the origin as an inner point, $H_B(r)$ is indeed a distribution function as its name would suggest. In this case even a probability density function $h_B(r) = H_B'(r)$ exists.

Rewriting the first definition by using the stationarity of Ξ gives us

$$H_B(r) = \mathbb{P}(o \in \Xi \oplus r\check{B} | o \notin \Xi) = \mathbb{P}(x \in \Xi \oplus r\check{B} | x \notin \Xi),$$

and with it a better intuition for what the contact distribution function actually implies. It is the probability for a given point outside the random set to be within a sort of B-skewed range r of the random set.

This can be imagined more easily by using the special case where B is the unit sphere $S = B(o, 1)$, with the resulting $H_S(r)$ being called the spherical contact distribution function. In this case we get that

$$\begin{aligned} x \in \Xi \oplus rS &\iff \exists \xi \in \Xi, \exists v \in S : x = \xi + rv \\ &\iff \exists \xi \in \Xi, \exists v \in S : x + rv = \xi \\ &\iff \exists \xi \in \Xi : \xi \in B(x, r) \\ &\iff \Xi \cap B(x, r) \neq \emptyset. \end{aligned}$$

The second equality follows from the fact that $S = \check{S}$, meaning that for each $v \in S$ we also have $-v \in S$. With this we can interpret $H_S(r)$ as the probability that the distance of any point outside Ξ to its closest neighbour within the set is at most r. We can thus interpret $1 - H_S(r)$ as the conditional probability that a ball with radius r and center $x \notin \Xi$ lies completely outside the random set. For this reason $H_S(r)$ is sometimes referred to as "the law of first contact". Similarly $H_S(r)$ can be viewed as the distribution function of the Euclidean distance from the origin to Ξ, i.e

$$H_S(r) = \mathbb{P}(d(o, \Xi) \leq r | o \notin \Xi),$$

for each $r \geq 0$. Next we want to find an estimator for the spherical contact distribution function. For this we transform Equation (9) into

$$\mathbb{P}(o \notin \Xi \oplus r\check{B}) = (1 - H_B(r))(1 - p).$$

The left side is the definition of $(1 - p_{\Xi \oplus r\check{B}})$, which means we can use the estimators we just obtained for the volume fraction. This results in the estimator

$$\begin{aligned} \hat{H}_B(r) &= 1 - \frac{1}{1 - \hat{p}}(1 - \hat{p}_{\Xi \oplus r\check{B}}) \\ &= 1 - \frac{1}{1 - \hat{p}}(1 - \frac{\nu_d(W \cap (\Xi \oplus r\check{B}))}{\nu_d(W)}). \end{aligned}$$

In the next step we want to apply minus-sampling, which means that instead of considering all $x \in W$ we only consider those for which $x \oplus r\check{B} \subset W$. This is done to correct any potential biases which may arise when any of the theoretical objects $x \oplus r\check{B}$ stick out of the observation window W.

We can rewrite the condition $x \oplus r\check{B} \subset W$ as $x \in W \ominus r\check{B}$ by using Minkowski-subtraction, which is defined as

$$A \ominus B = \bigcap_{y \in B} A \oplus \{y\}.$$

This results in the minus-sampling estimator

$$\hat{H}_B(r) = 1 - \frac{1}{1-\hat{p}} \left(1 - \frac{\nu_d((W \ominus r\check{B}) \cap (\Xi \oplus r\check{B}))}{\nu_d(W \ominus r\check{B})}\right) \quad \text{for} \quad r \geq 0.$$

Since for the spherical contact distribution we get $\check{S} = S = B(o,1)$, the estimator simplifies to

$$\hat{H}_S(r) = 1 - \frac{1}{1-\hat{p}} \left(1 - \frac{\nu_d((W \ominus B(o,r)) \cap (\Xi \oplus B(o,r)))}{\nu_d(W \ominus B(o,r))}\right) \quad \text{for} \quad r \geq 0.$$

The numerator of the large fraction can be written as an integral using indicator functions, which gives us the term

$$\int_W \mathbb{1}_{W \ominus B(o,r)}(x) \mathbb{1}_{\Xi \oplus B(o,r)}(x) dx.$$

If $\mathbb{1}_{\Xi \oplus B(o,r)}(x) = 1$, then x is at most distance r from Ξ. For this reason we can replace the indicator function by $\mathbb{1}_{d(x) \leq r}(x)$, where $d(x)$ denotes the Euclidean distance from x to Ξ. The resulting estimator thus includes an integral, which we will approximate by an appropriate sum. Since our window is a cuboid, this can be done by splitting W into equally sized cubes C_{x_k} around each grid point x_k. We then only evaluate the indicator function at each grid point and assume the same value for the rest of each small cube. By then summing up over all these values, which we also multiply by the respective volume of each cube, we get a decent approximation of the integral.

The final estimator we got and its approximation, which we will use, are thus given by

$$\begin{aligned}\hat{H}_S(r) &= 1 - \frac{1}{1-\hat{p}} \left(1 - \int_W \frac{\mathbb{1}_{W \ominus B(o,r)}(x) \mathbb{1}_{d(x) \leq r}(x)}{\nu_d(W \ominus B(o,r))} dx\right) \\ &\approx 1 - \frac{1}{1-\hat{p}} \left(1 - \sum_{k=1}^{n} \frac{\mathbb{1}_{W \ominus B(o,r)}(x_k) \mathbb{1}_{d(x_k) \leq r}(x_k) \nu_d(C_{x_k})}{\nu_d(W \ominus B(o,r))}\right) \quad \text{for} \quad r \geq 0.\end{aligned}$$

6.3 Mean geodesic tortuosity

Next, we consider another important characteristic, which can tell us a lot about how efficient the different phases are at transporting. The mean geodesic tortuosity τ_W is defined on the observation window W and is the expected length of a path through $\Xi \cap W$ in transport direction divided by the window thickness in transport direction. In this case the transport direction is defined as starting at some base for the pores and the YSZ-phase, which supplies them with hydrogen and oxygen-ions respectively, and ending at a collector for the generated electricity for the copper-phase. For a more rigorous definition of the mean geodesic tortuosity of a random set, refer to [10].

The minimum value of the tortuosity is thus 1, since that would equate to a path going straight through the material. Since all phases are completely connected, finding such a path is also always possible, given a sufficiently large window. It could of course happen, that for a very small window the entire starting or end layer does not contain any of the wanted phase or that a path through the phase does exist, but it leaves the window at some point. However, for the sizes one usually uses, this is not a concern.

Estimating this characteristic from the simulation or discrete image data means that we have to find some shortest path through the voxel grid. There are several ways this can be done and one of them is using Dijkstra's shortest paths algorithm. One can simply imagine the voxel grid as a graph, where we have an edge between two neighbouring voxels, if the share a side or possibly only an edge if one wants to allow diagonal movement. In our case we only assume edges between voxels if they share a side, giving us a very bumpy

path through the phase. However, this is remedied by smoothing the path afterwards, before measuring its length. We do this by starting at the first voxel of the path and connect it to the second one. This line segment will of course always be entirely contained within the current phase, since both voxels are part of it. We then instead try to connect the first voxel to the third one and check if the line segment is also contained within the wanted phase. We repeat this until some line segment is no longer part of the phase. At that point we mark the previous voxel, whose line segment was still contained in the phase, as the second voxel in the smoothed path. We then repeat this process with this marked voxel as the new starting point for the line segments, again marking the farthest reachable voxel as the next one for the smoothed path. Once we reach the last point of the original shortest path we are done, at which point we can measure the length of the smoothed path, to calculate the mean geodesic tortuosity.

We have some options for how we want to measure the shortest path, which will all result in slightly different lengths. As mentioned before, we could allow for diagonal movement in Dijkstra's algorithm, which can possibly give us a different shortest path. Next we have some leeway in how we define the smoothing of the path. For example, if some voxel is not part of the current phase, but two of its neighbours are, then it is likely that some part of the voxel would belong to the current phase, if we did not discretise the random set. It is therefore not too far-fetched to allow the smoothed path to also use this voxel.

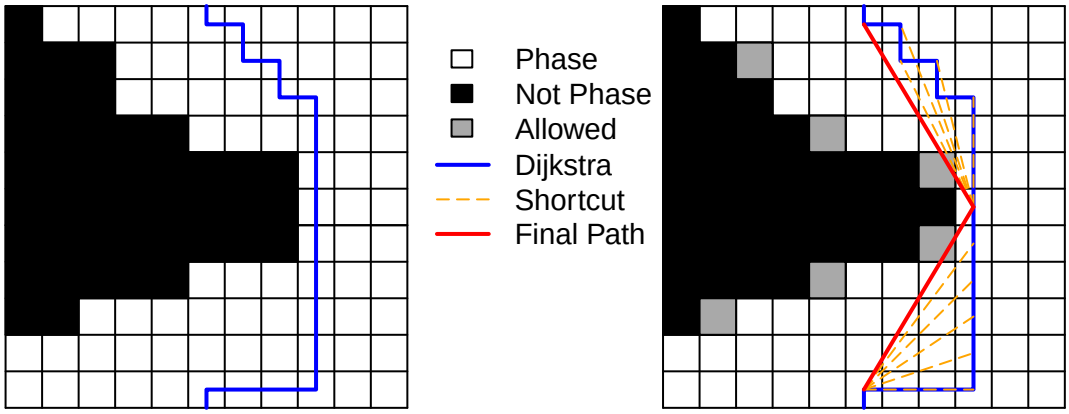

Figure 11: Path Smoothing in 2D

6.4 Specific length of Triple Phase Boundary

Since energy can only be generated at points where all three phases meet, the length of their boundary is a great indicator for the fuel cell's efficiency.

Computing the exact length is possible, since we know all vertices and edges of the beta-skeletons. However, this is not exactly an easy task, which is why we will again resort to using the voxel grid and estimate the length.

We use a pretty straightforward estimation, which counts all sets of 8 voxels which share a corner and contain at least one voxel from each of the three phases. This estimator can be computed quite easily, but is biased as the spatial arrangement of the triple phase boundary voxels is not considered [4]. Nevertheless, it still serves as qualitative information for the length of the triple phase boundary. Lastly we divide the resulting length by the volume of the observation window, as we are interested in the length of the boundary with respect to the size of the observed material. This adjusted length is then called the specific length δ of the triple phase boundary.

7 Implementing the Simulation

Implementing the point processes can be done exactly as described in Section 3.5, without any need for optimization. In principle, one could also implement beta-skeletons and the voxel grid without much trouble, since both can be done by comparing distances to all points. While this can work for small examples, as the number of points and voxels get larger this becomes less feasible. To compute a beta-skeleton, we would have to compare each pair of points, which there are quadratically many of, with each of the remaining points. In total this would result in a runtime of $\mathcal{O}(n^3)$, where n is the number of vertices, i.e. the number of points in the realization of the corresponding Poisson point process. For the voxel grid, we are interested in the distances $d'_{\gamma_i}(x, B_i)$ from a given voxel mid-point x to each of the beta-skeletons B_i, which were defined in Section 4.4. Doing this in the same way as before entails a distance calculation from each voxel to every vertex and edge in each of the skeletons. This results in a runtime of $\mathcal{O}(v(n_1 + n_2 + n_3 + m_1 + m_2 + m_3))$ which simplifies to $\mathcal{O}(v \max_{i=1,2,3}(n_i, m_i))$, with v being the number of voxels to assign to the phases and n_i and m_i being the number of vertices and edges of each respective skeleton. While this computation may seem faster, as it is in a sense only quadratic in its variables, we have to consider that the number of voxels is usually a lot larger than the number of points. Since we assume connected beta-skeletons, the number of edges will usually also be larger than the number of points.

7.1 R-trees

With all this in mind, it seems like we need a faster way of finding the nearest neighbours. A first step is to find an appropriate data structure, which stores our spatial data in a such a way, that objects which are near each other in space are also close in the structure in some way. For 1-dimensional data this is an easy task, as a simple list, which stores the data in ascending or descending order, would already do a good job at this. But in two dimensions it is already not that easy. This is where R-trees come in, which were first described in [11].

