Technische Universität München

Fakultät für Informatik
Lehrstuhl für Wissenschaftliches Rechnen

Efficient parallel algorithms for large-scale pedestrian simulation

Benedikt Sebastian Zönnchen, M.Sc.

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Nils Thürey
Prüfer der Dissertation:	1.	Prof. Dr. Hans-Joachim Bungartz
	2.	Prof. Dr. Gerta Köster (Hochschule München)

Die Dissertation wurde am 08.02.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.04.2021 angenommen.

I dedicate this thesis to my father, mother and sister,
who supported me during my whole life.

Abstract

The understanding and prevention of catastrophes at large-scale events are of utmost societal importance. For that, pedestrian simulation has proven to be a potent tool. Using microscopic pedestrian simulations, researchers and practitioners investigate the mechanisms and preconditions that lead to dangerous situations such as harmful crowd pressures. Simulations reveal the behaviors and characteristics of human crowds and suggest practical ways to prevent catastrophes.

However, microscopic pedestrian simulations are computationally expensive. Yet, it is necessary to model each individual to predict crucial phenomena. Despite their computational cost, real-time simulations are required to make reliable predictions during ongoing events and to enhance a research field that integrates more and more data-driven methods. To achieve this temporal requirement, we have to introduce and exploit efficient and parallel algorithms.

In this thesis, I follow the call for efficient and scalable simulations by analyzing existing and introducing new parallel algorithms. I introduce parallelism to a class of microscopic models, i.e., optimal steps models, and show that real-time simulations of half a million participants are possible. In addition, I develop efficient and parallel algorithms to construct navigation fields: a robust technique to model pedestrian wayfinding. A new meshing algorithm reduces the problem size and a novel numerical method exploits similarities of consecutively solved eikonal equations. In combination, real-time dynamic navigation field computation becomes possible for many large-scale scenarios.

Zusammenfassung

Katastrophen inmitten von Großveranstaltungen verstehen und verhindern ist von größter gesellschaftlicher Bedeutung. Für diese Aufgabe haben sich Fußgängersimulationen als wirksames Werkzeug erwiesen. Mit ihrer Hilfe untersuchen Forscher und unmittelbare Anwender die Voraussetzungen und Zusammenhänge, die zu gefährlichen Situationen, wie kritischen Personendichten, führen. Aus den gewonnenen Erkenntnissen über das Verhalten und der Bewegung von Fußgängern können wir praktische Maßnahmen ableiten und dadurch Katastrophen verhindern.

Mikroskopische Fußgängersimulationen sind jedoch rechenintensiv. Um aussagekräftige Vorhersagen zu erzielen, ist, bis heute, die Modellierung jedes einzelnen Individuums erforderlich. Trotz des dadurch entstehenden Rechenaufwands, wird der Ruf nach Echtzeitsimulationen immer lauter. Einerseits sollen sie die Anwender mit Vorhersagen während einer laufenden Veranstaltung unterstützen. Andererseits würden sie ein Forschungsfeld bereichern, welches immer mehr datengetriebene Methoden integriert. Um diese zeitliche Anforderung zu erfüllen, müssen wir effiziente und parallele Algorithmen für die Berechnung von Personenströmen entwickeln und nutzen.

In dieser Arbeit folge ich dem Ruf nach effizienten und skalierbaren Simulationen, indem ich vorhandene Algorithmen analysiere und neue parallele Algorithmen entwickle. Zunächst führe ich die Parallelität in die sogenannten Optimal Steps Modelle, eine Klasse mikroskopischer Modelle, ein. Ich zeige, dass dadurch die Simulation einer halben Million (virtueller) Fußgänger in Echtzeit möglich wird. Darüber hinaus entwickle ich effiziente und parallele Algorithmen zur Berechnung von Navigationsfeldern. Diese haben sich in der Vergangenheit als robuste Technik zur Modellierung der Wegfindung von Fußgängern etabliert. Ein neuer Algorithmus zur Netzgenerierung reduziert die Größe des zu berechnenden Problems und eine neuartige numerische Methode nutzt die Ähnlichkeit aufeinanderfolgend gelöster Eikonalgleichungen aus. In Kombination wird der Einsatz dynamischer Navigationsfelder für Echtzeitsimulationen für viele große Szenarien ermöglicht.

Preface

"The struggle itself towards the heights is enough to fill a man's heart. One must imagine Sisyphus happy."

– Albert Camus

My personal journey

I can pinpoint the exact moment when I decided to start my academic journey. At that time, my life was harshly disrupted by my own physical illness and the death of my father. Consequently, my family struggled on many levels. Despite, and probably because of the unfortunate circumstances and with my family's blessing, I quit my job and rejoined school to get my (technical) A level. I wanted to comprehend the world more than ever and escape the meaningless play of presenting. At that difficult time, there was a financial intensive to graduate as fast as possible. Therefore, I did not join the university but the computer science program at the Munich University of Applied Sciences.

During my undergraduate study, my former naive belief was crushed. I realized that I would never find any definitive objective truth about the physical world. It was a rather pessimistic but also liberating philosophical revelation that there is no definitive rule to follow and no absolute meaning to fulfill. My search was no longer aimed to find an objective meaning but a personal cause to follow. I still admired studying but for other more aesthetic reasons. I loved the clarity and usefulness of formal systems, the beauty of proofs, the elegance of algorithms, and the student's lifestyle. I observed a transforming world where computer science spread out into many branches of science, economics, and society. It was a time of playful experimentation and the deconstruction of personal barriers.

After I got my bachelor's degree, I finally joined a master program at the Technical University of Munich. This continuation was enabled by the individual and financial support I received from the Studienstiftung des deutschen Volkes & the Max Weber-Programm. I followed my aesthetic taste and visited rather unpopular formal lectures. At that time, I realized that the source of my personal cause to move on has to be bound to someone else. Beauty and aesthetic theories were a pleasant enjoyment, but they could no longer be the primary reason to live for.

Surprisingly my academic journey should not stop there, since Prof. Dr. Köster invited me into her research group. There are multiple reasons why I happily accepted her invitation. The desire to understand the world was reduced to the desire to understand at least one little part of it. Furthermore, I believed that simulations would influence science, economics, and society for

decades to come. And if I could make the world a little bit safer, it might be the cause I was looking for.

During my PhD, the wish to be useful and to help others was always a source of inner conflict and self-doubt. I started the whole journey to escape a world that I perceived to be shallow, empty, and driven by profit. From time to time, the scientific research project felt like this meaningless business world that came back to haunt me. Luckily I was in a superb position. Everyone tried to reduce this aspect of the scientific environment to a minimum, for which I am very grateful.

I think every PhD candidate has to deal with uncertainties and self-doubt – the uncertainty within science and the uncertainty of the journey's path. There is no guaranteed progress or graduation and one is constantly confronted with his or her own limitations. These factors and the ever-present questioning voice in my head acted as a catalyst for an unavoidable existential crisis. I looked into many philosophical ideas and rearranged, and possibly reinforced, my world view and many important values – the chapter's introductory quotes tell the tale. In my opinion, this process was only possible, because during my academic journey I received the tools to engage with difficult ideas. On the one hand, this crisis was unpleasant but on the other hand it enriched my life – a trade-off I am certainly willing to repeat. In the end, I had supportive companions that helped me to deal with all these issues.

It was not easy, and I guess it rarely ever is, but the struggle is part of the charm – as Camus said, "we must imagine Sisyphus happy". For me, to study is to train thoughtful thinking, perception, awareness, and tolerance. It sharpens the mind and opens up a little less ignorant new world. The ability to enjoy thinking and to share this enjoyment with other thoughtful people provides freedom and independence. It might be the greatest gift I received during my journey. Because of it and the people I met, it is a success story, and I am deeply thankful that I could have experienced it.

Acknowledgments

During the years of my dissertation, I was accompanied by many lovely people. They shaped my experience at my workplace in Munich, conferences in the Netherlands, China, the US and the Czech Republic, and, of course, my free time. This short paragraph can never dignify their significance but will act as a reminder for myself.

My primary thesis supervisor Prof. Dr. Hans-Joachim Bungartz gave me the support and stability that enabled my journey in the first place. He was and still is a source of inspiration beyond academia. A few meetings were enough to leave a permanent and defining impression. I am very grateful for the valuable time, expertise and advice he provided.

Early on, as an undergraduate, my supervisor at the Munich University of Applied Sciences, Prof. Dr. Gerta Köster, gently invited me to the scientific community. She and her PhD students Isabella von Sivers, Felix Dietrich, and Micheal Seitz showed me different perspectives on science and its environment. In her research group, I found a place of acceptance, care, and genuine curiosity that was constructed, led, and shielded by Prof. Dr. Gerta Köster. I deeply admired her philanthropic attitude reflected by her strong support for her staff and students and her sense of justice. Our weekly discussions during my PhD led me to new ideas and often helped me to overcome self-doubt and other obstacles. Thank you for everything.

Since the start of my dissertation at the pedestrian research group, I benefited in many ways from my colleagues. We worked together as a unit, exchanged strengths, discussed new ideas, and personal issues. Marion Gödel, Christina Mayr, Benedikt Kleinmeier, Daniel Lehmberg, Stefan Schuhbäck, and Lorenzo Servadei without you, I would not have been able to climb this rough mountain. It was a pleasure to work, struggle, and laugh with you. Thank you all!

The great spiritual and emotional support given by my family and friends was another important keystone that guided my journey. Besides many things, my family gave away the most beautiful experience that will forever shine through all things: unconditional love. My lovely mother and sister were always confident in me. I always knew: whatever happens, it will be ok and if things go wrong, there is always someone by my side. Regardless of the many duties my sister had to fulfill, she spent time to read my whole thesis and provided very useful remarks. My father could not be part of my journey, but he laid its foundation by providing meaningful lessons about a good life. One lesson consisted in the enjoyment of the small pleasures of everyday life. Therefore, I'm very thankful for all the culinary treats that were provided by Ingrid. Especially in the time of the pandemic, they made my life much more enjoyable.

My journey was shaped by solitude, but I was never alone. Luckily I met Mario when I started my undergraduate studies, and since then, he is a true friend: honest, constructive, avid, and reliable. We encouraged each other to pursue our way of life. Hours of philosophical, absurd, and funny debates created a playground for creative, free, and fruitful thought. In this space, I gained back my energy for the week.

Manu K., Jakob B., Mitch M., Daniel W., Felix S., Maika G., and Mr. Stein are friends who always cared even if I could not give back what I received. They are all true legends of my life, wise people, and teachers in their own way. Last but not least, I want to thank my friends from my online community. We spent many funny evenings together that helped me to silence the nagging voice in my head.

There are many more people I have not directly mentioned. I hope they will forgive me.