As the name suggests, R-trees use the general tree data structure as a basis, with the root node at the highest level and leaves at the very bottom. They are great at indexing multidimensional data, which they do by assigning a rectangle to each node. These rectangles are also the other namesake of the R-tree. The basic idea is to divide all objects into groups based on their proximity to each other and then represent each group by its minimum bounding rectangle, which are then stored in the next higher level of the tree. A minimum bounding rectangle is the smallest rectangle with edges parallel to the axes, which contains all points in a group. For simplicity we will also call it a bounding rectangle for higher dimensions, even if bounding cuboid would be a more accurate name.

The points are stored in the leaf vertices of the tree. Each leaf contains the spatial information of the corresponding point, so its coordinates, and also its index for later identification, which is usually simply an integer between 1 and the number of objects we work with. In theory we can store any spatial object inside the leaves, but for more complex objects its easier to only keep the information of the corresponding minimum bounding rectangle. Since our information consists not only of points but also edges, we could use this method of storing the latter. This however is not necessary, as storing the edge by using its edge points uses the same amount of space as a bounding rectangle would, while also containing more information. In addition, this loss of information would result in less accurate distance calculations or, if we wanted to calculate it correctly by referring from the rectangle to the corresponding edge, more computation steps. One should note, that some sources define the tree a bit differently, as they instead call the layer above the spatial objects the leaf layer, with all the spatial objects not actually getting their own node in the tree.

As mentioned before, the nodes which are higher up in the tree contain spatial information corresponding to a bounding rectangle, which contains all the spatial objects of its children. The inner nodes are also indexed similarly to the leaves and in addition to this also contain the index information of their children. Inner nodes are grouped in a similar way to the leaves. This means that they are put in the same group if they are close to each other and also that no nodes of the layer are ignored. At the top of the tree is its root, whose bounding rectangle contains every spatial object we started with, as well as all bounding rectangles of the inner nodes.

One could of course start the tree building process by placing all leaves in one group and therefore be done. Since this is of course not what the structure is meant for, we introduce additional parameters with which we can shape the tree. The first is the capacity M, giving us an upper limit on the number of children per inner node. The other one is $m \leq M/2$, which conversely gives us the minimum number of entries each inner node should contain. The only non-leaf node which is not bound by this is the root, as building the tree might not be possible otherwise. The only other restriction we have for the root is that it should contain at least two children, unless its children are already leaves. By building the tree as described, it will always be balanced, meaning that all leaves appear on the same level in the tree.

The height of the tree which stores N spatial objects is at most $|\log_m n| - 1$ while the maximum number of non-leaf nodes is $\lceil N/m \rceil + \lceil N/m^2 \rceil + \ldots + 1$. Usually nodes have more than m children, which decreases the height of the tree and as a consequence also the time needed to navigate it. Choosing a larger value for M results in a tree that is very wide, which means that most of the nodes are used for storing the primary spatial information in the leaves instead of the bounding rectangles. However if M is chosen too large it can negatively impact the R-tree's ability to help in finding the nearest neighbour much faster than a simple list can.

7.1.1 Recursive Building

There are two main methods one can use to build an R-tree. The first method involves building up the tree step by step, inserting new spatial objects at the leaf level, which was first described in the original paper on R-trees [11]. To ensure that the resulting R-tree conforms to our given parameters this has to be done with a lot of care.

The first step is to choose an appropriate parent to which the leaf is attached. We start at the root and check whether it is at the lowest non-leaf level. If it is, it is chosen as the parent of our new object, otherwise we look at its children. Since these children are not at the leaf level, they contain the information of the bounding rectangle of their respective children. We then choose the bounding rectangle which needs the least enlargement to fit the new object. This step is repeated until the current parent is located at the lowest non-leaf level.

If the chosen parent L still has enough space to fit the extra element, i.e. if it currently has less than M children, then we can simply add the new element to it. If needed, we then fix the bounding rectangles of the node's new parent and all its other predecessors and continue with inserting the next object.

Should the new element not fit however, then we have to be more careful. We start fixing the tree by first splitting the parent node L into L_1 and L_2. For this reason, the minimum number of children is $m \leq M/2$, since otherwise one of the new nodes would be below this threshold. The main idea is to split the set of children in such a way, that the resulting bounding rectangles are as small as possible, so similarly to how one chooses the parent of a new element. Splitting this way has the possibility of overlapping the bounding rectangles. This can increase the time it takes to find the nearest neighbour, as it is possible that before finding it in one of the rectangles, we might have already searched through a lot of the other one. While this can also happen with non overlapping bounding rectangles, it is less likely.

Choosing this split is not trivial, as finding the best one by complete enumeration implies checking all possible groupings into 2 nodes. The total number of these groupings is exponential in M, even for values of m close to $M/2$. There are however fast algorithms which, while not guaranteed to find the split with the smallest area possible, can find a reasonably good split.

The Quadratic Split algorithm described by Guttman runs in $\mathcal{O}(M^2 d)$ time, where d is the dimension of the objects, i.e. $d = 3$ in our case. It starts with the two objects whose bounding rectangle wastes the most amount of space, which are then put into two different groups. We define wasted space as the d-dimensional volume of the larger bounding rectangle minus the respective volumes of the smaller rectangles. For point data, the bounding rectangle of a single point has volume 0 and thus the wasted space is simply the volume of the larger bounding rectangle. In the next step, we are interested in minimizing the smaller bounding rectangles, so for each of the remaining points we calculate the additional volume needed to fit them into either of the rectangles. Again we choose the remaining points and rectangle which needs the least expansion

and add it to the corresponding group. This process is repeated until there are no more points remaining or until one of the groups needs all the remaining points to get above the threshold m.

We are now left with the two groups L_1 and L_2 with sizes between m and M. If L_2 fits into L's previous parent we simply add it and again fix the bounding rectangles as in the case without a split. However, if there is not enough space, we again have to split the parent node in a similar fashion to before. This split can propagate all the way to the root and we have to fix the bounding rectangles along the way. The only case we have to treat differently is the one where we have to split the root. At this point we have to introduce an extra layer above the current root in which the new root node is placed. The new root then has 2 children, namely the two nodes resulting from the split of the old root. Since the root is the only node which is not bound by m, it does not matter that it only has 2 children. However, since its children are no longer root nodes, they have to adhere to the lower bound, meaning this split is treated like all the others in this regard. We can repeatedly insert all the spatial objects until the tree's leaves contain all of them.

Deleting can be done in a similar manner, but instead of splitting when a root contains too many entries, we instead eliminate nodes if they do not contain enough, i.e less than m. This eliminating can propagate up the tree in a similar manner to the split, forcing us to also remove nodes which are higher up the tree. The root again requires a special treatment, since while it is not bound by m, it still becomes obsolete if it only has 1 child. At this point its child becomes the new root and the old one is removed. All removed nodes then have to be reinstalled by using the insertion process as described above, with the slight change that non-leaf nodes are placed at the correct height in the tree, together with all their children.

This method has the advantage of being able to stop at any time, since the resulting R-tree after every insertion always conforms to all the given parameters. It is also a lot more flexible, since we are always able to add or remove objects even after we are done building the tree. However, this flexibility comes at a cost, as building a tree this way is a lot more involved than the other method we will introduce.

7.1.2 Bulk-loading algorithms

The second method adds all the nodes at once instead, which is why it is called bulk-loading. There are different ways to bulk-load an R-tree. However, all of them first decide into how many groups the nodes of the current layer are split into, depending on their amount and the lower and upper bound on node sizes. The only difference is how one decides to group them up. The most commonly mentioned bulk-load methods, which also include the one we will be using, are Nearest-X, Hilbert Sort and Sort-Tile-Recursive. In all these methods one has to order the rectangles with respect to a single point, which is usually either assumed to be the center of the rectangle or its low or high points in some given direction. These methods as well as more information on them can be found in [12] or [13].

Nearest-X is the simplest of them, using only the x-coordinate of the rectangles for the purpose of dividing them into groups. This means that our space is cut into slices along the x-axis and objects whose x-values are close to each other will usually land inside the same group. The slices are done in such a way, that each slice contains between m and M objects. This can work well, if the data is spread across the x-axis a lot more than the other axes, since these slices are more likely to contain points which are also close to each other with respect to the other axes. If the data is spaced out equally in all directions, then this method can lead to points being in the same group while actually being quite far apart. Therefore, while easy to use, it is certainly not a method which should always be applied.

Another approach is called Hilbert Pack or Hilbert Sort, which uses a space filling Hilbert curve to order the rectangles. The idea is to put rectangles in the same group, if they appear on the Hilbert curve in succession. In the original paper on Hilbert packing by Kamel and Faloutsos additional packing algorithms were described, which were based on different space filling curves like the so-called Z-order or Gray-code curve. However, the Hilbert curve achieves the best clustering among these curves, as was shown by Faloutsos and Roseman in [14]. The algorithm was first only described with integer coordinates in mind, but can also be extended to floating point numbers. This can be achieved by converting the float into an integer in such a way, that the position relative to the other points stays the same, as is described in [12].

The last of these bulk-load algorithms we will visit is also the one which was used for the simulation. The

Sort-Tile-Recursive algorithm is in a sense simply a more sophisticated version of Nearest-X, since it works in a similar way, but also takes the other dimensions into account. This is done by first cutting the space along the x-axis as we did before, but this time each slice contains more objects. The reason for this is, that we then cut each slice again, but this time along the y-axis. This is repeated once for each dimension of our data.

The exact amount of slices is dependent on the lower and upper limits m and M, the amount of objects or rectangles in our current layer of the R-tree n and the number of dimensions of our data d. Since we want each node to have roughly the same amount of children, we separate our objects into n/M groups. Ideally we are allowed to separate the objects along each direction into $\sqrt[d]{n/M}$ groups. However, since both terms are usually not integer values, we have to instead use the ceiling function to convert them into such. Most of the time this leads to some groups being smaller than others in each step, which can cause some nodes to be filled with less than m objects. To remedy this, one does the first separation into $s_1 = \left\lceil \sqrt[d]{n/M} \right\rceil$ groups as before. But in the second step, we calculate a more accurate number of objects left in each group using $n_1 = \lceil n/s_1 \rceil$, which results in the amount of groups along the y-axis $s_2 = \left\lceil \sqrt[d-1]{n_1/M} \right\rceil$. The general forms of these numbers are

$$n_i = \left\lceil \frac{n}{s_1 \dots s_i} \right\rceil \quad \text{and} \quad s_i = \left\lceil \sqrt[d-i]{\frac{n_{i-1}}{M}} \right\rceil .$$

When splitting the layers in this way, the largest group contains only 1 element more than the smallest. One might at first assume that these differences of 1 can pile up to a larger one, as they happen in each layer. This is in fact not a problem, as each new layer means, that the larger and smaller groups from the previous layer are all treated as a single object regardless of size.

The only possible issue now is, that the larger groups might not contain M elements. This means that the smaller groups contain 1 fewer element, which might then be fewer than m. After testing this for a lot of different values of n and M, it turns out that this amount can very rarely go below $m = M/2$. In practice one could simply ignore this, as the amount is only ever 1 less than m and as mentioned, this is also not something that occurs very often. In addition to this, when the Sort-Tile-Recursive algorithm was introduced, there was no mention of a lower bound on the amount of elements in a node for any of the bulk-loading algorithms. For this reason we will not consider m as a hard cap on the minimum amount of elements, but rather a soft cap, which only takes hold when checking whether we can simply split all dimensions into groups of size $\sqrt[d]{n/M}$. We thus only use m to ensure that only a few nodes are sometime under-full, to keep the searching algorithm reasonably fast.

To give a visual representation of the three different variants, we apply them all to the same simple set of two-dimensional integer points. This can be seen in Figure 12.

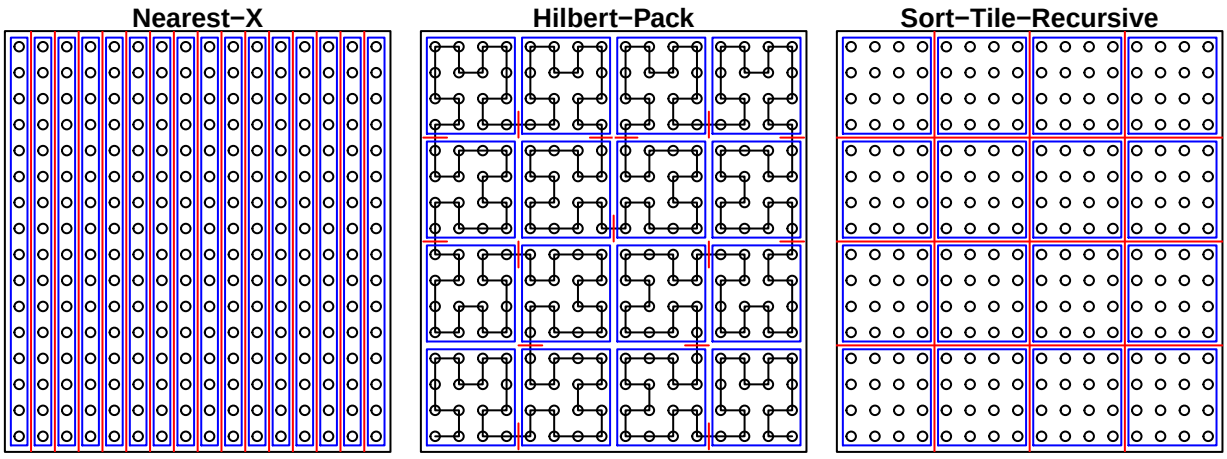

Figure 12: Different bulk-loading algorithms for n=256, M=16

In the example, we can clearly see the disadvantage of the Nearest-X algorithm, as a lot of points which are close to each other are placed in different nodes. While this is something that will always happen for any

such process, as some cuts always have to be made, it is still a lot more pronounced for this method. We also have a nice visualization of how the Hilbert-Pack functions, as one only needs to follow the curve, which in this case starts in the bottom left, and place the cuts whenever we passed the right amount of points. The amount of points can either just be M for all groups except possibly the last one, in case one does not care about the lower bound m, or can be be calculated in a similar manner to the Sort-Tile-Recursive.

The latter two algorithms actually result in the same tree structure for this particular example, as it is a quite simple one. We can however see, that the Sort-Tile-Recursive algorithm has the advantage of being a lot simpler to execute, as no complicated space filling curve is needed.

Next we want to observe how the Sort-Tile-Recursive algorithm acts on a bigger and not quite as simple set of points. For this, we generate a Poisson point process in two dimensions, as it is easier to visualize than the three dimensional case.

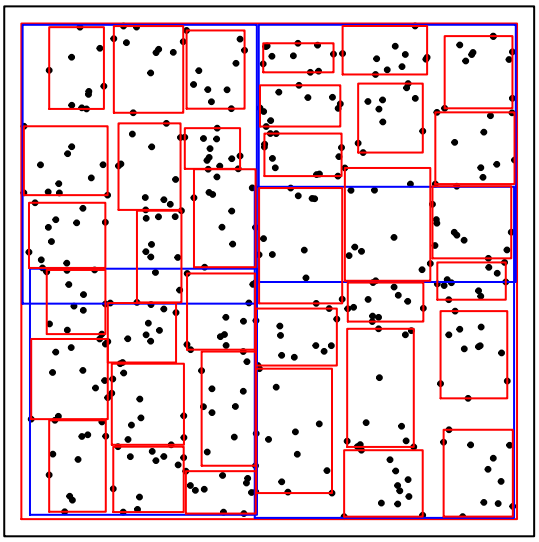

Figure 13: Sort-Tile-Recursive algorithm applied to Poisson point process

The small red boxes form the lowest non-leaf layer of the tree. One can clearly see, that none of them overlap, which is not only highly desirable, as it speeds up the searching process, but also expected when working with point data. That this is not necessarily valid for non-point data can be observed with the blue rectangles. These form the next highest layer of the tree, which together contain all of the smaller red boxes. They are also clearly overlapping in some areas, which is to be expected. However, despite this overlap no red box is ever fully contained within two blue ones. Another thing to notice about the overlap is, that it only ever occurs in the y-axis. The reason for this in this example is simply the order in which the red boxes were grouped. First we sorted the point data along the x-axis, meaning that the lowest x-value of any point in group $k+1$ is always larger than the highest x-value of any point in group k. The same can however not be said about the y-axis, as these were formed within each of the groups, completely independent of all other groups. Also relevant is the fact that in this case the x-axis was split into 6 groups. As it is an even number, the next layer could split it cleanly into 2 parts. This does not always happen, but certainly makes for a better tree structure. Lastly we have the large red box, corresponding to the root of the R-tree. It contains all blue rectangles and therefore also all smaller red ones and all the data points.

There are many advantages to building the tree using the Sort-Tile-Recursive algorithm. The first is a faster run time, as we do not have to worry about placing the vertices individually. While we have to invest extra time at the beginning to group the objects and their parents, this is easily made up by the much faster insertion of the nodes, as fixing the tree is no longer a concern.

Another advantage is, that each inner node has almost the same amount of children as the other nodes at its

level. This is not guaranteed with the other method, as it might have nodes which are almost completely full and others which only contain m elements. Having this more even distribution of children has the effect of averaging out the time it takes to find a nearest neighbour, as nodes with more children usually take more time to be searched than ones with fewer.

An added benefit is that the bounding rectangles will have minimal overlap, which also means a faster runtime for the search algorithm. As mentioned before, this is not a given for the other method.

Implementing this algorithm is also easier, since inserting or deleting elements is no longer needed. Because of this, we can also manage with a simpler underlying data structure, like an array. While basing everything on a tree should be faster in theory, it can also be a lot slower if the tree is not properly implemented. As arrays are a quite basic data structure, they are usually highly optimised for a lot of programming languages and thus do not face the same problem. The only other caveat to using arrays, besides being slower in theory, is the loss of flexibility. While we do not need insertion or deletion for building the tree, one might still prefer to have the option. Using an array for this is not advisable, as inserting additional nodes also means inserting a row into the array. While this can be done, it should be much faster to use a tree in this case.

7.2 Nearest Neighbour Algorithm

With our new data structure we can now look at how we best search for the nearest neighbour within it. This algorithm will have two different applications when discretizing the model for the morphology of solid oxide fuel cell anodes, which were introduced in Section 2. It is first used to speed up the process of building the beta-skeleton. While it would be possible to also do this by complete enumeration, it is much faster to calculate the nearest neighbours of a given point and only use these to find the needed edges for the skeleton. The second use case is to find the nearest point or edge of any of the beta-skeletons, when assigning the grid points to the three phases.

The algorithm we will apply is due to Hjaltason and Samet [15], which they called Incremental Nearest Neighbour algorithm. It is called incremental, as it can be implemented in such a way, that one could stop at any time and continue searching for more points later. Since this is not needed for our case, we simply use it to calculate an amount of points which we determine in advance.

Beside the already mentioned R-tree, the algorithm also uses a priority queue. This data structure is similar to a normal queue, in the sense that the object at the front of the queue is always the one which is next being used. However, in contrast to a normal queue, new objects are not simply placed at the end of it, but instead are put in order of their priority key. This value in a way tells us how important each object is. Since our main goal is to find our nearest neighbour, the key we use is the distance from a given object or rectangle to our query object q. In this case a smaller distance equates to a higher priority. With the way the algorithm works, a node is only examined once it reaches the head of the priority queue. This can only happen once all nodes representing objects and bounding rectangles which are closer to the query object have already been dealt with.

We start the algorithm by initialising the priority queue with only the root of the R-tree. Then, in each step we consider the element at the front of the queue. If it is an inner node, we add its children to the priority queue. As mentioned before, the children are put there with respect to their distance to the query object. Should the node at the front be a leaf, meaning that it corresponds to a spatial object, we return it together with its distance to q. Depending on how many neighbours we are interested in, we either put the object in a list together with the next nearest neighbours or we simply output it, if only one neighbour is needed. This process can be repeated until we either returned the required amount of neighbours or until the queue is empty, since in this case we demanded more neighbours than there are objects stored in the tree.

The algorithm is entirely dependent on the fact, that the element at the head of the queue is always closer to q than all elements which are yet to be considered, but farther away than all elements which have already been processed.

To see that this is the case, we use the following clever visualisation of the algorithm from [15]:

1. Start at the query element $q \in \mathbb{R}^d$.
2. Our search region is a d-dimensional ball $B(q, r)$ with center q and expanding radius $r > 0$.
3. Whenever the expanding ball touches a rectangle, its children are put into the queue
4. Whenever the expanding ball touches an object, it is the next nearest one to q.

It should be mentioned that while the tree in the following example is a binary tree, this is not something which is required. However, the binary tree is chosen as an example since it is easy to visualize.

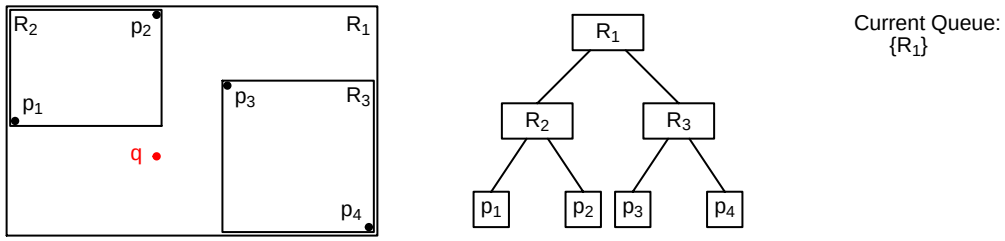

Figure 14: Visualization of nearest neighbour algorithm: Initialization

Before we start expanding the ball around q, we first initialize the queue with R_1. The reason for this is, that no matter if we start the search inside or outside the rectangle of the root node, it will always be the first object we will reach with this approach.

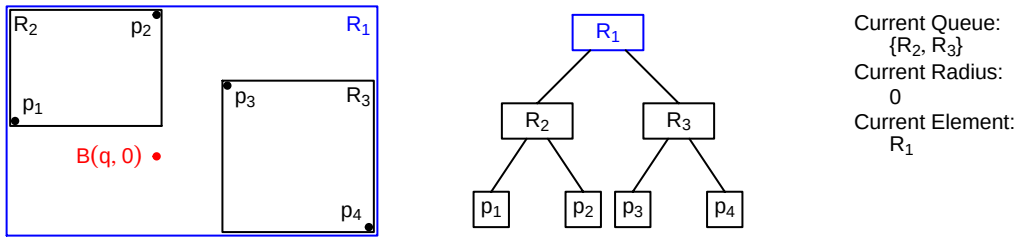

Figure 15: Visualization of nearest neighbour algorithm: Step 1

In the first step of the algorithm we start at the query object q. When trying to expand the ball around it, we immediately touch the rectangle R_1, since we start inside it. This means it gets removed from the queue and its children get added to it instead. If q was positioned outside of R_1, we would have to first expand the ball until we reach its border before working with it. If q was inside not only the root rectangle, but also one of its children, then we would still assume that we hit the root rectangle first, as we assume it to be the outer layer of the whole structure in a way.

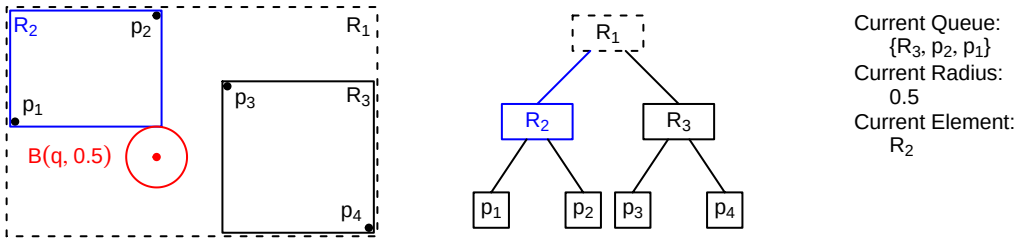

Figure 16: Visualization of nearest neighbour algorithm: Step 2

In the second step we actually touch a rectangle from the outside, which in this case is the rectangle R_2. As before, it gets removed from the queue and replaced by its children. With this method we don't necessarily

know in which order to put the children inside the queue, since finding out the distance is done by reaching something with the expanding ball. However in practise we would put p_2 before p_1, since the first one is closer, but both get put behind R_3.