Contents

List of algorithms

List of figures

Notation

x	$x \in \mathbb{R}$
$\mathbf{x}$	$\mathbf{x} \in \mathbb{R}^d$
$\mathbf{X}$	$\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}$
μ	mean
σ	standard deviation
$\mathcal{U}(a, b)$	uniform distribution with bounds $(a; b)$
$\mathcal{N}(\mu, \sigma)$	normal distribution with mean μ and standard deviation σ
$\mathcal{N}_{\text{tr}}(\mu, \sigma, a, b)$	truncated normal distribution with $a < b$
$\mathbf{Pr}(A)$	probability of an event A
$\mathbf{Ex}(X)$	expected value of a random variable X
$\mathcal{A}$	set of agents
$\mathcal{W}$	set of obstacles
$n_{\mathcal{A}}$	number of agents
$d_{\mathcal{W}}(\mathbf{x})$	Euclidean distance from $\mathbf{x}$ to the closest obstacle
$\Omega \subset \mathbb{R}^2$	spatial simulation domain
$\partial\Omega$	boundary of the spatial domain
$\Gamma \subset \Omega$	spatial agent destination
$\mathbf{n}_\Gamma(\mathbf{x})$	destination direction of an agent positioned at $\mathbf{x}$
t	simulation time
Δt	simulation time step
$\rho_{\mathcal{W}}(\mathbf{x})$	obstacle density at $\mathbf{x}$
$\rho_{\mathcal{A}}(\mathbf{x})$	agent/pedestrian density at $\mathbf{x}$
v_0	mean free-flow speed
v_l	free-flow speed of agent l
δ_{tor}	diameter of the agent's torso
P_l	possible next agent positions of agent l
r_l	step length of agent l
u_l	utility function of agent l
$u_{\mathcal{W}}$	obstacle utility function
$u_{\mathcal{A},j}$	utility imposed by agent j
$d_\Omega(\mathbf{x})$	geodesic distance from $\mathbf{x}$ to $\partial\Omega$
δ_c	cell width and height of the linked cell data structure
n_c	number of cells of the linked cell data structure

$n_{\mathcal{A},c}$	number of agents of the most populated cell
w_c	number of columns of the linked cell data structure
h_c	number of rows of the linked cell data structure
w_Ω	width of a domain enclosing rectangle
h_Ω	height of a domain enclosing rectangle
$\mathbf{C}[i]$	value of the array $\mathbf{C}$ at index i
E	set of footstep events
$F(\mathbf{v})$	force applied to vertex $\mathbf{v} \in \mathcal{T}$
$F(\mathbf{u}_1, \mathbf{u}_2)$	force applied to vertex $\mathbf{u}_1$ caused by $\mathbf{u}_2$
$f(\mathbf{x})$	travel speed of the propagating wavefront at $\mathbf{x} \in \Omega$
$\Phi(\mathbf{x})$	travel time of the propagating wavefront at $\mathbf{x} \in \Omega$
$\Phi_0(\mathbf{x})$	initial travel time of the initial wavefront $\mathbf{x} \in \Gamma$
$\phi(\mathbf{x})$	approximated travel time at $\mathbf{x} \in \Omega$
$\lceil \cdot \rceil$	ceiling function
$\lfloor \cdot \rfloor$	floor function
θ_∞	maximal angle of an element of a mesh
θ_0	minimal angle of an element of a mesh
$\mathcal{T}$	a triangulation
$\mathcal{V}$	vertices of a triangulation
$\mathcal{E}$	edges of a triangulation
τ	a simplex
$\rho(\tau)$	quality of the triangle τ
$\rho(\mathcal{T})$	quality of the mesh $\mathcal{T}$
$\mathrm{con}(X)$	convex hull of a point set $X \subset \mathbb{R}^d$
$\mathrm{aff}(X)$	affine hull of a point set $X \subset \mathbb{R}^d$
$\lvert\tau\rvert$	the underlying space of a simplex
$\lvert\mathcal{T}\rvert$	the underlying space of a triangulation
$\mathcal{P}$	a planar straight-line graph
$\mathcal{DT}(\mathcal{V})$	the Delaunay triangulation of a vertex set $\mathcal{V}$
$\lvert\tau\rvert_0$	smallest angle inside the simplex τ
$\lvert\tau\rvert_2$	Euclidean norm of the edge lengths of τ
$\lvert\tau\rvert_\infty$	infinity norm of the edge lengths of τ
θ_0	smallest angle within a simplex
θ_∞	largest angle within a simplex
d_Ω	distance function of the spatial domain
$\mathbf{x}_e$	midpoint of an edge e
$\mathbf{x}_\tau$	midpoint of a simplex τ
$d_\mathcal{T}(\mathbf{v}, \mathbf{u})$	shortest path from $\mathbf{v}$ to $\mathbf{u}$ with respect to the mesh $\mathcal{T}$
$\sphericalangle \mathbf{u}_1\mathbf{v}\mathbf{u}_2 = \alpha_\mathbf{v}$	smaller angle at $\mathbf{v}$ of the of the triangle $\mathbf{u}_1\mathbf{v}\mathbf{u}_2$
$\pi(\mathbf{v})$	defining vertex relation of vertex $\mathbf{v}$
$\mathcal{V}_\mathbf{v}$	defining vertex set of vertex $\mathbf{v}$
$\mathcal{G}$	a graph
u_{FMM}	number of updates required by the FMM

CHAPTER 1

Introduction

"When one has once fully entered the realm of love, the world — no matter how imperfect — becomes rich and beautiful, it consists solely of opportunities for love."

– Søren Kierkegaard

1.1 Motivation: a call for real-time large-scale microscopic pedestrian simulations

In pedestrian dynamics, we study human motion mainly in urban environments. The interaction of multiple individuals is of special interest. Historically, pedestrian dynamics is motivated by the urbanization of human settlement and an increase in large-scale events. In the past, horrible accidents took place during the gathering of large crowds: 21 people died at the Loveparade in Duisburg (2010), many more were injured, an estimate of 717 pilgrims died during the Haji in Mina (2015), 36 people died at the New Year's Eve celebration in Shanghai (2015), and the list goes on. We must learn from these accidents, find their cause, and develop mechanisms to prevent them. Preventing harm to humans is our duty and most important goal of research.

Researchers in pedestrian dynamics study human behavior on a microscopic level to prevent dangerous crowded situations. They come from many different fields, which is reflected in a multitude of competing models, each of which stresses different characteristics of crowd motion. Even though the community will discuss novel models, there are many, such as cellular automata [31, 245, 155, 172, 19, 115], optimal steps models [257, 165, 300, 260, 327, 168, 158], and social force models [118, 326, 137, 190, 49, 167], that already reproduce important phenomena. They predict pedestrian behaviors and support practitioners in evaluating the efficiency of evacuation strategies and the safety design of facilities in buildings [72, 75].

New possibilities inspire new objectives, and each given answer provides additional questions. Today, I observe an increasing demand for *large-scale real-time* simulations. By the term *large-scale*, I refer to the number of simulated pedestrians and the area in which they move. In practice, simulations are part of the planning phase of large events. Nowadays, they should also serve as artificial intelligence that event managers can consult during an event – they should support the

ongoing decision-making process. Multiple calls for research projects promoting such systems were made. For example, the research contract of the S^{2}UCRE project [8] includes the study of rescue and safety technology in the context of large-scale events, such as the Hafengeburtstag Hamburg, the Cannstatter Wasen, or the Oktoberfest in Munich. Within the project, pedestrian simulations that are initialized by extracted, aggregated, and analyzed video footage, predict the future in an *online* setting. For data-driven modeling, online parameter learning [24, 25] also gained popularity.

There are many pedestrian simulation models. And before we, as a community, move on to introduce yet another model, I believe we have to better understand the existing ones. It is time to stop for a moment and to look around at what we have in front of us. There is a gap between the number of models and the effort to calibrate, validate, and analyze them. Quoting Duives et al. [76]:

> *"Even though calibration and validation are considered to be essential to determine the reliability and validity of simulation models, researchers currently apply inconsistent procedures or only partially test the simulation tools due to the lack of international standards for verification and validation of pedestrian flow and crowd dynamic simulation tools for general use."* – Duives et al. [76]

Recent studies [319, 138, 252, 303, 99, 98] point in the right direction. Uncertainty quantification (UQ) was introduced to pedestrian dynamics to support model analysis [175, 303]. It is a mathematical tool to reveal the relation of the behavior of a system to uncertainties in its parameters [70]. For more information about uncertainty quantification, I refer to [93]. Because of the high computational cost of quantifying the uncertainty of pedestrian dynamics, surrogate models proposed by, for example [70, 176], offer a data-driven solution. Dietrich et al. [69] stated concisely: "Surrogate models extract the most important features of a computationally intensive model from data produced by that model." Unfortunately, the construction of accurate surrogate models requires multiple sample points. For example, the authors in [69] used 1800 simulation runs for a rather simplistic bottleneck scenario. To my best knowledge, surrogate models for more complex large-scale scenarios have yet to be constructed. Aside from these sophisticated methods, plain calibration can also lead to a heavy workload, especially for large-scale scenarios.

In summary, large-scale pedestrian simulations are important, because most critical real-world situations emerge in a large-scale setting. The need for predictive *online* simulations impose temporal requirements with respect to the simulation run time – a simulation that runs slower than real-time can never predict the future. Additionally, I claim that the research community benefits from faster large-scale simulations. They open the door to new data-driven methods, intensified calibration, and more simulations in general. Therefore, an acceleration of simulations might translate into an acceleration of the research in pedestrian dynamics.

One open question remains: why should we use *microscopic* instead of *multiscale* or *macroscopic* simulations for large-scale pedestrian simulation? Microscopic approaches model each pedestrian as an individual agent with personalized properties such as its desired walking speed. The state of each agent is described by position and velocity, and other time-dependent variables [23, p. 4]. Contrary to this, macroscopic approaches model the stream of pedestrians as an aggregation of individuals. They can match the overall behavior characteristics, but the interaction

between individuals and individual behavior habits are ignored [72]. They are less computational expensive but

> *"even though these models are efficient and fast, [most of them] can generally not be used to assess large crowd movements within a complex infrastructure (i. e., train station, festival terrain, large square)."* – Duives et al. [76]

Examples for macroscopic models can be found in [120, 117, 129, 130, 291, 128]. Multiscale models in pedestrian dynamics, such as [283, 23, 179, 22, 29, 290, 270] have two scales: a microscopic and macroscopic scale. On the microscopic scale, pedestrians and their interactions are modeled explicitly. Information about the spatial local and inhomogeneous pedestrian behavior is transferred to the macroscopic scale. On that scale, the aggregated stream of pedestrians (globally) flows homogeneously with respect to local information. A more general and extensive description of multiscale models can be found in [56]. In [75], Duives et al. argue that microscopic models are "highly precise" but slow and macroscopic modeling attempts seem "behaviorally questionable".

> *"Both models have their use, which is highly dependent on the application the model has originally been developed for. Yet, for practical applications that need both precision and speed, the current pedestrian simulation models are inadequate."*
> – Duives et al. [75]

In my opinion, this statement from 2013 has aged well and poorly at the same time.

It is still true that the model of my choice depends on its application, including the phenomena of interest. Secondly, most microscopic models are more precise than most macroscopic models. Inhomogeneous pedestrian behavior disqualifies most macro- and multiscale models if inhomogeneity should be preserved. I argue that using either micro- or macroscopic models has less to do with the actual scenario scale (number of agents and domain size) but with the degree of homogeneity. One defines an airport with a hundred thousand passengers to be a large-scale scenario. On the other hand, a unidirectional corridor inside a building is undoubtedly a small-scale scenario. However, if we consider the degree of homogeneity, a microscopic model is needed for the airport to reproduce microscopic phenomena. On the contrary, a macroscopic model suffices in the much less populated corridor if we are only interested in macroscopic measures. In general, there is no 'correct' model or scale. Modeling is a question of complexity, or cost and accuracy [39, p. 11]. There is a relation between phenomena and scale, for example, self-organizing movements (lane formation, stop-and-go waves) take place at the microscopic scale.

The second part of the above statement aged rather poorly since there is a lot of progress in fast large-scale microscopic simulation, including this thesis and other contributions discussed in Section 4.3.

Overall, I conclude that there is a need for large-scale microscopic simulations.