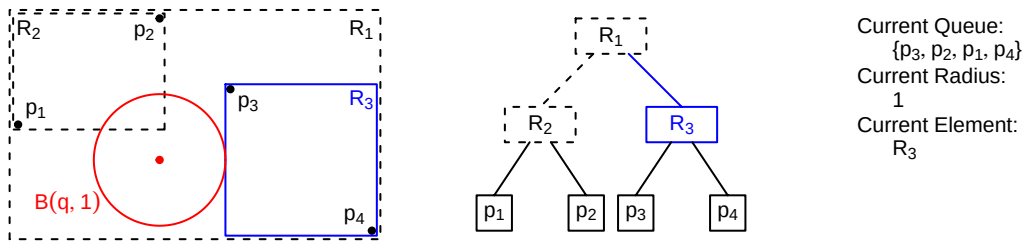

Figure 17: Visualization of nearest neighbour algorithm: Step 3

The third step is pretty much the same as the second step in terms of what we have to do next. The element which is reached is again not a data object, therefore we only replace it in the queue by its children. The main difference to the last step is, that the queue did contain data objects in the form of p_1 and p_2. However, these are not relevant for this step, as they were not at the front of the queue.

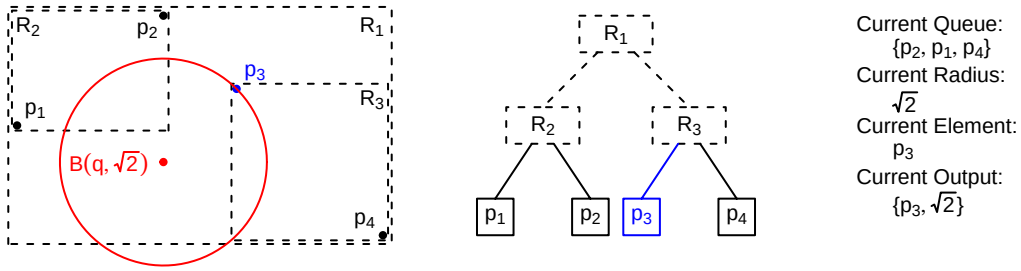

Figure 18: Visualization of nearest neighbour algorithm: Step 4

In the final step, which we will look at, the ball touches the object p_3. It thus gets removed from the queue and, since it has no children, no new entries get added. Instead we take the current radius of our ball, which is the object's distance from q, and return it, together with the object. Our nearest neighbour to q is thus the point p_3.

To find out which are the next nearest neighbours, one would simply have to expand the ball until reaching the other points, outputting them in the order in which they are reached by the ball.

From this example we can see that the claim we made before is true. The point p_3 was at the front of the queue, because it was closer to the query object than all the other points. However it was not closer than all the rectangles, which is especially true for R_1 and R_3, as it is contained within them.

We also see from this example why the algorithm is called Incremental Nearest Neighbour algorithm. After the last step we could simply pick up from where we stopped the algorithm and also get the next nearest neighbour. This of course would again be the object at the front of the queue, which in this case is the point p_2. The algorithm thus stops if we have our desired amount of closest neighbours or once all objects have been returned, since at that point the priority queue would be empty.

7.3 Distance functions

Now that we know how to find the nearest neighbour, we still need to define the distance functions from a given query point to another data point, an edge or to the nearest rectangle.

For point data it is quite straightforward, as it is simply the Euclidean distance between the two points.

Finding the shortest distance to an edge is already a bit more complicated. The basic idea is to project q onto the edge $[A, B]$ and then measuring the distance between the query object and the projected point like we did for point data. The only problem is finding the projection, as computing it for a line segment like our edge is a bit harder than computing it for an infinitely long line. This is why we compute the projection of q onto the line which contains A and B. The idea is to check whether this projection lies between the two points or not. If it is inside, we calculate the distance from q to the projection, as it lies on the edge which we are interested in. If it instead lies outside the interval, then the closest point on the edge must either be A if the projection is part of the rest of the line with end point A or B otherwise. The algorithm thus consists of the following steps:

We first project q onto the line spanned by A and B using the formula

$$\text{proj}(q) := A + \overline{AB}\ \frac{\overline{Aq} \cdot \overline{AB}}{||\overline{AB}||^2}, \tag{10}$$

where $\overline{Aq} \cdot \overline{AB}$ is the dot product of two vectors and the norm used is the Euclidean norm. To find out onto which part of the line we projected the point, consider that each element of the line x has the representation $x = A + \eta\,\overline{AB}$ for some $\eta \in \mathbb{R}$. Since $\text{proj}(q)$ lies on the line, it also has a corresponding η value, which we denote by η_q. Thus, the closest point on the edge is A if $\eta_q \leq 0$, B if $\eta_q \geq 1$ and $\text{proj}(q)$ else. Comparing Equation (10) with the representation of a point on the line also immediately gives us the formula

$$\eta_q = \frac{\overline{Aq} \cdot \overline{AB}}{||\overline{AB}||^2}.$$

Lastly we are interested in the shortest distance between a point P and a rectangle, since these are used in the construction of R-trees. This is made a lot easier by the fact that these rectangles' edges are parallel to the axes of the used coordinate system. Because of this, one can simply consider each coordinate of the point P and check whether it is inside or outside the corresponding minimum and maximum values of the rectangle. The different coordinates of the closest point to P on or inside the rectangle can then be found in a similar manner to the nearest point on a line segment.

7.4 Building the beta-skeletons

Now that we know how to calculate the distances, we can start building the structure. Since the simulation of the Poisson point process is quite straightforward with the way it was described in Section 3.5, we can go directly to the details of building up the beta-skeletons, as there are several ways this procedure can be sped up.

We start with some first vertex v_1 of the point process and calculate its nearest neighbours. Starting with the first neighbour n_1 we go through the list of the nearest neighbours to see if any of the other points are inside the critical region. Our first optimization is, that we do not need to check all other points. We only need to check the nearest neighbours within a certain radius of v_1, which depends on the distance between v_1 and n_1 and the parameter b. For $1 \leq b \leq 2$, we only need to consider points within radius $d(v_1, n_1)$ of v_1, as the farthest point in this case is n_1. If $b > 2$, then we instead multiply this radius with $\sqrt{b/2}$. To get this we consider Figure 19.

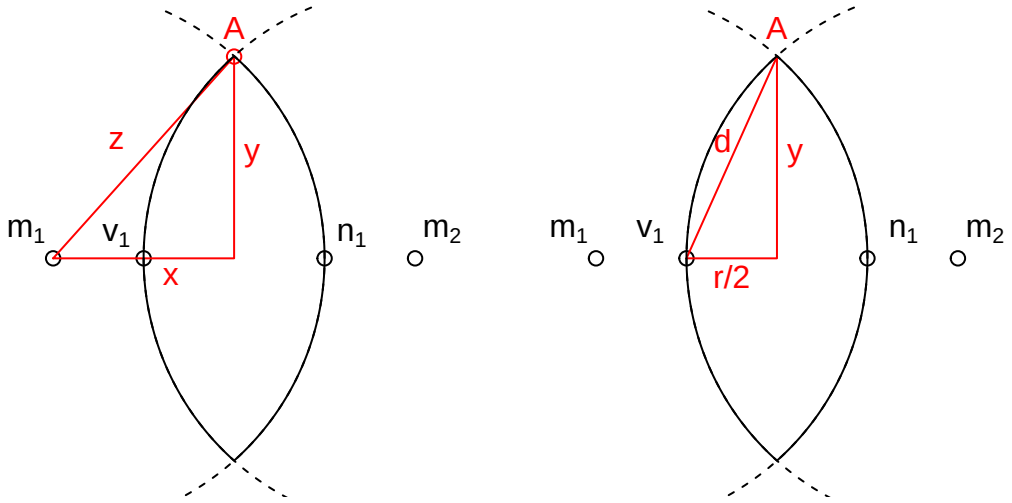

Figure 19: Farthest points in critical region for b>2

The farthest point from v_1 inside the critical region is the point A. To find its distance to v_1 we first compute the side y. We denote the distance between v_1 and n_1 by r, which results in

$$x = \frac{r}{2}(b-1) \quad \text{and} \quad z = r\left(\frac{b}{2}\right),$$

both of which follow from the definitions of m_1 and m_2. Using the Pythagorean theorem, we get that

$$y = \frac{r}{2}\sqrt{2b-1}.$$

The distance d between v_1 and A is then given as mentioned by

$$d = \sqrt{y^2 + \left(\frac{r}{2}\right)^2} = r\sqrt{\frac{b}{2}}.$$

Having found all edges of the beta-skeleton that contain v_1 this way, we continue to the next vertex v_2. As before we only check if a point is inside the critical region, if it is within a certain radius around v_2. But unlike before, we do not check whether to put an edge between v_2 and v_1, as this was already done in the first step. Similarly we do not check $v_1, ..., v_{k-1}$ when considering the vertex v_k.

The last optimization is to not actually compute and check all nearest neighbours. Since we would have to compute all nearest neighbours for each of the points, this would equate to $n(n+1)/2$ computations, where n is the number of points in a process. However, after extensively testing how many neighbours are actually needed to fully compute all edges for a given point inside a beta-skeleton, the number of neighbours which needed to be checked was never more than 50 even in the most extreme cases. This number includes the farthest edge partners as well as the farthest neighbours, which were in the critical region of other points. The actual amount needed for an exact computation can of course be exceeded in some examples, but these cases seem to be extremely unlikely. This means that one trades having to only compute $\mathcal{O}(n)$ nearest neighbours instead of quadratically many for some small inaccuracies in the beta-skeleton, which are very unlikely to happen.

7.5 Building the voxel grid

With the Poisson point processes simulated and the beta-skeletons calculated we can finally get to computing the phases of the energy cell. This is the most computationally expensive part, as unlike before we are not

finding the nearest neighbour for a few thousand vertices, but instead for several million points in space. This high number stems from the fact that while the parameters like the intensities refer to voxels with a side length of 1 μm, the voxels which are used to represent the phases are much smaller. In the example we simulate, the voxel size is only 30 nm. While computing which skeleton is the closest for each voxel is a pretty straightforward task, it is also greatly inefficient. This is why we will next look at several optimizations which one can use to greatly reduce the runtime of the algorithm.

7.5.1 Optimizations

The first such optimization is the most important one. It relies on a pretty intuitive fact, namely that if a voxel is quite close to one skeleton and very far from the other two, then the same thing will most likely hold true for any adjacent voxels, which means we should put them in the same phase. The way this is accomplished in the program is, that we start with some random point in the grid. For this starting point we calculate the distances to the nearest points and edges of the beta-skeletons of all three phases. We assign the point to the nearest phase with respect to the defined distance functions and remember the closest point and edge of the respective skeleton. For any neighbouring voxels we then compute the distance to the same point and edge. We then proceed differently for the other two phases, as we only remember their respective edge and vertex distances instead. With this we compute what I call the worst-case distances. They are the closest possible distances the neighbouring voxels can have to the other two phases, if we only know the distances for the original voxel and its distance to its neighbours. For phase i these are given by

$$d_{i,V}^{WC}(p,q) = d(q,V_i) - d(p,q),$$

for the worst-case distance to the nearest vertex and

$$d_{i,E}^{WC}(p,q) = d(q,E_i) - d(p,q),$$

for the nearest edge. Here, p is the neighbouring voxel we are interested in, q is the start voxel and V_i and E_i are the vertices and edges of phase i. For the same voxel p we now compute its distances to the vertex and edge which we remembered from the closest beta-skeleton. We can then assign voxel p to the same phase as q as long as all other worst case distances are larger than the distances between p and the remembered beta-skeleton. When comparing these values, one still has to multiply the distances to the edge sets with the corresponding parameter γ_i.

First we want to show that this assignment of p is valid. For this reason we proof, that the worst-case distances are really the shortest possible to the other two phases.