1.2 Scope and overview of this work

In this thesis, I develop efficient parallel algorithms to accelerate navigation field-based microscopic pedestrian simulation. Thematically, I discuss three distinct topics: *parallel large-scale*

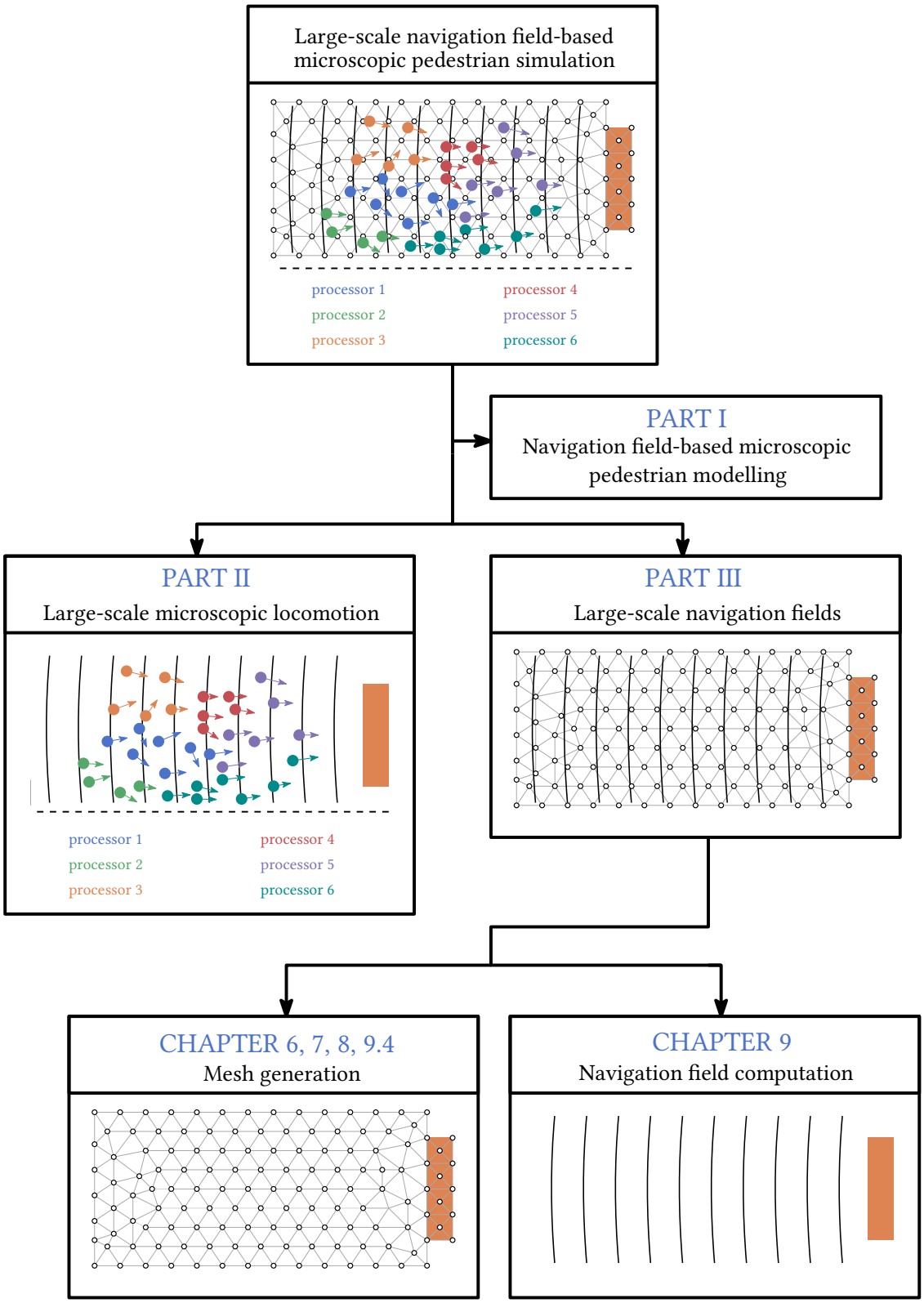


Figure 1.1: Overview of the content of the thesis.

locomotion models, *mesh generation for pedestrian dynamics*, and *efficient solution of the eikonal equation in the context of pedestrian dynamics*. By combining *mesh generation for pedestrian dynamics*, and *efficient solution of the eikonal equation in the context of pedestrian dynamics*, I enter the realm of *large-scale navigation field computation*. In the big picture, illustrated in Fig. 1.1, all three topics come together. I introduce the necessary foundation to keep the required background knowledge at a minimum. Consequently, this leads to a lot of explanatory content. I advise the hurried reader to skip introductory sections if he or she feels comfortable with the subject. I split the development of efficient parallel algorithms for microscopic pedestrian dynamics into three parts:

Part I: In Part I, I give an overview of microscopic locomotion models (Chapter 2) and wayfinding based on navigation fields (Chapter 3).

I analyze locomotion models algorithmically and evaluate their suitability for a large-scale setting. I identify optimal steps models to be especially important because their foundation is motivated by social psychology and biomechanics. Aside from my modeling perspective, optimal steps models are attractive with respect to computational costs.

In Chapter 3, I discuss the (neuroscientific) justification, interpretation, usage, and importance of navigation fields in the context of large-scale simulations. I show that many models rely on navigation fields, and others can benefit from them. Consequently, efficient navigation field computation affects the computational performance of many pedestrian models. Models can not neglect large- and medium-scale wayfinding, and dynamic navigation fields can serve as a robust model.

Part II: In Part II, I change the focus from modeling to large-scale simulations. Since clock frequencies of single central processing units (CPUs) no longer increase significantly, introducing parallelism is essential. In Chapter 4, I identify pedestrian dynamics as a partly complex (many homogeneous objects) and partly complicated (inhomogeneous subjects) problem. To accelerate large-scale simulations, we have to focus on the computationally expensive and complex part of the problem: the operational level, i. e., microscopic locomotion. However, I firmly believe we should not compromise our modeling ideas for the sake of computational efficiency. And we do not have to, since pedestrian dynamics is a naturally parallel process. Therefore, locomotion models should promote data independence without any compromise. In Section 4.3, I review different parallel locomotion models and extract algorithms and data structures of their implementation. This overview should be helpful for developers that want to accelerate their model.

In Chapter 5, I exploit the agents' independence efficiently to accelerate optimal steps models. I advocate against parallelization via model changes, because the model would lose essential properties. My algorithm (ParallelEventDrivenUpdate) introduces parallelism to optimal steps models and is especially suitable for single instruction multiple data architectures. I present the degree of parallelism ParallelEventDrivenUpdate achieves. Additionally, I compare computation times of ParallelUpdate and ParallelEventDrivenUpdate, both executed on a graphics processing unit (GPU). ParallelUpdate in a former attempt that compromises optimal steps models for the sake of parallelism and acts as a baseline for the performance evaluation.

Part III: In Part III, the last part, I discuss the efficient computation of large-scale static and dynamic navigation fields. Navigation fields are the solution of eikonal equations computed on some space discretization. Therefore, this part of my thesis might also be interesting for readers outside of the pedestrian community interested in eikonal equation solvers or mesh generation. I identify two ways to reduce the computation time required to compute navigation fields: First, I enter the area of computational geometry and consider different space discretizations.

In the introductory chapter, Chapter 6, I discuss state-of-the-art mesh generation methods for pedestrian dynamics. I advocate unstructured triangular meshes because they conform to any desired geometry with localized resolutions – an important property for accuracy (of simulation results) and the reduction of discretization points to improve performance.

In Chapter 7, I introduce and analyze the meshing algorithm DistMesh which was designed by Persson and Strang [221]. DistMesh generates high-quality unstructured meshes but requires some adaptation to work robustly for geometries used in large-scale pedestrian dynamics.

Therefore, I develop an extension called EikMesh (Chapter 8). EikMesh inherits the force-based mesh improvement technique from DistMesh but avoids computationally expensive computations of Delaunay triangulations to improve the run time of the mesh generation process in a large-scale setting. Additionally, I introduce new mesh operations to deal with complex geometries defined by planar straight-line graphs.

Chapter 9 finally combines mesh generation and numerical methods to solve the eikonal equation, and thus to compute static and dynamic navigation fields. The chapter starts with another introductory part, where I discuss state-of-the-art methods (Sections 9.1 to 9.3) to solve the eikonal equation on Cartesian grids and unstructured triangular meshes. I evaluate the efficiencies of techniques concerning the spatial domain and travel speed function of the eikonal equation. This deep dive into numerical methods offers me and, hopefully, the reader a fundamental understanding of the problem on an algorithmic level. In the remaining sections, I show the importance of a localized mesh resolution. I develop a novel iterative eikonal solver that uses adaptive mesh refinement to achieve accurate results while keeping the number of mesh points small. In Section 9.5, I develop the InformedFastIterativeMethod, another eikonal solver that is specialized to compute dynamic navigation fields. It exploits similarities of consecutive solutions of eikonal equations, requires a minimal amount of workload, and still promotes parallelism.

In the last chapter, I conclude my thesis and discuss future works.

1.3 What is new?

This thesis focuses on efficient algorithms to enhance simulations based on existing microscopic simulation models. No new model is proposed, but I suggest improvements for optimal steps models at the end of Section 2.2.6 and introduce navigation fields for the Behavioral Heuristics Model (Section 3.4). I analyze the importance and suitability of different models in a large-scale context (Part I). The novel algorithms contained in this thesis incorporate known techniques. Whenever I integrate a well-known algorithm or idea from another contribution, it is indicated in the text and highlighted by its reference.

I analyze which level of the hierarchical modeling approach offers the most potential to improve simulations' efficiency (Sections 4.1 and 4.2). I review existing parallelization techniques

(Section 4.3) and combine and adapt well-known data structures and algorithms to parallelize optimal steps models by ParallelEventDrivenUpdate (Chapter 5). Subroutines of the algorithm are based on existing techniques. In combination, I establish **a new parallel algorithm for large-scale microscopic simulation using optimal steps models**.

In Part III, my main contributions are **a new meshing algorithm** called EikMesh, **an iterative eikonal solver based on mesh refinement**, and the InformedFastIterativeMethod, that is, **a new numerical method to solve consecutive and similar eikonal equations**. The new meshing algorithm EikMesh (Chapter 8) combines existing techniques. Its force-based improvement strategy is borrowed from DistMesh [221] (Chapter 7). The construction of the element size (Section 8.6.1) and distance functions (Section 8.7) and the mesh element ordering based on a space-filling curve (Section 8.5) are known methods integrated into EikMesh. New is the smoothing of element size functions accomplished by ElementSizeConstruction (Algorithm 13) and, most notably, **a combination of local mesh operations for mesh improvements**. Consequently, EikMesh supports geometrical constraints (Section 8.4), avoids Delaunay triangulation computations (Section 8.2), and offers a localized memory order of mesh elements (Section 8.5). Compared to DistMesh, the time complexity is reduced, and its parallel potential increased. Additionally, EikMesh generates meshes of higher quality for all test cases.

The **iterative eikonal solver** developed in Section 9.4 is a novel approach suitable for reducing the mesh size while achieving accurate approximations of the eikonal equation's solution. Furthermore, InformedFastIterativeMethod (Section 9.5) is **a new adaptation** of the well-known FastIterativeMethod designed **to compute dynamic navigation fields**. InformedFastIterativeMethod executes well-known subroutines to solve the eikonal equation on an unstructured mesh (Sections 9.2 and 9.3).

1.4 Advise to the reader

Whenever I use the first person "I", I refer to myself and give colleagues credit by referring to their publications. In the explanatory parts of my work, I use the personal pronoun "we" to refer to myself and the reader to include the reader into my thoughts. Sometimes I use "we" to refer to the research community, but it should be clear from the context which of the two meanings I use. By the term *agent*, I refer to a simulated pedestrian, and whenever I talk about *pedestrians*, I refer to real human beings.

I expect to confront readers with diverse knowledge bases and motivations. Therefore, I summarize each chapter at the end. Readers familiar with the chapter's topic might want to read the

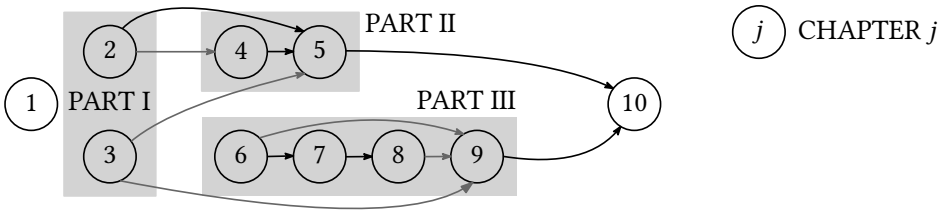


Figure 1.2: Dependencies among all chapters: black arrows indicate strong and gray arrows weak dependencies.

summary before they decide to continue. Readers from pedestrian dynamics might want to skip familiar sections of Part I. Readers interested in mesh generation for pedestrian dynamics might focus on Part III. Those interested in the large-scale aspect of pedestrian locomotion might want to concentrate on Part II. And researchers searching for new methods to solve the eikonal equation might find interesting new ideas in Chapter 9. Figure 1.2 illustrate the dependencies among all chapters. I choose a level of detail for the algorithm descriptions that facilitates implementation. Developers should be able to extract and use ideas for their own needs. In sections titled "Source code", I refer to my implementation.

1.5 Infrastructure

My research required extensive programming to implement novel and state-of-the-art algorithms and to generate, analyze, and visualize results. Implemented algorithms can be found in the Git [97] repository of the open-source Vadere project [7, 294]:

https://gitlab.lrz.de/vadere/vadere.

The repository also contains code to measure mesh element qualities, and to transform Vadere scenarios and unstructured meshes into TikZ [281] code. Via the TexStudio [5], TikZ [281] code is compiled into vector graphics. As a development environment, I used the community edition of IntelliJ IDEA [1], a customizable editor written in and for the development of Java projects. I also used Java libraries such as the Lightweight Java Gaming Library [195]. These libraries and other software-specific information are listed in the Maven [4] project files of the Vadere project. For the purpose of visualization, I made use of popular Python libraries including Pandas [314], NumPy [113], SciPy [296], Matplotlib [131], and Seaborn [310]. All other figures are generated by hand using the Ipe extensible drawing editor [2]. For the document itself, I used TexStudio combined with the software system for document preparation called LaTeX using the MacTex [3] distribution. Without these excellent software packages, my research would not have been possible. We often take these beautiful technologies for granted, but behind the million high-quality code lines are people who value the free spirit of creative work. I am deeply grateful for the effort they put into their work.

If not indicated otherwise, computations are carried out on my workstation: Intel i5-7400 Quad-Core (3.50 GHz), 8 GB DDR4 SDRAM, and a graphics card NVIDIA GeForce GTX 1050 Ti / 4 GB GDDR5 VRAM.

PART I

Navigation field-based microscopic pedestrian modeling

CHAPTER 2

Microscopic pedestrian modeling

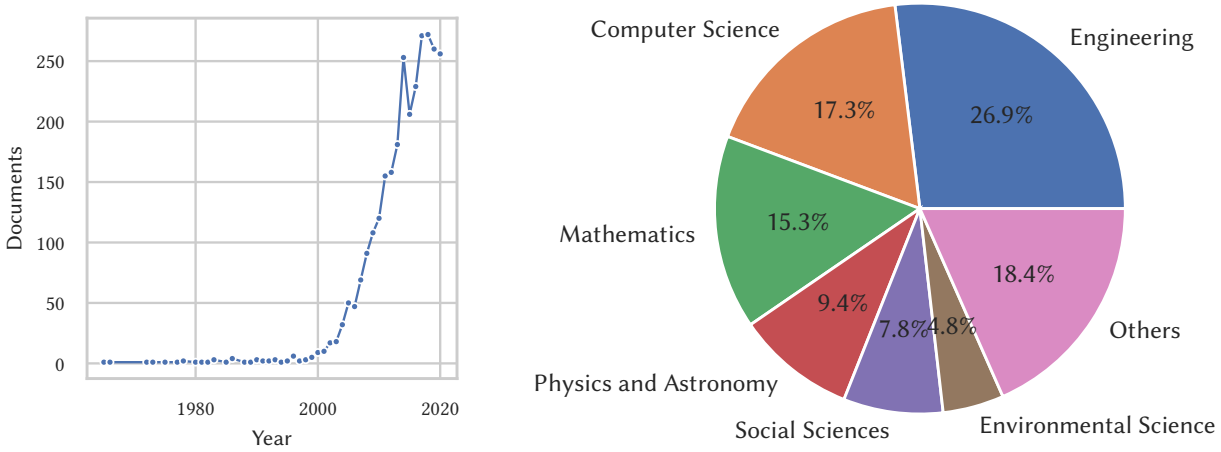
"Men must live and create. Live to the point of tears."

– Albert Camus

Modelers take into account a large variety of influences on pedestrian behavior. This becomes evident if one lists some of the principles modelers used in the past and will continue to apply. From social forces, Newtonian mechanics, queuing and game theory, computational geometry, velocity obstacles, fuzzy logic, social identity, to the personal space theory, ideas, and techniques influence the development of modern microscopic pedestrian models. The wide variety of publication areas, depicted in Fig. 2.1, is another indicator of the field's complexity. Thus, the exact nature of microscopic pedestrian behavior is open to debate, and the scientific discourse will continue. I expect that the number of models will further diversify. However, it is possible to filter and analyze some core concepts. This chapter is my attempt to do so.

I give an introduction to the landscape of microscopic pedestrian models. I start by explaining the well-established hierarchic model approach. It structures different aspects of decision making into different levels and is well-accepted by the overwhelming majority of the pedestrian dynamics community. After discussing the hierarchic model approach, I review different locomotion models that are of high relevance in microscopic crowd simulation. The aim is to give the reader an overview of the concepts and techniques and how modelers realize them algorithmically. The review is neither complete nor extensive but includes all major milestones of the development. I bypass minor model differences and details in favor of a well-rounded and compact discussion.

Since I am especially interested in large-scale pedestrian simulation, I discuss the computational complexities of the models. Insights from this analysis reveal the main leverage to accelerate large-scale pedestrian simulations and motivate my contribution. An experimental study would require access to all model implementations. Instead, I base my analysis solely on model definitions. Even if I had access to each model implementation, a comparison based on simulation times could be misleading. A fair comparison would require a highly optimized code base and the usage of efficient algorithms and data structures. Furthermore, it would be desirable that each implementation uses the same programming language and is compiled by the same compiler. In the end, such an experimental study requires a lot of effort and knowledge, mistakes can be easily



(i) (ii)

Figure 2.1: Scopus search result for the term "pedestrian dynamics" on the 22nd of November in 2020; overall 2,876 documents are listed: (i) publications per year and (ii) publications per subject [157].

made, and results give minimal insights. Therefore, I focus on the model definition and different model types.

2.1 The hierarchic model approach

Blumberg and Galyean [32] suggested to model the decision-making process and behavior of pedestrians (and other behavior-based autonomous creatures) by a layered architecture. The concept of a hierarchically structured decision-making process was incorporated and further developed by the pedestrian dynamics community. To my best knowledge, every microscopic pedestrian model supports or at least motivates a hierarchical decision-making approach. Blumberg's and Galyean's concept was motivated by a software architectural perspective. Later, it was used and further popularized by Reynolds [234]. Within the pedestrian dynamics community, Hoogendoorn and Bovy [126] suggested a similar hierarchical approach to model the decision-making process and behavior of pedestrians. Figure 2.2 illustrates this three level approach. It may or may not be inspired by Reynolds.

Hoogendoorn and Bovy suggested that, at the *strategic level*, pedestrians decide what and when they want to achieve a particular goal. These long-term goals range from visiting the toilet, buying some food, or leaving a building as fast as possible. The goals of pedestrians comprise their motivation. They might visit a concert and want to listen to music. Maybe pedestrians want to stroll around, or they are in a more organized setting like a demonstration. A model of the strategic level prioritizes a set of activities over another set. Examples are listed in [265, 316, 149].

At the *tactical level*, models split goals of the strategic level into an ordered set of necessary actions. Because researchers are interested in pedestrian movement, a specific logic translates

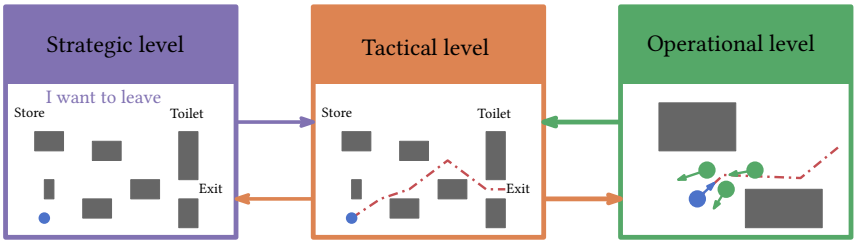


Figure 2.2: The three-level model approach suggested by Hoogendoorn and Bovy [126].

these actions to a set of immediate spatial destinations. Some models apply routing algorithms, such as [92, 123, 161, 162, 160], others introduce non-graph concepts, such as the application of the floor-field method introduced by [156, 115, 174, 116, 165]. To leave a building as fast as possibles – a strategic decision – one might have to leave a room first. Then, one might go to the staircase, which leads the person down to the ground floor, where he or she can exit the building using the closest door. One might also consider avoiding crowded areas by using faster detours. A father or mother may want to find a missing child before leaving [304]. Sometimes pedestrians might even conform to a specific socially expected behavior like queuing. Apart from the pedestrians' motivation, external motivations can also influence the behavior of crowds. Such external motivations range from authorities like the police or the service staff of large-scale events to signs and other notifications pedestrians receive. Furthermore, the knowledge base and awareness of pedestrians differ. They may or may not be familiar with their surroundings. And when intermediate destinations are not in sight, the diversity of knowledge and awareness is especially influential [15].

The actual walking from a start to an endpoint is part of the *operational level*. Distances for which agents operate according to the operational level vary from hundreds of meters to less than one meter. Models on the operational level range from cellular automata [40, 323, 255, 307, 81, 239], force-based models [118, 178, 48, 187], velocity-based models [142, 288, 68], heuristic approaches [59, 261] and optimal steps models [257, 258, 260, 300, 301]. At this lowest level, pedestrians make in-the-moment decisions. For example, they already know the floor or street to go through, but they adapt their navigation on the fly. Different motivations drive pedestrians' decisions, such as collision avoidance with other pedestrians and obstacles, following others, and moving as a group.

Hoogendoorn and Bovy model pedestrians as *homo economicus*, that is, they assume pedestrians maximize some utility. The expected utilities at lower levels influence choices at higher levels. Furthermore, choices at higher levels, condition choices at lower levels [126]. The information available to pedestrians might be imperfect, wrong, or incomplete.

In the past, modelers considered the decision-making process at the strategic and tactical level to be exogenous to pedestrian simulation. Research from the social science community (sociology & psychology) was required [247]. Therefore, it is no surprise that many of the early microscopic pedestrian models, such as [118, 16, 257, 68], are only concerned with the operational level. They only include physical aspects of the navigation process. Note that I follow the notation of [256] and distinguish between walking as a physical process and mental activities, like decision making, even if all thought can be traced back to chemical reactions. Since modelers first excluded the strategic and tactical level, they designed simple laboratory experiments to validate

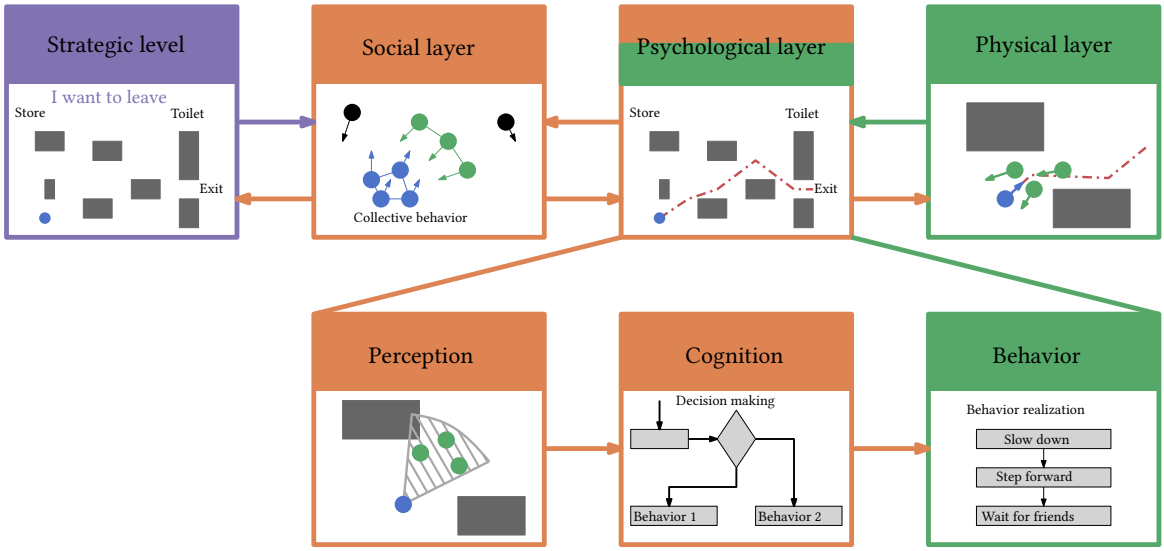


Figure 2.3: The hierarchical model approach suggested by Kleinmeier et al. [158]: Seitz [256] transformed the tactical and operational levels of Hoogendoorn and Bovy [126] into the social, psychological and physical layer. Kleinmeier et al. [158] give a more accurate definition of the psychological layer. They introduce three sub-layers: perception, cognition and behavior.

the operational level, that is, locomotion models. Therefore, their design eliminated or at least reduced the decision making required by participants. In such a design, the experiment setup defines all the next intermediate destinations. Compare, for example, experiments described in [328, 329]. The simplicity of laboratory experiment setups is not necessarily a drawback. Since pedestrian dynamics deal with the most sophisticated organism on earth, the human being, researchers have to reduce complexity to test hypotheses effectively. Nowadays, many authors such as [157, 148, 261, 303, 305, 262, 166, 302, 158, 273, 284, 285] argue for an invitation of the social sciences to help to understand and model human behavior at all levels. Currently, modelers are beginning to shift their focus to the strategic and tactical levels. Different informal definitions of the three-level hierarchy are listed in the literature. Therefore, it is debatable which model or model part is strategic, tactical, or operational. For example, it is not obvious what counts as a chosen intermediate destination and what becomes an intermediate destination due to the path that emerges from the decisions made by the operational level.