Theorem 7.1. *Let p and q be points in space and $d_{i,V}^{WC}(p,q)$ and $d_{i,E}^{WC}(p,q)$ the respective worst-case distances for p. Then no point or edge in phase i is closer to p than the respective worst-case distance, i.e $d(p,V_i) \geq d_{i,V}^{WC}(p,q)$ and $d(p,E_i) \geq d_{i,E}^{WC}(p,q)$.*

Proof. We only prove the theorem for the closest vertex, since the proof for other case works analogously. Assume that there is another vertex v in phase i, which is closer to point p than $d_{i,V}^{WC}(p,q) = d(q,V_i) - d(p,q)$. By the triangle inequality we get that

$$\begin{aligned} d(q,v) &\leq d(q,p) + d(p,v) \\ &< d(q,p) + d(q,V_i) - d(p,q) = d(q,V_i). \end{aligned}$$

This however is a contradiction, as $d(q,V_i) \leq d(q,v)$, since v is an element of V_i, proving the theorem. □

We thus assign neighbours around the point q to the same phase, as long as the worst-case distances allow us to do so. The reason for only checking the worst-case distances for the other phases instead of computing the distance to the nearest point or edge is quite simple. It is possible that the nearest part of the beta-skeleton for point p and point q are separate entities. Using only the information of the closest parts might therefore lead to wrong assignments of the neighbouring points. This could be remedied by computing the distances

from p to several points or edges of the other phases, to be sure that we work with the closest one. Doing it this way has the advantage of possibly being able to assign more neighbours, but at the cost of some extra calculations.

Afterwards, we continue with the next random voxel inside the grid, again checking the distances to all beta-skeletons and using the information for the neighbours of q. Going through the voxels in random order has the advantage of needing fewer nearest neighbour calculations, as we can rely on the worst-case distances instead. If we instead calculate the grid layer by layer, then a lot of neighbouring points will usually already be calculated. Another possibility would be to choose a predetermined order, which allows for the same exploitation of worst-case distances as the random order. But since the random order is a very simple, albeit not perfect, answer to the potentially hard question of which order is best, it was chosen in this case.

This optimization already saves us a lot of computations, but we can even expand on it. The idea is similar to before, in that it uses worst-case distances again. Instead of only marking voxels from their neighbours information as belonging to some phase, we instead mark which phases it could be a part of. As an example we consider the grid in Figure 20. Here we have two voxels which we already checked and which are marked by an orange border. The neighbouring voxels which surround them can only belong to the blue phase, since the other phases are very far away, so the respective worst-case distances are still bigger than the distance to the blue vertices. Surrounding those are voxels which could belong to either the blue or the red phase on the left side and ones which could belong to the blue or the green phase on the right side. The voxels which are left blank could be part of any of the three phases. The voxel which we are interested in now is the one where we have neighbour-information from both starting voxels. From the left voxel we know, that it can only belong to either the red or the blue phase and from the right voxel we know that it can only belong to the green or the blue phase. Combining this information, we know that the voxel in the middle can only be part of the blue phase, as the other two have been excluded by the information gained from one of the starting voxels.

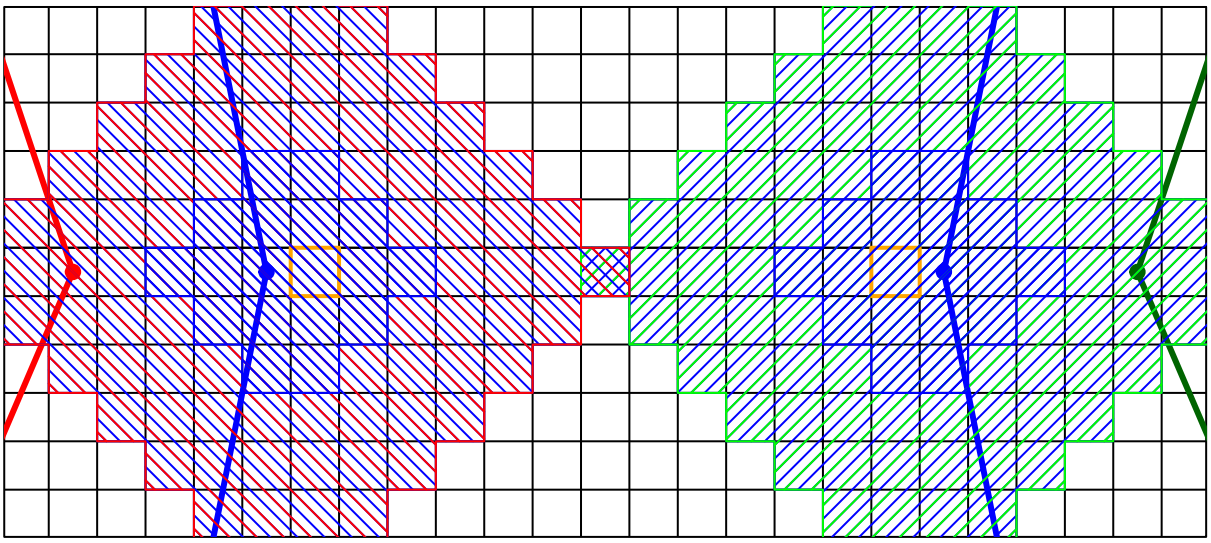

Figure 20: Using multiple neighbours' information

We can even take this idea one step further. Instead of only marking which phase a neighbour can belong to or not, we also distinguish between nearest vertices and edges. This means that we can potentially exclude a phase, because the information from one starting vertex excluded only its vertices, while the information from another vertex excluded its edges.

Saving this information has an additional purpose. In case we already know which phases' vertices or edges are possible candidates for being the closest to some new starting point, we only need to compute the

respective nearest neighbours and compare those. However, this is only done if we do not need to calculate any neighbours of this new starting point, since in that case we might need the information.

Lastly we do not check any neighbours in unnecessary directions. This can be implemented in several ways, but the general idea is to only check neighbours which are farther away, if we were able to exclude some phase for closer neighbours in the same direction.

8 Comparison to other approaches

Finally we want to see how the algorithm compares to other approaches. First we will compare it to the naive approach to the problem of building the beta-skeletons and the voxel grid, since at the beginning of the last section we claimed that this would not be a feasible option. Instead of using an R-tree, this approach simply computes the distances to all points as well as to all edges in the case of the voxel grid, to see which one is the closest.

8.1 Runtime Differences

First we compare how much of an improvement R-trees provide, when we only compute the beta-skeletons. The corresponding results are shown in Figure 21, which depicts the running times of the algorithm that computes the beta-skeletons. On the x-axis we are given the volume of the cubic observation windows in which we simulate the Poisson point processes with intensity $\lambda = 1$, resulting in roughly the same amount of points as the cubes' volumes. On the y-axis we have the time needed to compute the skeletons in minutes. It should be noted that the times are the means from several simulations, which results in a smoother graph. Also, the same parameter $b = 1.5$ was used for the simulation of all skeletons.

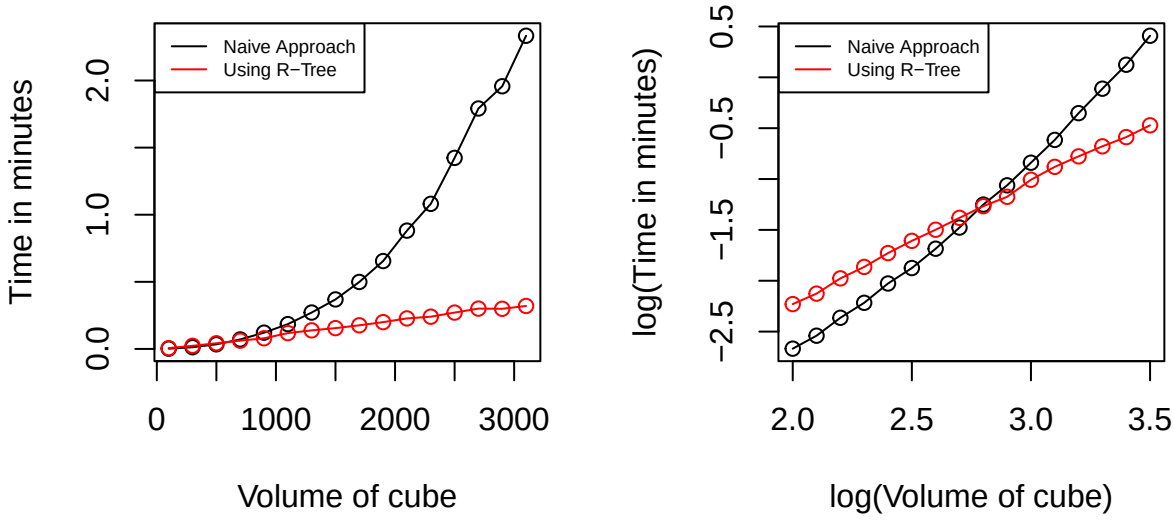

Figure 21: Comparison 1: Computation Times for Beta-Skeletons

In the left graph we can clearly see, that the naive algorithm takes a lot longer to compute the skeletons as the volume of the window gets larger. However, for smaller volumes it is hard to tell which algorithm is faster and by how much. Also getting a rough idea of the asymptotic runtimes would be nice, but it is not easy to tell from the first graph. Both of these problems can be alleviated by instead looking at the log-log-graph, which is that same graph, but we consider the logarithm of both axis instead, in this case with base 10.

It is easy to see that this change of perspective solves the first problem, as it is now clearly visible that the naive approach is actually faster when working with fewer points. The reason for this is, that the runtimes for the R-tree approach also include the time it takes to build the R-tree. This extra effort is not needed when using the naive algorithm, since it can simply take the list of points as its input. Only when working with more than about 500 points it pays off to build the R-tree.

For the other problem however, it is not so clear how the logarithmic axes help us. This is why we first consider the following equation, which assumes that the runtime is a polynomial function of the volume of the observation window.

$$\text{Time} = c \cdot \nu_3(W)^n$$

If this is the case, then taking the logarithm of both sides instead gives us

$$\log(\text{Time}) = \log(c) + n \cdot \log(\nu_3(W)),$$

revealing a linear trend. Since both log-log-graphs indeed look like linear functions, we can try to approximate the value of n and with it get a good idea of the algorithms asymptotic runtime. To do this approximation we fit a linear model to both log-log-graphs, which directly gives us the intercept $\log(c)$ and the slope n. The resulting values can be found in Table 1.

	Naive Approach	R-tree Approach
$\log(c)$	-6.925	-4.582
n	2.051	1.183

Table 1: Parameters of linear regression of time spent on building the beta-skeleton

The intercept values do not actually tell us a lot by themselves, but they allow us to compute where the two linear functions which best approximate the graphs intersect. The two graphs meet at $x = 2.7$ which equates to a cube volume of almost exactly 500.

However, the more interesting values are the two values we get for n. Since for the naive approach it is almost exactly 2, we get an asymptotically quadratic runtime in the number of points. This is also what one would expect, as this algorithm has to compare $\binom{N}{2}$ pairs of points, where N is the number of points. We then get the quadratic runtime from

$$\binom{N}{2} = \frac{N(N-1)}{2} = \frac{N^2 - N}{2} \in \mathcal{O}(N^2).$$

The other value of n is a bit harder to interpret, since it is not an integer. We only know that it is a lot smaller than 2, so far better than quadratic runtime, but the algorithm is still not quite linear.

Since the simulation only considered somewhat small examples, going up to 3100 points in the point process, the time to build a beta-skeleton does not seem that bad with the naive approach, as it only goes up to about 2.5 minutes. Going up to 7000 points, which is roughly how many the examples from the original paper contain, the time increases to 10 minutes for the naive approach and to only about 1 minute for the R-tree approach. Since we need to compute 3 skeletons, these time saves add up to 27 minutes. But unless one wants to simulate a lot of these skeletons, it would still be feasible to use the naive approach in this case.