Seitz [256] expanded on the three-level hierarchy of Hoogendoorn and Bovy by splitting the tactical and operational levels into three different *layers*, compare Fig. 2.3. He was more concerned with the different spheres of human nature: the *social layer* covers various aspects that build on the *psychological* and *physical layer*. As humans, we are social, psychological as well as physical beings. Seitz argues that our social "body" influences our behavior, including our movement patterns. For example, the well-documented phenomena of sub-groups [134, 50, 17, 274, 202] is a social behavior. Therefore, models for sub-groups are part of the social layer. Note, however, that the social layer covers various aspects that build on the psychological and physical layer [256]. The psychological layer is responsible for the generation of movement decisions. These decisions determine the position where the agent wants to step next. The physical layer models how pedestrians move physically, that is, how physical forces move the human body towards the

next position. This layer is closely related to the operational level of [126] but emphasizes that physical motion is a biomechanical process that includes the whole three-dimensional human body. Seitz describes two locomotion models of the physical layer.

Kleinmeier et al. [157] further expanded on the ideas of Seitz. They reconsidered Seitz's psychological layer. The authors argue that on this layer, there should be more than the generation of movement decisions. In the end, researchers in the field of pedestrian dynamics are mainly interested in the movement pattern of pedestrians and not their mental status or process. Therefore, we tend to overlook the fact that mental processes of perception and cognition affect the behavior of pedestrians, and consequently, their movement pattern. One might argue that our abstract models should not consider the mental process because the level of detail would be to high. I think that we should not model each muscle of a person, because it is too computationally expensive, we would require unavailable information (the exact muscle structure of a person), and it contributes very little to the actual question we ask. This argument is also valid for the mental process if our goal is to model each thought of a person – it is just not possible. Instead, choosing the right abstraction of the mental process should be our goal to achieve realistic simulations for complex situations. The authors in [157] show that many well-established models fail to reproduce even simple scenarios because agents lack collective cooperation – an ability at which humans excel. They integrate three additional layers into Seitz's psychological layer:

(1) *perception sub-layer*: models the human perception, for example, vision,

(2) *cognition sub-layer*: models the processing of the information from the perception sub-layer and additional information about the vicinity like other agents. This process might lead to a change in behavior,

(3) *behavior sub-layer*: models the actual behavior by choosing an ordered set of actions which are realized by the locomotion model, that is, the operational level.

Figure 2.3 illustrates the hierarchic approach suggested by Kleinmeier et al. [157]. I should note that in 1995 Blumberg and Galyean [32] already considered to model perception and cognition. Furthermore, the idea is well-known in the field of robotics. Seitz notes:

> *"[Robots] perceive the environment through sensors, make decisions according to some rules, and the robot's mechanic carries out actions."* – Seitz [256]

The idea that agents move according to different behaviors furthered the development of each hierarchical structure. Early models on the tactical and strategic levels solely modeled the path planning of pedestrians. For each development step, the hierarchy became more and more concrete to fit the need for a broader set of different behaviors. Because pedestrian dynamics is a research area that attracts and needs scientists from many different research areas [157], hierarchical model approaches not only reflect the human decision-making process but also improve collaborative and interdisciplinary work. Since modular software architectures also reflect the hierarchical structure of the model, experts in, for example, psychology can implement their models at a specific layer without worrying about other parts of the software [158]. In the future, we, as the pedestrian dynamics community, might be able to reach a consensus for one specific

standardized hierarchical approach. Such a standard would undoubtedly improve collaboration further and would enable researchers to use their expertise effectively. Moreover, comparing different models would become more comfortable, efficient, and convincing. I conclude that the hierarchical approach is well-established and accepted by the majority of the pedestrian dynamics community. And I expect it will dominate the modeling and software development in the future.

2.2 Locomotion models

In this section, I discuss different well-known model approaches of microscopic locomotion models. They are all part of the operational level introduced in the previous section. One important characteristic they all share is that agents are driven by their spatial next intermediate destination. This seems obvious, because we want to model crowd movement but nonetheless, we have to keep in mind that the choice of the destination and how the driving factor is modeled influences the agent's movement fundamentally. For this review, I exclude specialized model parts such as how to model doors, elevators or stairs and focus on the basics, that is, obstacles, agents and spatial destinations. Listing every single model would lead to a repetitive and uninspiring discussion. Instead, I review models that have a specific set of distinct characteristics and are the pillars of the modeling process. Furthermore, I analyze the computational complexity of the model based on its definition.

2.2.1 Cellular automata

One of the well-known and well-studied models are cellular automata (CA) models [317, 318]. Their name originates from the principle of *automata* occupying *cells* according to localized neighborhood rules of occupancy. Blue and Adler [31] fittingly lay out the motivation to use CA models:

> *"The attractiveness of using CA is that the interactions of the entities are based on intuitively understandable behavioral rules, rather than performance functions."*
> – Blue and Adler [31]

If we interpret the above statement in the context of pedestrian dynamics, it suggests that complex crowd behavior emerges from simple rules that each individual follows. In this sense, CA models are complex systems. Complex behaviors are challenging to understand, but CA models promise that we can observe a simple (and finite) set of rules that reproduce real-world complex human behavior.

CA models are discrete in time and space. Their domain is a possibly infinite grid of equally shaped and sized cells. Therefore, the domain is represented by a highly regular structure. The geometry definition partitions the cells into reachable and unreachable cells. Unreachable cells model obstacles. A cell can be in one of a finite number of states. This state is updated for each cell and for each discrete time step. State changes of cells also model the movement of agents. If for time step k, an agent l moves from cell i to cell j, cell i changes its state from occupied

to unoccupied. Cell j becomes occupied. For most CAs, cell j has to be unoccupied at time step $k-1$ because a cell can only contain one agent. Many CA models are probabilistic. They use randomness as a shortcut to resolve conflicts – multiple agents competing for the same cell – or a probabilistic state transition function.

The development of CA models in pedestrian dynamics began with Gipps and Marksjö [96]. They proposed a model very similar to cellular automata. It uses reverse gravity-based rules to move agents over a grid of cells. Blue and Adler [31] introduced the first CA model for crowd simulation. CA models for traffic simulation, such as [249, 244] inspired the model. But instead of streets, the CA models corridors. Like in traffic simulations, agents move on lanes. Each time step consists of three sub-steps, each executed in parallel for all agents: (1) change lane, (2) step forward, and (3) compute gap. Agents can step forward up to m cells, where m relates to the agent's free-flow velocity. If they are blocked, they can also sidestep to another lane. The model defines the cell length to be 0.457 m, such that an agent occupies approximately 0.21 m^2.

Schadschneider [245] introduced a CA model that uses a floor field as a memory to add long-ranged interactions. Probabilistic transition functions guide agents. They can move to any cell of their current Moore neighborhood. The CA resolves conflicts probabilistically. A *floor field* – the second grid of cells containing real values – modifies the transition probability of local transition rules, such that, moves towards the direction of large field values are preferred [245, 155]. A *static floor field*, which does not evolve with time, is used to define more attractive regions. The second, *dynamic floor field*, represents the virtual trace left by agents. Similar to chemotaxis, virtual traces guide agents through the spatial domain. After an agent leaves a cell (i, j) the field value is increased by $\Delta D_{i,j}$. A diffusion and decay process, defined by

$$\frac{\partial D}{\partial t} = \lambda \cdot \Delta D - \delta \cdot D, \tag{2.1}$$

reduces the values of the floor field. In the equation, λ is the diffusion and δ the decay constant [245, 20]. In [40], the model is used to simulate the evacuation of a room and a lane formation scenario. Kirchner et al. [152] extended this concept to support a probability factor in the diffusion and decay functions to model different behaviors what they call "regular", "panic", or "herding". Later Bandini et al. [19] used the floor field to model the action-at-a-distance behavior and [172, 169] extended the model and introduce agents that move according to different velocities.

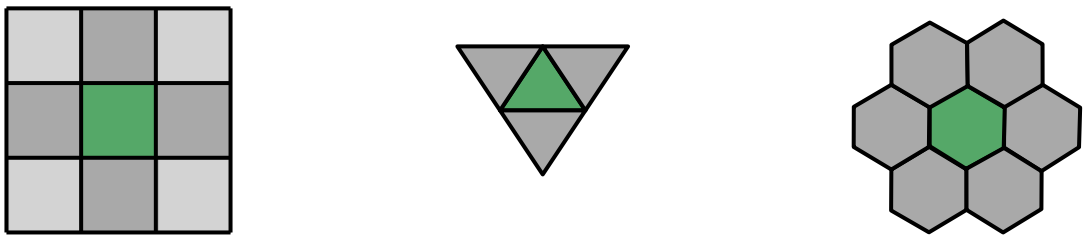
Kirik et al. [156] presented another model approach of long-range interactions. Instead of using an additional dynamic floor field, they suggested dynamics should influence the floor field that directly navigates agents towards their destination. The new dynamic field no longer represents the distance but the travel time required. Crowded areas require more time to travel through, and agents choose the fastest instead of the shortest path. One year later, Hartmann [115] suggested a similar approach that was based on the solution of the eikonal equation. Using the eikonal equation as a mathematical framework for navigation was introduced by Treuille et al. [291] four years earlier. The technique gained attraction and was explored by many modelers, for example, [173, 174, 165]. In Chapter 3, I discuss this technique in more detail. Part III of this thesis, introduces efficient algorithms to compute navigation fields for microscopic continuous space models.

Since their introduction, static floor fields have been a common technique to navigate agents towards a destination. They depend on the scenario and have to be initialized accordingly. Since

they influence the agents' behavior fundamentally, one might argue that static (and dynamic) floor fields contradict the statement made by Blue and Adler. Interactions are not only based on *"understandable behavioral rules"* but also on floor fields, modeling the geometry (*static*) and long-range interactions (*dynamic*).

Many additions to the standard concept of CA models were made, adding more and more complexity. In fact, many CA simulation models are cellular network models (CN) (compare [33, Chapter 7]) rather than cellular automata in the strict sense. Bandini et al. [20] modeled the vision of pedestrians by introducing the *observation fan*. Sequential and other update schemes, different from the parallel update, were introduced to take specific interactions into account. Was et al. [307] introduced *social forces* according to the personal spaces theory of Hall [111, 112]. These forces push standing agents away from the social space of others. Furthermore, they use smaller cell sizes and a different grid topology. High-resolution grids allow agents to occupy more than one cell [243, 307, 308]. This technique allows for smoother agent movement and higher maximal capable densities. Kirchner et al. [154] concluded that by choosing a high grid resolution, the models fall, not only in another model category but macroscopic measures of the simulation outcomes, like density and flow, are different. They noted that in the limit (*cell size* $\rightarrow 0$), the discrete space of the model becomes continuous. Therefore, it might be possible to compare the model with continuous models, which I describe below. Their investigation revealed that: on the one hand, the flow increases with a higher grid resolution. On the other hand, agents tend to block each other more frequently [154]. In summary, simulation outcomes are influenced by the cell size, the shape of cells, and the topology of the grid. The shape size and topology determine the number of neighbors (3, 4, 6, 8) of each cell. Maniccam et al. [196] found that, when the drift of moving objects is high, the critical density of transition between freely moving and jamming is higher for hexagonal cells than for squares. Sarmady et al. [243] introduced an even higher grid resolution. They decide to use a cell size of $0.025\,\mathrm{m}^2$.

CA models have been successfully applied to modeling and simulation of single-, bi-, and four-directional pedestrian flows [31, 40] as well as collective phenomena [246, 153]. The list of models in this section is not complete but captures the main features and differences of cellular automaton models. Some CAs are deterministic, some stochastic; some use floor fields to guide their agents and to introduce long-term interactions, and some rely only on local interactions. Cells are either rectangular, triangular or hexagonal.



(i) (ii) (iii)

Figure 2.4: Cell shapes and CA topologies: (i) a rectangular grid with a four- or eight-neighborhood, (ii) a triangular and (iii) a hexagonal grid a three- and six-neighborhood, respectively.

They have in common that they consist of many identical components, each simple, but together capable of complex behavior [318]. The basic rules of cellular automata are simple, thus easily understandable. There is no specialized knowledge required. Implementing them requires not much effort because they mostly rely on highly regular data structures. One can represent the grid of cells, the static and dynamic floor fields by multiple two-dimensional arrays of primitive data types. Compared to continuous space models, CAs are significantly less computationally expensive, because many calculations, such as collision tests, are inexpensive. Due to their regularity, many CAs scale very well. For example, [309] executed their CA on the GPU (a massively parallel architecture). If computational power is an issue, they offer excellent results for many scenarios, even for slow hardware systems. However, the spatial discretization of cellular automata leads to unwanted artifacts [96, 115]. High-density scenarios, which are of great interest, are difficult to reproduce using CA models. Continuous motion is, by definition, not observable. By using higher grid resolutions, one reduces these unwanted effects but also increases the computational power needed. Consequently, instead of using higher and higher resolutions to fix the downside of space discretization, one might switch to a continuous space model.

2.2.2 Force-based models

Force-based models are continuous in time and space. They change the position of their agents by applying forces to their bodies. This concept assumes that a force field guides changes in the movement of pedestrians. As Chraibi et al. stated:

> *"[Force-based models] are motivated by the observation that motion of pedestrians deviates from a straight path in the presence of other pedestrians. Therefore, their motion is accelerated which, according to Newton's law, implies the existence of a force."*
> – Chraibi et al. [49]

In general, force-based models follow Newton's second law

$$m \cdot \ddot{\mathbf{x}} = F, \tag{2.2}$$

where m is the mass of an object positioned at $\mathbf{x}$ accelerated by a force F. The force F is the superposition of attractive and repulsive effects. Let $\mathcal{W}$ be the set of obstacles and $\mathcal{A}$ be the set of agents. Then the following equation of motion formally defines the movement of each agent

$$m_l \cdot \ddot{\mathbf{x}}_l = F_{l,\Gamma} + \sum_{j \in \mathcal{W}} F_{l,j} + \sum_{k \in \mathcal{A}} F_{l,k}, \tag{2.3}$$

where m_l is the mass of agent l, $\mathbf{x}_l$ its position, $F_{l,\Gamma}$ the force driving it towards its destination Γ, and repulsive forces $F_{l,j}$ pushing it away from obstacle j and $F_{l,k}$ pushing it away from agent k [49]. The smooth acceleration and deceleration translate to continuous and smooth motion. Where CA models allow agents to change their direction abruptly, force-based acceleration prohibits sharp turns and sudden changes of the agents' velocity vector.

The fluid crowd modeling method introduced by Henderson [120] was the starting point of force-based models. He presented a theory of the flow of pedestrians along a channel. However,

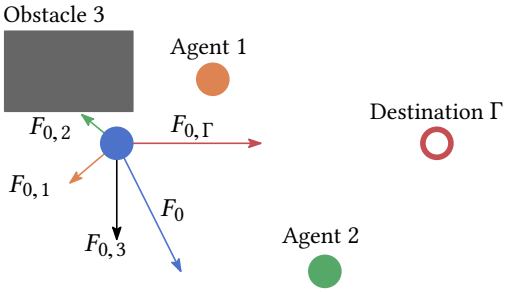


Figure 2.5: Force-based model concept: the obstacle 3, agents 1 and 2 repulse the left agent 0 while the destination on the right attract it. The net force F_0 pointing to the bottom right.

the development of actual force-based models started in Japan in 1975, inspired by the motion of a shoal [122]. Four years later, Okazaki [212] proposed a model inspired by forces between magnets.

If we discuss force-based models today, we usually refer to social force models. Their name emphasizes that forces are caused not by physical but "social" phenomena. The concept is inspired by Lewin's idea [185] to explain crowd motion by "social fields" and is strongly connected to floor fields introduced in the field of cellular automata. For example, in 1985, Gipps and Marksjö [96] already mentioned the concept of repulsive forces between pedestrians. In the gaming and animation field, force fields were also known and applied to simulate flocks of birds [233]. However, Helbing and Molnar [118] introduced the first social force model in 1995, and they called it Social Force Model. As a side remark, in 2010, Löhner et al. [190] presented a very similar force-based model, which I will not discuss separately. The model of Helbing and Molnar defines a primary force

$$F_\Gamma = \frac{1}{\tau}\left(v_0 \cdot \mathbf{e}_\Gamma - \dot{\mathbf{x}}\right), \tag{2.4}$$

that drives agents towards their next (spatial) destination. v_0 is the agent's free-flow velocity, $\mathbf{x}$ its position, $\mathbf{e}$ the unit vector representing the direction pointing towards the destination, and τ the agent's reaction time. Acceleration is defined by

$$\dot{\mathbf{v}} = F + F_\epsilon \tag{2.5}$$

where $\mathbf{v}$ is the unrestricted velocity induced by the sum of all forces F acting on the agent [118]. Some force fluctuation term F_ϵ takes random variations into account. The actual velocity $\dot{\mathbf{x}}$ is limited to some maximum speed $v_{\max}$

$$\dot{\mathbf{x}} = v(\mathbf{v}) = \begin{cases} \mathbf{v}, & \text{if } \|\mathbf{v}\| \le v_{\max} \\ \mathbf{v} \cdot \frac{v_{\max}}{\|\mathbf{v}\|}, & \text{otherwise.} \end{cases} \tag{2.6}$$

Several problems become evident when one observes motion in social force models. Some of them can be mitigated. The following fundamental observation made by Seitz explains the source of most problems. He argues that

> *"in the end, the model's behavior is validated through its outcomes, that is, the crowd's movement. However, they lack a meaningful interpretation as a psychological process:*

> *social force models neglect the transition from decision making to actual pedestrian motion because they translate the psychological tension directly to physical motion."*
> – Seitz [256]

I agree with this view. The problem originates from mixing the mental process with its physical realization – there is no distinction between thinking and acting. At first glance, social forces model social and psychological aspects. For example, repulsive forces that act between agents, model the pedestrians' personal spaces. Or attractive forces model sub-grouping of pedestrians. However, agents do not evaluate the effect of these forces to make a decision. Instead, they act as if social forces are real. They decide nothing at all but passively realize the movement imposed by social forces as if they are physical. If we consider Newton's law, that is, acceleration implies the existence of a force, we fall into the trap of using social forces for acceleration. But the forces that move the human body are fundamentally different from the forces of social force models. Individuals use muscle power to step forward. There is no real attraction and repulsion effect induced by the destination and other pedestrians, respectively. I step towards my desired destination by muscle power because I want to – because I'm driven by social 'forces' but not because of social forces. A mental-biomechanic process controls my muscles while social motivations drive my decision making.

Chraibi et al. [49] listed some of the problems of the original Social Force Model and fixed or mitigated them by an extension: the Generalized Centrifugal Force Model (GCFM) which is a generalization of the Centrifugal Force Model (CFM) introduced by [326]. Problems can be understood if we consider Newton's laws of motion. By his third law, two particles interact by forces of equal magnitudes and opposite directions. This symmetry causes unrealistic symmetric behavior. Pedestrians are unable to evaluate the distance between them and others behind them. Therefore, a pedestrian is more repulsed by others in front compared to the ones behind. Therefore, Johansson et al. [326, 137] suggested a weighted force

$$w(\theta_{l,k}) \cdot F_{l,k} \tag{2.7}$$

which replaces $F_{l,k}$ in Eq. (2.4). $\theta_{l,k}$ is the angle between the normalized distance vector between agent l and k and the walking direction e_l of agent l. Johansson et al. [137] also noted that in reality, in addition to the angle $\theta_{l,k}$, the distance pedestrian keep from each other depends on their step size. And because the step size translates to speed, they introduce velocity-dependent anisotropic agent forces. This idea was further developed in the works of Chraibi et al. [49]. Another problem emerges due to of the superposition principle, that is, the assumption forces are additive. In dense situations, this leads to unnatural backward movement or high velocities [49].

Since the Social Force Model in [118] does not take the solidity of bodies into account, unrealistic excessive overlaps of agent bodies occur. The shell of the human body is, to some degree, elastic, and humans can adjust their skeleton, but this should only contribute to slight overlaps. Choosing a high value for the elasticity constant helps to avoid overlaps but introduces forces larger than 6000 N [178]. Lakoba et al. [178] also pointed out that even for simple scenarios, one has to choose unrealistic physical parameters to achieve a visually realistic behavior. They improved simulation results by using an explicit numerical integration scheme. They came closer to be able to use realistic parameters, but there was still a discrepancy. To avoid significant overlaps, they changed the parameters suggested by [119]. However, to achieve good results for a scenario

with many participants, they made social forces density-dependent. Köster et al. [167] introduced the Mollified Social Force Model, which allows the use of high order fast converging numerical solvers to increase the computational speed. The authors pointed out numerical pitfalls caused by discontinuity introduced by, for example, Eq. (2.6), and how to overcome them.

The last problem observed by [49] is that agents do not stop and keep moving independently of the actual situation. As a consequence, sharp directional changes are impossible. Because acceleration governs the velocity of agents, the movement of pedestrians tends to oscillate unnaturally. The GCFM mitigates oscillations and unnatural backward movement by varying space requirements of agents while in motion – faster-moving agents require more space. Pelechano et al. [219] avoided overlaps by a separated collision detection mechanism. Their background in computer graphics and animation motivated them to introduce a model (High-Density Autonomous Crowds) that generates realistic-looking movement patterns. Likewise, [217] proposed a self-stopping mechanism to prevent agents from continuously pushing over each other.

Force-based models successfully simulate pedestrian evacuation under stress [119], dynamic route choice [174], pedestrians walking in groups [202] and waiting pedestrians [139]. However, given one specific set of parameters, no force-based model can model all these scenarios. My experiences lead me to the conclusion that the more mental activity pedestrians invest in their movement, the less appropriate Newton's laws become. Therefore, forced-based models are appropriate to reproduce evacuations through a single bottleneck but challenging to use for more complicated situations. An unsolved problem of all force-based models is the multiple roles of the relaxation time τ in Eq. (2.5). It affects how precisely agents follow their preferred path, and at the same time, how they avoid collisions [138]. These two behaviors may not be correlated. Johansson et al. [138] suggests that it is up to the scientists to decide which artifacts can be tolerated given their specific research questions. I agree with that.

2.2.3 Velocity-based models

In this section, I discuss the second type of continuous space models: velocity-based models. Similar to force-based models, velocity-based models are continuous in space and time, and agents move according to a velocity vector. But in contrast, the velocity is not based on acceleration. Instead, velocity-based models change the agent's velocity vector more directly.

For velocity-based models, the first derivative of the agent's position $\dot{\mathbf{x}}_l$ relates to some speed

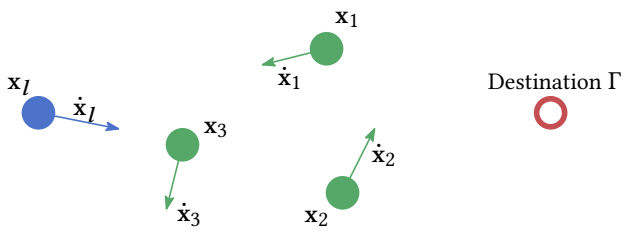


Figure 2.6: Velocity-based model concept: agent at position $\mathbf{x}_l$ adapts its velocity $\dot{\mathbf{x}}_l$ to avoid collision. Agent 2 and 3 only slightly disrupt agent l while agent 1 cause agent l to choose a detour to its destination t.

function v which depends on the positions (and possibly velocities) of the other n agents:

$$\dot{\mathbf{x}}_l = v(\mathbf{x}_l, \mathbf{x}_1, \ldots, \mathbf{x}_n) \cdot e(\mathbf{x}_l, \mathbf{x}_1, \ldots, \mathbf{x}_n), \tag{2.8}$$

where e returns a unit vector pointing towards the walking direction of agent l [289]. Note that v and/or e depend on the desired direction towards the agent's destination given by a strategic or tactical model.

One velocity-based approach relies on the so-called *optimal-velocity*. This concept was first introduced and studied in car traffic simulations [210, 315, 207, 288]. Tordeux et al. [289] brought it into the pedestrian dynamics community and proposed a collision-free minimal speed-based pedestrian model. In their model, agents adjust their velocity using the optimal velocity function v, which depends on the minimum spacing in front. Their model reproduces phenomena of self-organization observed in force-based models and field studies but requires less computation power.