The same however cannot be said when actually computing the entire voxel grid, as we are about to see. Since we already considered the time to build the R-tree and the beta-skeleton in the last simulation, we will only look at the time it takes to compute the voxels. We would also like to know how much time the optimizations saved, which is why we will first only consider the time it takes to build the grids without said optimizations. In Figure 22 we see exactly those runtimes, once with normal axes and once again with logarithmic axes. Since these runtimes start to get really big really fast, they are only the results of extrapolation. This means that instead of letting the algorithm compute all grid voxels, which for the larger cubes would be several million, we instead look at how long it takes to compute a few hundred voxels and scale it up to the entire observation window. The chosen parameters for this simulation were $\lambda = 1$ for all Poisson point processes and $b = 2$ for all beta-skeletons.

In the last comparison the naive approach was faster up until about 500 points, whereas this time it is already slower at around 30 points, as we can see in the second graph. The reason for this is the large increase in nearest neighbour computations needed to compute all the voxels, as we are working with the same resolution

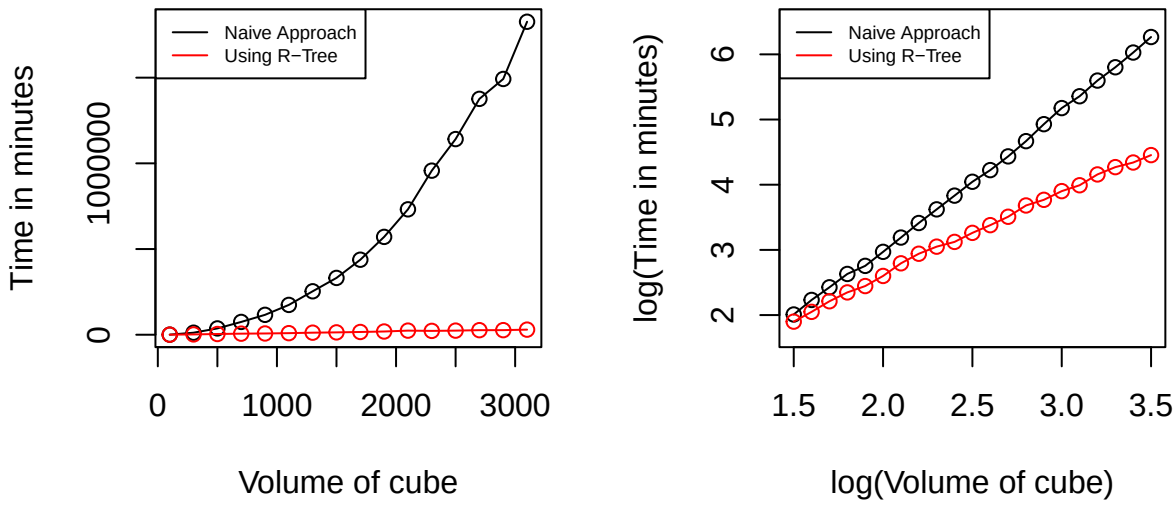

Figure 22: Comparison 2: Computation Times for Voxel Grids

as in the original paper of 30 nm. The volume in this case is given in cubic micrometers, resulting in a little over a million voxels even at the starting volume of 30. We now also see why the naive approach really is infeasible, since for a cube of volume 3000 it already takes 1.5 million minutes which is about 3 years.

Since the log-log-graphs still look like linear functions, we again fit a linear model for both to get an approximation of their slopes. The corresponding regression parameters can be found in Table 2.

	Naive Approach	R-tree Approach
$\log(c)$	-1.244	0.05255
n	2.128	1.277

Table 2: Parameters of linear regression of time spent on building the voxel grid

The values for the slopes are quite similar to before, which is to be expected, since they use the same algorithm to find the nearest neighbours as the one for the beta-skeleton did. The only difference is, that for the beta-skeletons the base number of points for which we had to find the nearest neighbours was always about equal to the volume of the observation window, whereas now it is roughly equal to $\nu_3(W)/\text{res}^3$, with res being the resolution or in other words the side length of a single voxel. This also explains the differences between the values of $\log(c)$, as the scaling might be similar, but the base amount of computations is much larger in this case.

As mentioned we are also interested in the amount of time saved by the optimizations which were implemented. For this reason we compare the runtimes to compute the entire grid using R-trees, but once with the optimizations and once without them.

As we can see in the graph with normal axes of Figure 23, the optimizations really do speed up the process by quite a bit. It is so much faster in fact, that it is really hard to tell just how much faster it is or how quickly the runtime grows. For this reason we again consider the log-log-graph, in hopes that we will get more information from it. We can clearly see a linear pattern again and in fact it shows pretty much the same slope as the algorithm without the optimizations. However, the intercept is quite a bit smaller. Both of these are what we would expect, if we consider what kind of optimizations were actually used. Parallelization increases the number of nearest neighbour calculations and the worst case distances only reduce their amount which we need to compute, but neither of them change the calculation themselves. This is why we have a much smaller intercept, as we only need to compute the nearest neighbour for about 5% of all voxels, while

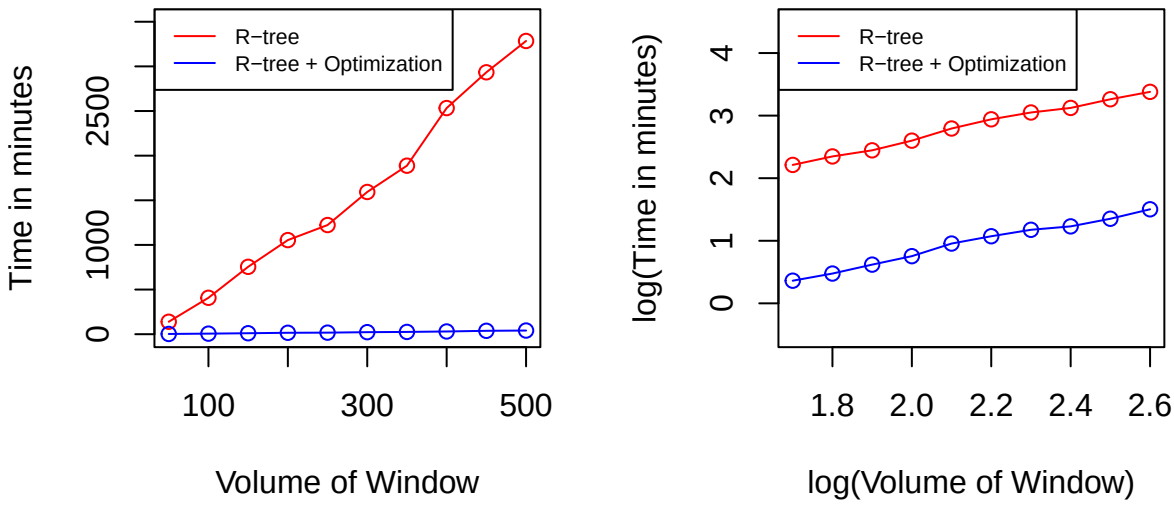

Figure 23: Comparison 3: Computation Times for Voxel Grids

also being able to do these computations in parallel. This however does not change the slope of the graph, since it is influenced the most by the time it takes to do a single nearest neighbour calculation, which of course stays the same.

Lastly we will compare the runtime of the optimized algorithm to the approximative algorithm used in [4]. For this we only need to consider the normal graphs, as it is known that the other algorithm only takes linear time.

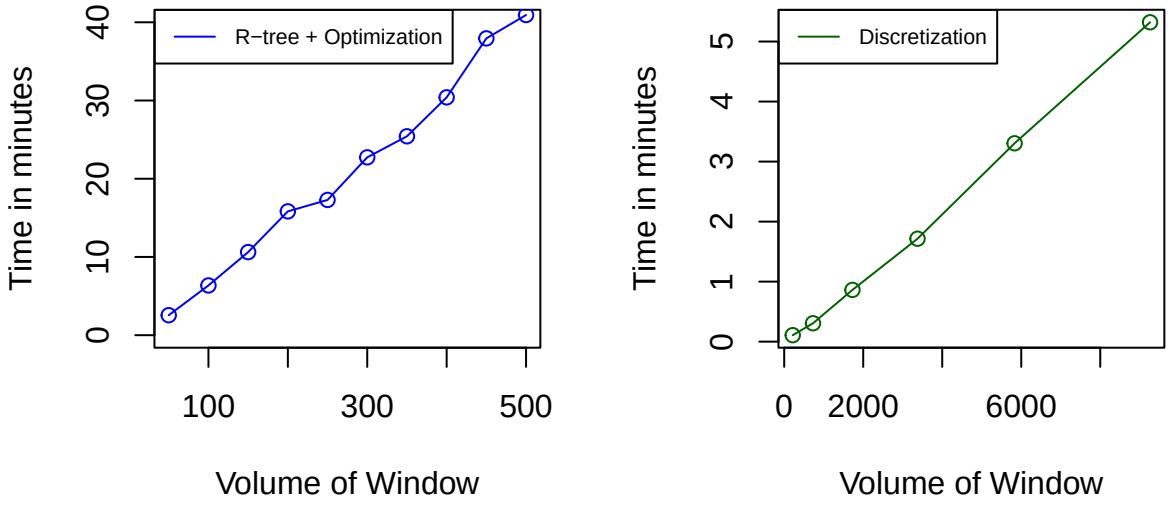

Figure 24: Comparison: Computation Times for Voxel Grids

The first graph shows us, that the runtimes for windows of size up to 500 are quite reasonable, only going up to 40 minutes. This time can also be improved by using more cores to compute the grid, as in this case only 12 computations were running in parallel. As already mentioned for the last log-log-graph, the runtimes are still not linear, even with the optimizations.

The same can not be said for the approximative algorithm. We see that the runtimes only increase linearly with the volume of the cube. Even computing the grid for a relatively large window, which would take about 5-6 hours with the optimized R-tree algorithm, can be done in only 5 minutes.

8.2 Accuracy of Approximative Approach

The approximative algorithm proposed in [4] is a lot faster and also its linear scaling is something which we cannot beat using the R-tree approach. The large difference of course is, that the R-tree approach can compute the discretization exactly. In order to investigate whether the extra time which we have to invest is worth it, we want to compare the accuracy of the two algorithms. Since we know that the R-tree approach computes each voxel exactly as we described, we can use it to compare the approximative grid to it.

In the following comparison, we compare 10 approximative voxel grids to 10 voxel grids which were computed exactly using the R-tree method. For each of these comparisons we started with the same Poisson point processes for both voxel grids. The chosen intensities λ_i, as well as the parameters b_i and γ_i, are the ones which were fitted to experimental image data in [4], which can be found in Table 3.

λ_1	λ_2	λ_3	b_1	b_2	b_3	γ_1	γ_2	γ_3
0.87	1.18	0.95	2.11	1.97	1.94	4.47	4.31	4.12

Table 3: Model parameters

Since the same values for the b_i were chosen for both methods and since the original paper did not approximate the beta-skeletons, we also get the same beta-skeletons for both methods. We only start to notice differences once we look at the computed voxel grids.

Each of the grids has a volume of 216 μm^3, which, since the grids are cubical, equates to a side length of 6 μm. We consider the same resolution as the voxel grids in the original paper, which means each voxel has a side length of 30 nm. Therefore each cube is made of $200 \times 200 \times 200$ voxel, or 8.000.000 voxels in total. However, to avoid edge effects, plus sampling was used for the point processes and the beta-skeletons. The extra space which was used was 10% in each direction, so for those purposes the side length of the observation window was 8.4 μm instead. It should be noted, that some parameters, especially the tortuosities, might look very different if one considers observation windows of different sizes.

For our first comparison we consider the amount of voxels which the approximative approach assigned correctly. The percentage of correctly assigned can be seen in Figure 25.

The points in blue indicate how many voxels were correctly assigned in total. As we can see, the amount of correctly assigned roughly stays the same for all 10 grids, all lying around 97%. This means that throughout the entire voxel grid only 3% get incorrectly assigned.

However, some voxels are less important than others when computing certain characteristics. This is why we also consider the amount of correctly assigned voxels at border regions. In our case those are all voxels for which one of the 8 adjacent voxels that share a side with it are not part of the same phase. For this the border voxels of the exactly computed grid are considered, however the values should not be off by a lot if the border voxels of the approximate grid were looked at instead. We see that for these voxels the accuracy drops by quite a lot to only around 87-87.5%. The percentage of incorrectly assigned border voxels is thus 4 times as large as it is for all voxels.