The computer graphics community has proposed another type of model that implements a vision-based collision avoidance [216, 214, 142]. Karamouzas et al. [142] build on the idea of *velocity obstacles* [83, 295] and bears also some resemblance to the model proposed by Pettré et al. [222]. Agents of their model actively seek a free path through a crowd by calculating the required velocity changes to avoid collisions with other agents or obstacles. First, the set of the first n agents that are on a collision course of a specific agent is computed. In the second step, orientations and speeds, which lead to collision avoidance, are calculated. Then an optimal velocity is picked. There are some restrictions on the change of the velocity, but it can change more directly compared to force-based approaches – a distinct feature of all velocity-based models.

Shiller et al. [271] generalize the concept of velocity obstacles [83] to obstacles moving along arbitrary trajectories. Berg et al. [26] developed a collision avoidance method for multiple mobile robots by reducing the problem to solving a low-dimensional linear program. They called the model Optimal Reciprocal Collision Avoidance Model. Based on these ideas, Curtis and Manocha [57] developed extensions to model human behavior. They consider the idea that agents adapt their intent (chosen at the strategic or tactical level) to local conditions. As a side remark, Moussaïd et al. [203] use the concept of collision avoidance by actively changing the velocity and integrate it into a force-based model. The velocity is still adjusted smoothly by acceleration, but heuristic rules are used to choose the desired direction and speed of agents.

Dietrich et al. [68] introduced the Gradient Navigation Model (GNM), which is not mainly influenced by Newtonian physics. Instead, humans navigate freely based on internally driven motives. Similar to force-based approaches, the GNM is an ODE-based model. However, like for many floor field based CA models, agents steer directly towards the direction of steepest descent on a given navigation function. More precisely, the direction is defined at first order while the speed is of second order. Since the magnitude of the velocity relies on acceleration, the Gradient Navigation Model is not purely velocity-based but shares some mathematical aspects with force-based models.

2.2.4 Data-driven modeling and calibration

CAs, force-based, and velocity-based models are analytic approaches. Some combine analytical methods with heuristics and other decision-making techniques. The mathematical equations,

parameters, and rules are based on logical principles observed in physics, psychology, and sociology. Some model parameters might not be observable in the real world. However, these models can be calibrated by real-world data. Modelers apply the classical modeling framework:

(1) *observation*: observe the real world,

(2) *modeling*: use our observations to deduce an abstract model,

(3) *calibration*: calibrate your model based on real-world data,

(4) *validation*: validate your model, for example, find new phenomena in your simulations which you can also observe in the real-world.

However, some try a different path: the data-driven approach. They change their focus from logical principles and cause and effect to the data. Models are learned by using a massive amount of data, mainly video footage.

Lerner et al. [184] proposed a data-driven modeling approach based on real-word trajectory segments. They extract these segments from video data. During the simulation, agents query these real-world trajectory examples to extend their virtual trajectory. Zhao et al. [332] also relied on video data. But instead of using trajectories, they use the positions of pedestrians, their neighbors of the last 2 seconds, and their velocities. They trained a neural network classifier that selects a suitable real-world example to updates the agent's velocity. Musse and Thalmann [206] used a data-driven approach for pedestrian behavior in non-dense scenarios. They automatically extract trajectories and use them to learn the desired velocity of agents. To increase the realism of animated crowds, Eunjung et al. [141] use a database of formations and trajectory segments. Each animated agent selects a trajectory segment from the database to extend its moving trajectory. Their selection algorithms ensure that no collisions occur. Zhong et al. [334] presented a data-driven crowd modeling framework. They aimed to learn and reproduce macroscopic behavior patterns by learning the velocity field, which guides pedestrians towards their destination region. Their model is based on a dual-layer modeling architecture. They also learned behavior patterns from video data. Bera et al. [24, 25] presented a method to extract the dynamic behavior features of real-world pedestrians. They use the extracted footage to learn movement characteristics on the fly.

To this day, data-driven approaches are not yet competitive compared to classical models. They are not yet capable of dealing with the variety of scenarios that classic models successfully capture. However, this might change in the future. Furthermore, they are not only used to learn a model but also to calibrate it. In fact, calibration is always data-driven, but there are simple approaches, such as calibration against fundamental diagrams, and more sophisticated methods. One might argue that each distinct set of input parameters is in and of itself a different model. The area between data-driven modeling and (data-driven) calibration is gray. Therefore, I also want to mention more sophisticated calibration methods. Johansson et al. [137] were one of the first that proposed an evolutionary algorithm to calibrate the parameters of the Social Force Model. In [220, 251], the authors introduced optimization algorithms, such as genetic algorithms, to apply video-based calibration. Yamaguchi et al. [321] applied machine learning techniques to estimate model specifics and parameters of a force-based model. Wolinski et al. [319] compared three different microscopic pedestrian simulations models: the Social Force Model [119], Reynold's boids

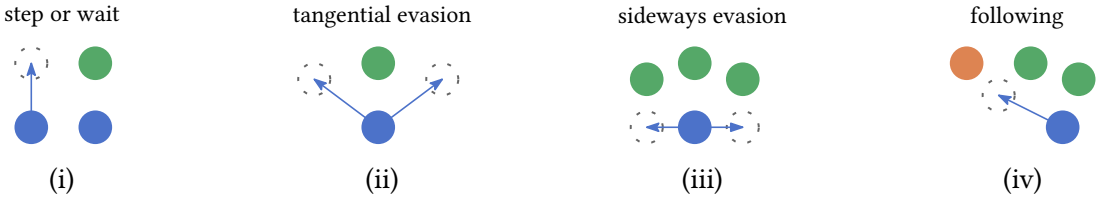


Figure 2.7: Behavior with four simple heuristics introduced by [261].

model [233, 234] and the Optimal Reciprocal Collision Avoidance Model [26]. Combining macro- and microscopic metrics with an evolutionary algorithm, the authors established a general framework for model calibration and comparison. Before comparing the models, they calibrate each model based on video data. Seer [252] also calibrated different models (different social force models, the Optimal Reciprocal Collision Avoidance Model, and different optimal steps models) based on trajectories recorded by the Microsoft Kinetic. Seer also developed a framework for evaluating microscopic pedestrian models. For calibration, he used two approaches. The first one called *model estimation by nonlinear least square methods* [127, 163] and the second one *comparison of real and simulated trajectories* [201]. The authors base their estimation method on an objective function $\xi(\omega)$, which compares the acceleration of multiple pedestrians and simulated agents at many points in time for a specific set of parameters ω. The method minimizes ξ to find suitable parameters ω. The second approach is similar to [319]. But instead of simulating all pedestrians, each individual is simulated separately. The remaining agents move on their observed (real-world) trajectory. Zhong et al. [333] proposed differential evolutions to calibrate parameters of pedestrian simulation models based on video data. Gödel et al. [99] introduced Bayesian inversion as a systematic method for the calibration of input parameters for microscopic pedestrian simulation models. One crucial input parameter of all the mentioned models is the set of spatial destinations of the agents. The choice crucially influences the simulation outcomes. The authors in [100] tried to learn this critical parameter, which can often not be observed directly by the video footage or other sensors. They use density heatmaps, which indicate the pedestrian density within a given camera cutout.

2.2.5 Cognitive heuristics models

This section discusses important microscopic models that do not fit in any of the previous categories but follow another idea: simple cognitive heuristics. These models are motivated by findings in cognitive science made by Gigerenzer et al. [95]. Antonini et al. [16] introduced a discrete choice framework for pedestrian walking behaviors. Agents choose their velocity direction according to several radial cones. They (1) slow down, (2) keep some speed, or (3) accelerate. Seitz et al. [261, 256] introduced a model that uses simple heuristics rooted in cognitive psychology, which I call Behavioral Heuristics Model (BHM). Moussaïd et al. [203, 200] motivated the work by suggesting a rule based approach. However, Seitz et al. argue against the required complicated numerical calculations because humans use efficient and straightforward rules that do not necessarily lead to the global optimum but yield good-enough results [261]. The authors point out that many models use expensive computational techniques like optimization, probability distributions evaluation, or force integration that are inadequate descriptions of cognitive

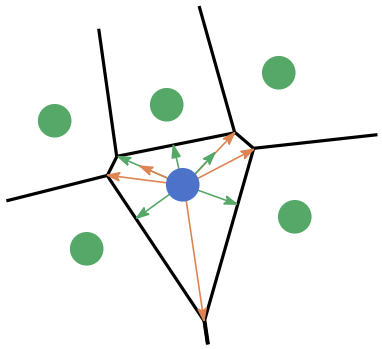


Figure 2.8: Direction choice heuristic introduced by [320]: green arrows indicate following directions while orange arrows indicate detour directions. Note that there may exist a detour direction that is identical to a following-direction.

processes. Instead, they suggest 4 simple heuristic behaviors that determine the movement of agents: (i) agents step forward or wait, (ii) they can evade tangentially if they are blocked, (iii) evade sideways, or (iv) follow others, compare Fig. 2.7. They emphasize that heuristics can help explain the cognitive effort connected to moving in a social environment. And they encourage the community to find and introduce new or better heuristics in the future. Xiao et al. [320] answered that call and extended the works of [261]. They refined the directional choices of agents. Based on the Voronoi diagram of all agents' positions, they introduced three heuristics for the choice of agent directions: (1) towards the destination, (2) to follow another agent, (3) to make a detour. Figure 2.8 illustrate this concept. On the one hand, the direction pointing to the Voronoi node corresponds to the intermediate space of two neighboring agents. Therefore, it is a possible detour. On the other hand, the direction perpendicular to the Voronoi edge corresponds to the neighboring agent and defines a possible following-direction. An optimization mechanism evaluates all possible detours and following-directions.

Cognitive heuristics should and will play a role in future model developments. However, separated from other techniques, they are not used to simulate complex scenarios. For example, Seitz introduced them on the *psychological layer* [256] and proposed the realization of the *physical layer* by optimal steps models or a force-based model that does not model the decision making with forces.

2.2.6 Optimal steps models

Optimal steps models define discrete and instantaneously executed footsteps that model the physical movement of pedestrians while "repulsive" and "attractive" potentials model the decision-making process. Even though Seitz and Köster [257] define repulsive and attractive potentials, there are no forces present. Instead, an optimization problem is solved to compute the next position of an agent [257]. Seitz and Köster used the term "potential" to enrich the reader's understanding but already cultivated the term *utility* to draw a line between optimal steps models and social force models. Optimal steps models all share the *optimal step principle*:

(1) agents move by a series of discrete footsteps

(2) to increase their utility defined by a utility field u.

Seitz and Köster [257] introduced the original Optimal Steps Model (OSM) in 2012. On that basis, the pedestrian dynamics group at the Munich University of Applied Sciences developed it further. Two years later, Köster and Zönnchen [165] introduced long- and medium-range interaction via dynamic navigation fields into the model. In [300], von Sivers and Köster refined the model by integrating the theory of (inter-) personal space. Simultaneously, Seitz et al. [260] were more concerned with walking as a biomechanical process and studied the stepping behavior of pedestrians. Zeng et al. [327] and Köster et al. [168] extended the *optimal step principle* for motion on stairs. Recently, Kleinmeier and Köster [158] investigated collective human behavior. Based on the optimal steps model introduced in [300], they showed how an existing locomotion model could be extended to model these behaviors.

In this section, I explain why optimal steps models are suitable for large-scale microscopic simulations. For the introduction of a parallel implementation in Chapter 5, I have to go into the details of the models. The aim is not to examine the content of each paper but to extract presented ideas and findings and show how researchers realized them algorithmically.

The natural stepping behavior

In contrast to many other models, optimal steps models capture the principle of human locomotion: bipedalism. As Seitz et al. remarked:

> *"The principle of human locomotion is bipedalism. Although this seems a trivial statement, it has largely been ignored in pedestrian and crowd simulations."* – Seitz et al. [256]

The spatial discretization of agents' motion defined by optimal steps models, reflects the natural stepwise movement of pedestrians. While in cellular automata, pedestrians are represented by cells and limited to move from cell to cell, optimal steps models represent pedestrians by the agents' position and extension in space. Like humans, agents move by performing discrete footsteps in the continuous space. Therefore, they adjust their movement direction only between each *footstep*. For each individual l, the most *valued* position contained in P_l determines its next position. Importantly, $P_l \subset \mathbb{R}^2$ is a subset of the space covered by the agent's step circle. The utility u models the pedestrian's assessment of locations. Areas close to other agents and obstacles are undesirable, while positions close to the agent's destination are desirable. Note that since the two feet of humans are not modeled explicitly, a *footstep* moves the center of the agent's body to the next best position. The concept of a region of interest situated in front of pedestrians and discrete choices is similar to the approach introduced by Antonini et al. [16]. Before we move on, I add some essential terms to avoid confusion:

(i) The *free-flow speed* v_l is the speed of pedestrian l in an open space with no incentives other than reaching a spatial destination.