Since we mentioned that the triple phase boundary is a very important part of the grid, we will also take a look at how many voxels at this border got correctly assigned. Again we consider the voxels which are at the triple phase boundary of the exactly computed grid. We consider a voxel to be part of the boundary, if it is part of a $2 \times 2 \times 2$ cube, which contains at least one voxel from each phase. Looking at the graph, we see that these voxels are even more susceptible to being incorrectly assigned. On average only 78% of them were correctly assigned. The error rate is therefore almost double that of the border voxels and about 7 times as high as it for general voxels.

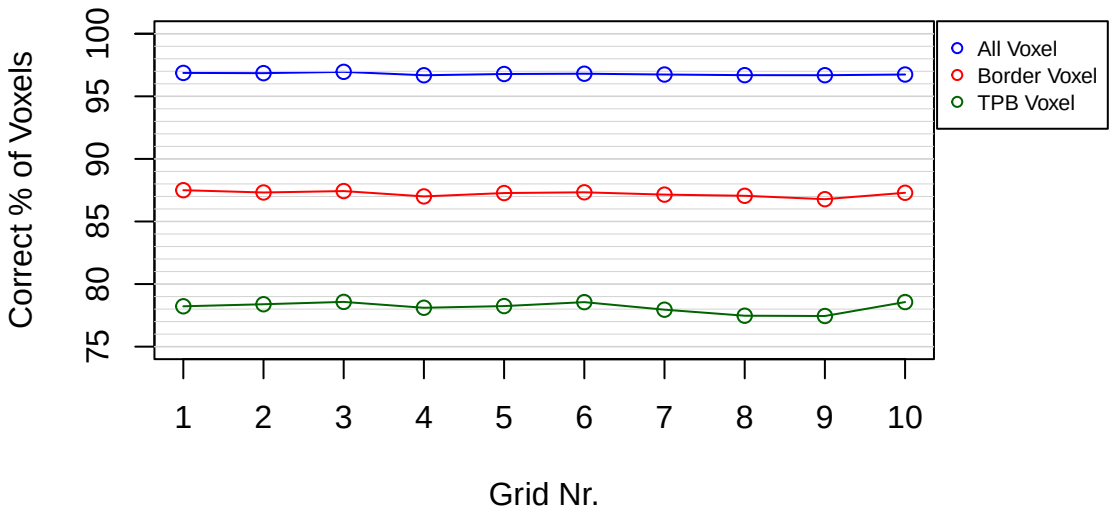

Figure 25: Correctly Assigned Percentage of Voxels

All this means, that if someone is interested in building the voxel grid, then it would be advisable to choose the R-tree method over the approximative one. However, since our main goal is to simulate a realistic structure, one might not care about the exactness of this grid too much, as long as estimated characteristics are not too far off from what they should be. For this reason, we next compare all the characteristics which we mentioned in Chapter 6.

We start off with the easiest to compute and compare, the volume fractions, which are depicted in Figure 26.

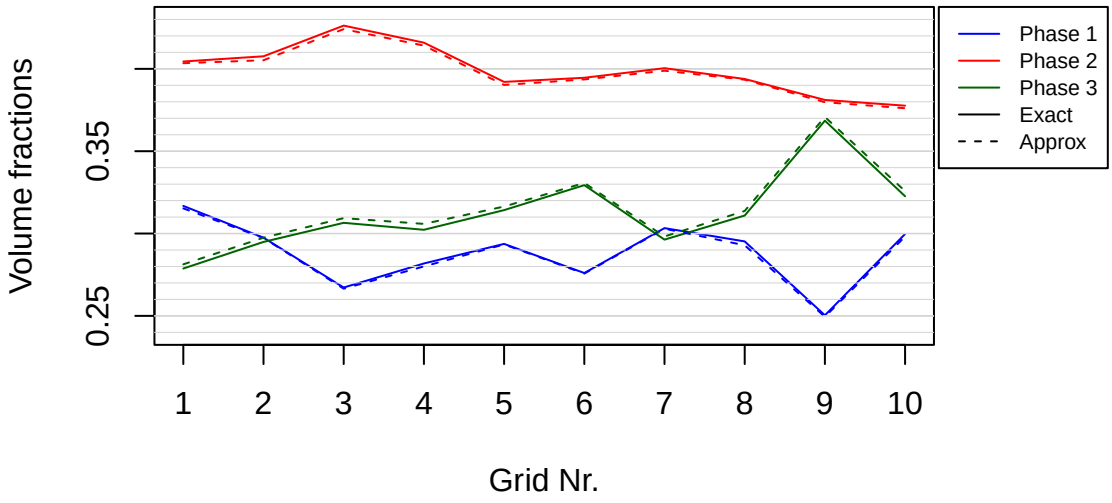

Figure 26: Comparison of Volume Fractions

With this comparison the approximative approach already looks a lot better, as none of the values are off by more than 0.005. However, to see exactly how correct these values are, we should also check by how much percent the fractions are off. Doing so reveals that the error rate is below 1% for all three phases. This is quite a bit lower than the amount of incorrectly assigned total voxels from before, even though these

two metrics should be closely intertwined. The reasoning for this is that multiple errors were made, which cancelled each other to some extent. For example, while some voxels were mistakenly not assigned to phase 1 at some point, there were also other voxels which should not have been part of phase 1, but are anyway.

From these volume fraction values we can already make some assumptions as to how they are effected by the different model parameters. We can observe that phase 2 has by far the largest volume fraction in all 10 simulations. Since its values for b and γ are both average, this is most likely due to its intensity, which is a lot bigger than for the other two phases. On average phase 3 also seems to occupy more space than phase 1, which could also be due to its slightly bigger intensity. However in this case it is not as clear-cut as before, since phase 3 also has a smaller b value, which leads to more edges in the skeleton, as well as a smaller γ value. Both of these will usually result in more voxels being assigned to phase 3.

A standard approach to finding out how the parameters influence the characteristics would be to fit a model. However, this is not possible in this case, since all predictor variables are the same for every observation. Another idea, which is not applicable here for the same reason, is to plot the two values whose connection one wants to investigate, as was done in [4]. There, several three-phase microstructure models were simulated, which only differed in the parameter $\gamma_1 = \gamma_2 = \gamma_3 = \gamma$. These different values of γ were then plotted against an estimator for the constrictivity $\hat{\beta}_{i,W}$, a characteristic which we only mentioned in passing in Section 2.2, to find an inverse correlation between them.

The next characteristic we will consider are the mean geodesic tortuosities. Again we plot the values as before, resulting in Figure 27.

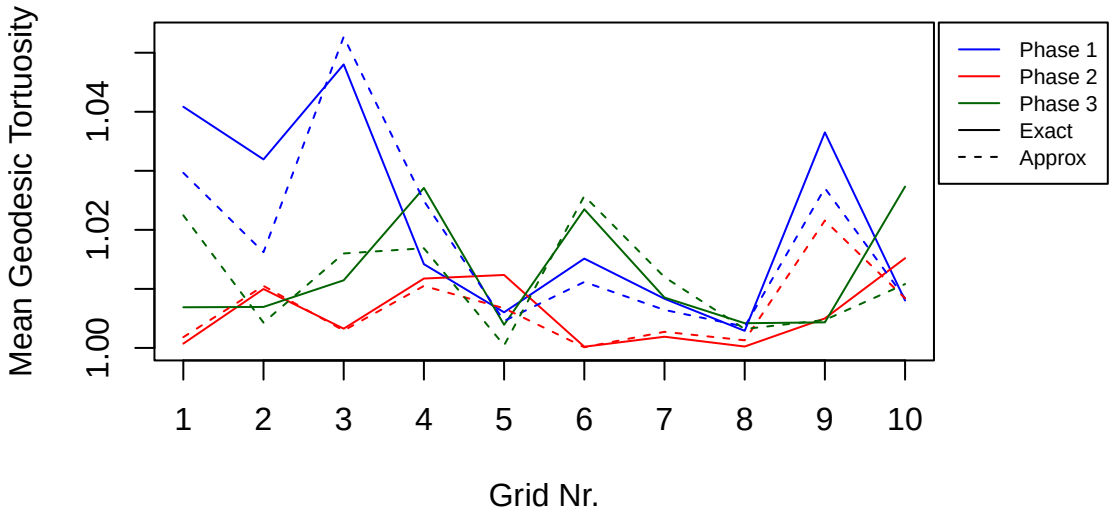

Figure 27: Comparison of mean geodesic tortuosities

At first glance it looks like the tortuosity values are off by quite a bit for some simulations. This however is mostly because of how the axes are depicted, as the smallest value on the y-axis is 1 instead of the usual 0. If we again only consider by how many percent the values are off, then the tortuosities are effected even less then the volume fractions, as they are on average only off by 0.5%.

Something which still should be considered however is that while the actual error might be smaller, it still is big enough to effect the ranking of the values in some cases. For example in grid number 4 phase 3 has the biggest tortuosity in the exact case, whereas phase 1 has the biggest in the approximate case. This however might simply be due to the fact that the tortuosities aren't very large in the first place, since the observation windows are not very large. The larger the window becomes, the higher is the possibility for large mean geodesic tortuosities, since paths which pretty much go straight through the material become more unlikely. So it is likely that this change in ranking would not happen, if the exact values were a lot farther apart, as

would be the case for larger cells.

Next we consider the last of the easily comparable characteristics we mentioned, namely the specific length of the triple phase boundary. This time the graph in Figure 28 is a lot less cluttered, as we only have 1 value for both approaches, unlike for the other 2 characteristics.

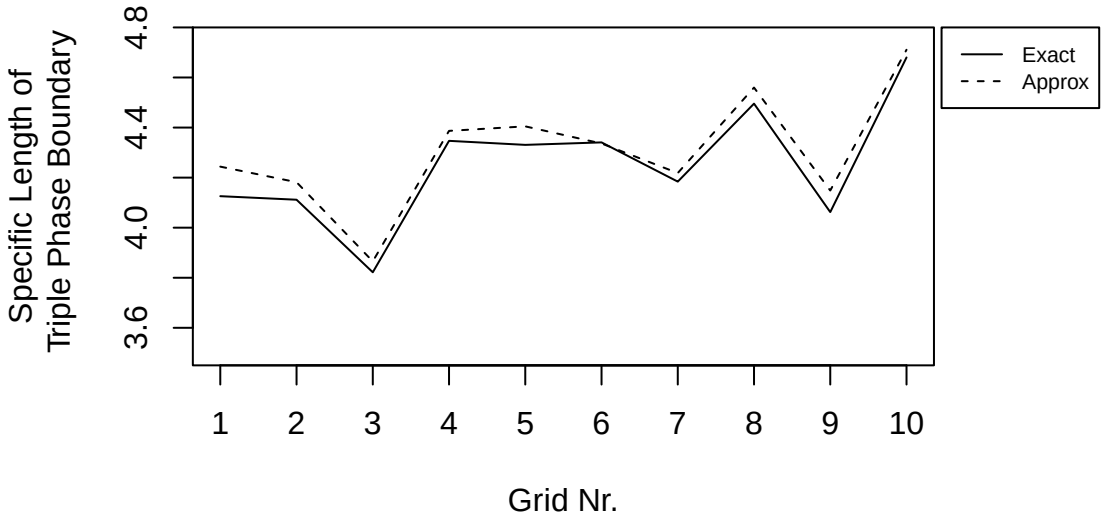

Figure 28: Comparison of specific lengths of triple phase boundary

Again the values are not off by quite as much as the voxels were. While one would assume a far greater discrepancy for this characteristic in particular, since the approximative model got almost every fourth triple phase boundary voxel wrong, it does not seem to matter too much for the actual length of the boundary. Although the difference is again a bit larger than for the mean geodesic tortuosities at about 1.3%. The reason for this is most likely that while voxels on the border and especially at the triple phase boundary are incorrectly assigned, the boundary mostly only gets moved around instead of assuming an entirely different shape. This also means that its length stays roughly the same, since most of the edge points which form the triple phase boundary are roughly at the same spot.

Lastly we will compare the contact distribution functions for both approaches. However, because this characteristic is not just a single value, but instead an entire function, we have a harder time comparing all them for all 10 simulations at the same time. Doing so would result in a very cluttered image, which is why we will only compare the distribution functions as a whole for a single simulation.

From Figure 29 we can obtain a few different informations. First off we see that for this first simulation, all the contact distribution functions are slightly larger for the approximative approach for most values of r. This is in fact the case for all 10 simulations.

We can also find a connection to the volume fractions. For example, for the first simulation, phase 2 has the biggest contact distribution function, whereas phase 3 is at the bottom. The same also holds for their respective volume fractions, as phase 2 has by far the biggest and phase 3 is again at the bottom. Even though we will not show this comparison for the other simulations, this same trend also holds for them. It is however less clear when two of the volume fractions are close to each other, like in simulation 2 for example.

In Figure 30 we can see the largest discrepancies between the two functions in percent. The difference is much larger for values where the approximative approach's contact distribution function is the bigger of the two. Thus, the graphs all show how much larger the approximative approach's function is at the point where the two differ the most. The only place where the exact approach is larger is at the very top of the graph, where the two functions are pretty much the same.

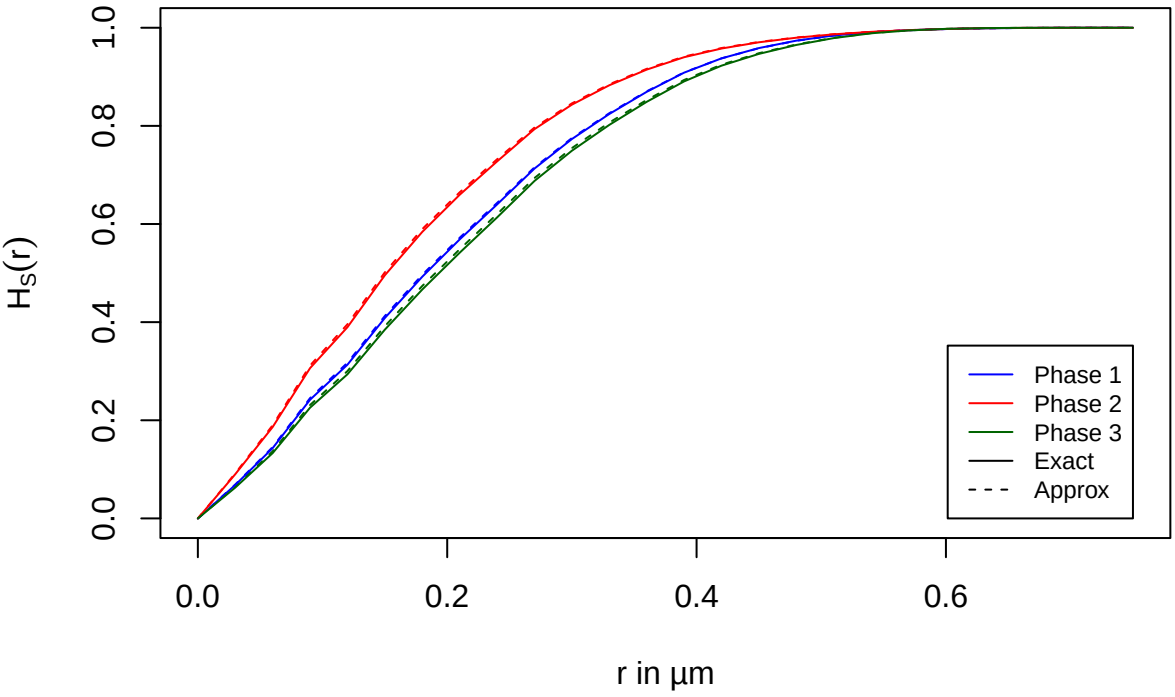

Figure 29: Comparison of spherical contact distribution functions

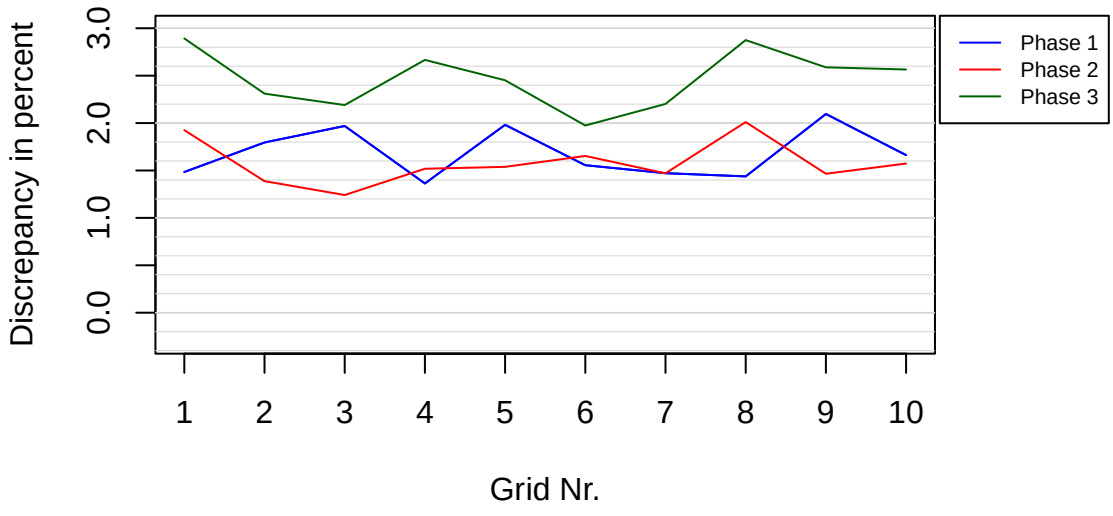

Figure 30: Maximum differences of spherical contact distribution functions

These errors themselves are not too big however, as they are only ever as big as 3%. This is also only the case for phase 3, since the errors for the other two phases are only ever as big as 2%.

Because the contact distribution function is the probability that some point outside the random set is within radius r of the set, it seems that something about the approximative approach causes the borders between the random sets to be less smooth than they are for the exact approach. The reason for this is that any small bulges in the smooth border of a set can lead to more space which is within radius r. One might assume that indentations would lead to less space within a certain radius of the random set, as the set itself would be smaller. However, the actual space which matters for the contact distribution function, which consists of all points outside the set, which are still within the radius, also become larger in this case. These two effects can be observed in Figure 31.

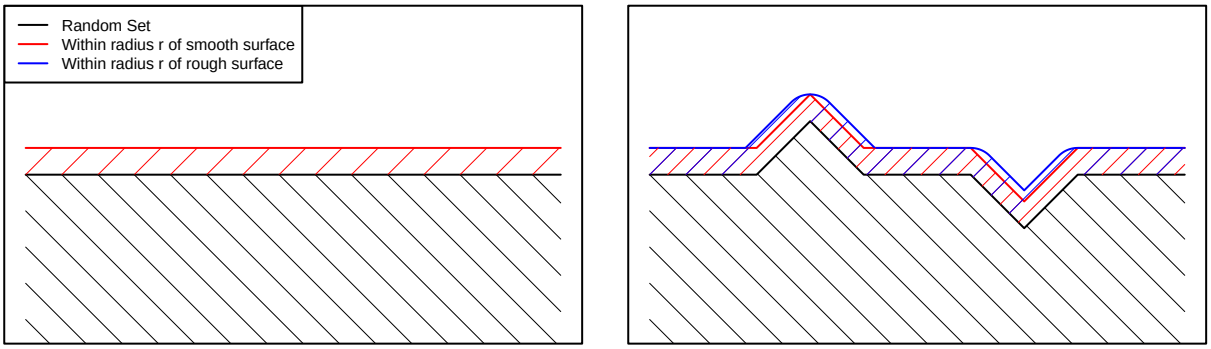

Figure 31: Effects of a rough surface on contact distribution function

The red area on the right graph is as big as the one on the left and is only here to compare the two. We can clearly see that even though the two random sets are the same size, the area which is relevant for the spherical contact distribution function is slightly bigger for the rough surface. The other characteristic where this slight roughness of the approximative approach can be seen is for the triple phase boundary, since it is also slightly larger for all 10 simulations. It also makes intuitive sense, that a rough surface can lead to more points at the triple phase boundary.

9 Conclusion

As we can see in Section 8.2, the approximate approach which was originally used is not too far off for all characteristics which we considered. All of them showed errors of up to only 3%, with most of their errors being even smaller. So even though the approximative approach assigns quite a few voxels to the wrong phases, especially at the border between the phases, this is not that big of a deal if one is more interested in the actual properties, which the resulting simulated cells exhibit.

One thing which should be mentioned is the reproducibility of the voxel grids. For the exact approach, one only needs the corresponding Poisson point process, as well as the remaining parameters, to produce the same voxel grid if one follows the simple instructions of assigning the voxels based on their proximity to the beta-skeletons. For the approximate approach however, one needs all of those things as well as the exact instructions of how the approximation was done.

So if one wants an easily reproducible and exact method of simulating these voxels grids and is fine with needing extra time to do so, one should go with the exact method. If however one wants to simulate a lot of voxel grids, which only come at the cost of some small errors for the characteristics, then the approximate approach is definitely the way to go, as it is quite a lot faster.

References

[1] Vafaeenezhad S, Hanifi A R, Laguna-Bercero M A, Etsell T H and Sarkar P 2022 Microstructure and long-term stability of Ni–YSZ anode supported fuel cells: A review *Materials Futures* **1** 042101

[2] Prifling B, Neumann M, Hlushkou D, Kübel C, Tallarek U and Schmidt V 2021 Generating digital twins of mesoporous silica by graph-based stochastic microstructure modeling *Computational Materials Science* **187:109934**

[3] Neumann M, Abdallah B, Holzer L, Willot F and Schmidt V 2019 Stochastic 3D modeling of three-phase microstructures for predicting transport properties: A case study *Transport in Porous Media* **128** 179–200

[4] Neumann M, Staněk J, Pecho O M, Holzer L, Beneš V and Schmidt V 2016 Stochastic 3D modeling of complex three-phase microstructures in SOFC-electrodes with completely connected phases *Computational Materials Science* **118** 353–64

[5] Shri Prakash B, Senthil Kumar S and Aruna S T 2014 Properties and development of Ni/YSZ as an anode material in solid oxide fuel cell: A review *Renewable and Sustainable Energy Reviews* **36** 149–79

[6] Pecho O M, Stenzel O, Iwanschitz B, Gasser P, Neumann M, Schmidt V, Prestat M, Hocker T, Flatt R J and Holzer L 2015 3D microstructure effects in Ni-YSZ anodes: Prediction of effective transport properties and optimization of redox stability *Materials* **8** 5554–85

[7] Chiu S N, Stoyan D, Kendall W S and Mecke J 2013 *Stochastic geometry and its applications* (Chichester, UK: John Wiley & Sons)

[8] R Core Team 2025 *R: A language and environment for statistical computing* (Vienna, Austria: R Foundation for Statistical Computing)

[9] Eppstein D 2002 Beta-skeletons have unbounded dilation *Computational Geometry* **23** 43–52

[10] Neumann M, Hirsch C, Staněk J, Beneš V and Schmidt V 2019 Estimation of geodesic tortuosity and constrictivity in stationary random closed sets *Scandinavian Journal of Statistics* **46** 848–84

[11] Guttman A 1984 R-trees: A dynamic index structure for spatial searching *SIGMOD Rec.* **14** 47–57

[12] Leutenegger S T, Lopez M A and Edgington J 1997 STR: A simple and efficient algorithm for R-tree packing *Proceedings 13th international conference on data engineering* pp 497–506

[13] Kamel I and Faloutsos C 1993 On packing R-trees *Proceedings of the second international conference on information and knowledge management* CIKM '93 (New York, NY, USA: Association for Computing Machinery) pp 490–9

[14] Faloutsos C and Roseman S 1989 Fractals for secondary key retrieval *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems* PODS '89 (New York, NY, USA: Association for Computing Machinery) pp 247–52

[15] Hjaltason G R and Samet H 1999 Distance browsing in spatial databases *ACM Trans. Database Syst.* **24** 265–318