(ii) The *desired speed* is the speed of pedestrian l in an open space, including all incentives, for example, slowing down for a friend to catch up.

(iii) The *desired step length* r_l is the step length of pedestrian l if its speed is equal to its *desired speed.*

(iv) The *step length* is the length of a performed footstep.

The vast majority of models assume that free-flow speed equals desired speed, and optimal steps models are no exception. In the following, I explain how the desired step length r_l of agent l is computed.

Agents draw their free-flow speed from a normal distribution truncated at $-\sigma_v\lambda_\sigma$ and $+\sigma_v\lambda_\sigma$:

$$v_l \sim \mathcal{N}_{\text{tr}}\left(\mu_v, \sigma_v, -\sigma_v\lambda_\sigma, +\sigma_v\lambda_\sigma\right). \tag{2.9}$$

Seitz et al. [257] follow the results found by Weidmann [312], who suggested $\mu_v = 1.34\,\frac{\text{m}}{\text{s}}$ and a standard deviation of $\sigma_v = 0.26\,\frac{\text{m}}{\text{s}}$. The authors choose $\lambda_\sigma = 2$, such that the cutoff $\sigma_v\lambda_\sigma = 0.52$ is twice the standard deviation. Therefore, free-flow speeds are in between $0.82\,\frac{\text{m}}{\text{s}}$ and $1.86\,\frac{\text{m}}{\text{s}}$. μ_v, σ_v and λ_σ depends on the modeled population and have to be calibrated for each specific scenario. Seitz et al. [257] derive the desired step length r_l of agent l by its established relation to the agent's free-flow speed. Experiments revealed a linear relation of the individual desired step length to the free-flow speed:

$$\mu_l(v_l) = \lambda_0 + \lambda_1 \cdot v_l, \tag{2.10}$$

with $\lambda_0 = 0.462$ and $\lambda_1 = 0.235$. Slightly different coefficients ($\lambda_0 = 0.38, \lambda_1 = 0.27$) have been found in [259]. Seitz et al. [257] proposed a quadratic term for faster speeds, for example, if pedestrians start jogging. Each agent l draws its desired step length r_l from another truncted normal distribution

$$r_l \sim \mathcal{N}_{\text{tr}}\left(\mu_l(v_l), \sigma_{r_l}, -\sigma_{r_l}\lambda_\sigma, +\sigma_{r_l}\lambda_\sigma\right). \tag{2.11}$$

Given its desired step length r_l and the current position $\mathbf{x}_{l,k}$ of an agent, the choice for the next agent's position follows the principle of the *homo economicus*: the next position $\mathbf{x}_{l,k+1}$ maximizes the agent's utility

$$\mathbf{x}_{l,k+1} = \underset{\mathbf{y} \in P_l + \mathbf{x}_{l,k}}{\arg\max}\, u_l(\mathbf{y}). \tag{2.12}$$

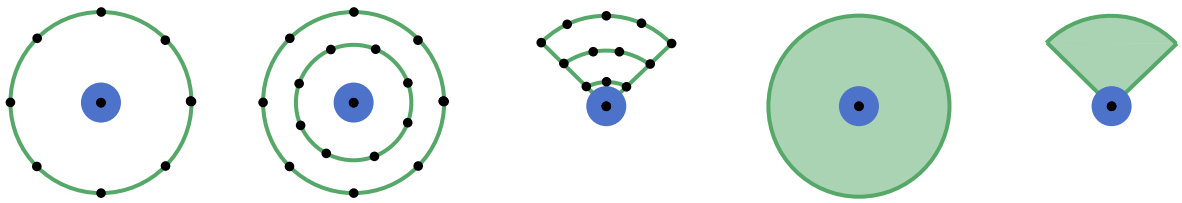
Restricted stepping

Different choices for the set of possible next positions P_l were introduced, each satisfying the following condition

$$\mathbf{y} \in P_l \Rightarrow \|\mathbf{y}\| \leq r_l. \tag{2.13}$$

One can differentiate between a connected infinite and discrete and finite set of positions. Strictly speaking, if the set is finite, the model is not only discrete in time but also in space, but in contrast to cellular automata, there is no global grid. Assuming P_l is finite and fixed for each individual for the whole simulation, each agent l moves on a specific grid defined by its initial position $\mathbf{x}_{l,0}$ and P_l. In [301] von Sivers and Köster proposed to use a connected infinite set

$$P_l = \{\mathbf{y} : \|\mathbf{y}\| \leq r_l\}, \tag{2.14}$$



(i) (ii) (iii) (iv) (v)

Figure 2.9: Different possible next positions P_l of an agent l at the center: while the step circle (i), multiple circles (ii) and the speed dependent restricted step circles (iii) are finite discrete sets, the step disc (iv) and the speed dependent step disc (a step circle segment) (v) are connected infinite sets.

such that agents adjust their step length accordingly. If P_l is infinite, the optimization problem is solved by the NelderAndMead algorithm [209], or evolutionary algorithms [297, 145]. Other gradient-free methods are applicable as well. I tested a particle swarm optimization (PSO) method, which achieved similar results compared to the NelderAndMead algorithm.

Instead of using a connected infinite set, modelers also considered a finite number of positions. In the introduction of the optimal steps models [257] the authors proposed a set of $m+1$ positions that lie on a circle of radius equal to the desired step length of the agent:

$$P_l = \left\{ (\cos(\theta), \sin(\theta)) \cdot r_l,\ \theta = \frac{2\pi}{m}(i + \epsilon_\theta) \,\middle|\, i = 0, \ldots, m \right\} \cup \{(0,0)\}. \tag{2.15}$$

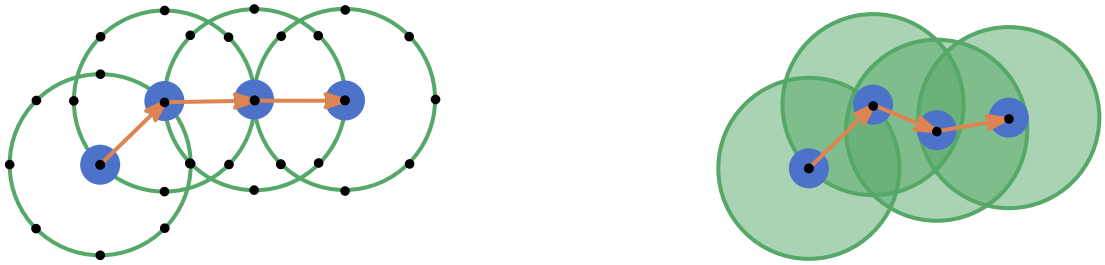
To eliminate unwanted discretization artifacts, they shift positions by $\epsilon_\theta \sim \mathcal{U}(0,1)$ each time an agent steps forward. The authors showed that by using a specific finite set P_l, the model could emulate a cellular automaton [257]. Seitz et al. [260] extended this concept to multiple circles to allow different step lengths. In [300], von Sivers and Köster used this construction to approximate the infinite set. I identified two arguments to use finite discrete over infinite connected sets:

(1) *computational perspective*: it reduces the computational burden while results are still accurate,

(2) *modelers perspective*: heuristics drives the cognitive process, therefore, finding "good-enough" values is realistic.

Since evaluating the utility function is one of the primary sources for the overall computational load, it is reasonable to use a small number of possible positions such that results are still accurate. Seitz [256] showed experimentally that, at some point, increasing the number of possible positions does not influence the measured (macroscopic) outcomes. From the modeler's perspective, researchers argue that people use heuristics to solve complicated problems, such as catching a ball [95, 101, 94]. Modelers such as Seitz [261, 256], Moussaïd [203, 200] and Xiao [320] argue that heuristics are also used to decide where to go next. Consequently, even though pedestrians may follow the principle of the *homo economicus*, it is reasonable to assume they find the best next position within a small margin.

Regardless of using a finite discrete or infinite connected set, P_l is also used to model physical restrictions, such as the desired step length r_l. Another restriction is the relation between *turning*



(i) (ii)

Figure 2.10: Different stepping behaviors: by using one step circle, agents move with a constant velocity or stop completely (i). By using a step disc, allows agents to use smaller steps and therefore smaller speeds than their free-flow speed (ii).

angle and *walking speed*, which was examined by Seitz et al. [260]. They suggest restricting the turning angle of the agent by its walking speed such that

$$\textit{turning angle} \leq \pi - \textit{waling speed}, \tag{2.16}$$

holds.

Slower stepping

If the agent's desired speed remains unchanged, the footstep duration is fixed at r_l/v_l. Since condition 2.13 ensures

$$\|\mathbf{x}_{l,k+1} - \mathbf{x}_{l,k}\| \leq r_l, \tag{2.17}$$

agents slow down by stepping on a position that is less desirable to reach their destination but is overall the best possible option. The free-flow speed v_l is the maximal achievable speed of agent l. Since the speed of an agent does not influence its desired step length r_l, there is no acceleration modeled. Instead, an agent l can adjust its speed to any value that is not greater than its free-flow speed v_l. In that sense, optimal steps models are velocity-based.

Instantaneous stepping

Since a footstep

$$((t_{l,k}, \mathbf{x}_{l,k}), (t_{l,k+1}, \mathbf{x}_{l,k+1})), \tag{2.18}$$

from $\mathbf{x}_{l,k}$ to $\mathbf{x}_{l,k+1}$ requires $t_{l,k+1} - t_{l,k}$ seconds, one has to decide at which point in time it is executed. During the development of optimal steps models, researchers made two suggestions. At first, Seitz et al. [257] proposed that agents perform their footsteps at the time they end ($t_{l,k+1}$). Later in [258], the authors reconsidered this questions more deeply and proposed the time they start ($t_{l,k}$). They justify their decision by arguing that pedestrians can anticipate the movement of others.

In the same contribution, the authors concluded that different update schemes influence the simulation outcomes. They compared a parallel and event-driven update scheme. The parallel update scheme relies on a global clock that increases the simulation time t iteratively by Δt while

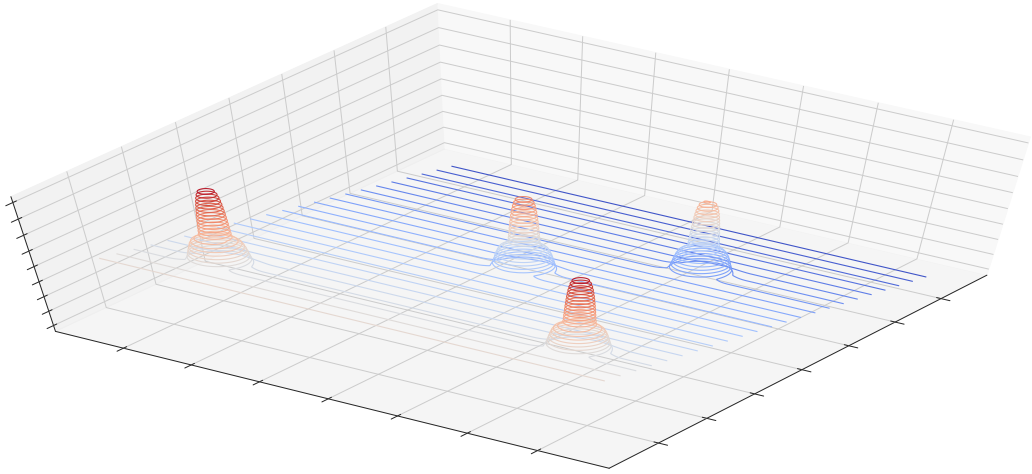


Figure 2.11: Contour plot of the overall utility field $-u_l$ of agent l: at the center of each of the four bells other agents are positioned. The destination utility $-u_\Gamma$ is indicated by the tilted plane and drives agent l towards the right.

the event-driven update scheme is clock-free. It executes footsteps when they start while the other update scheme might violate this condition by Δt seconds. I will explain these update schemes and their role in the acceleration of optimal steps models in Chapter 5.

Continuous utility fields

In all optimal steps models, the utility u_l of an agent is the superposition of three sub-utilities:

(1) The *global* destination utility u_Γ drives the agent towards its next intermediate destination.

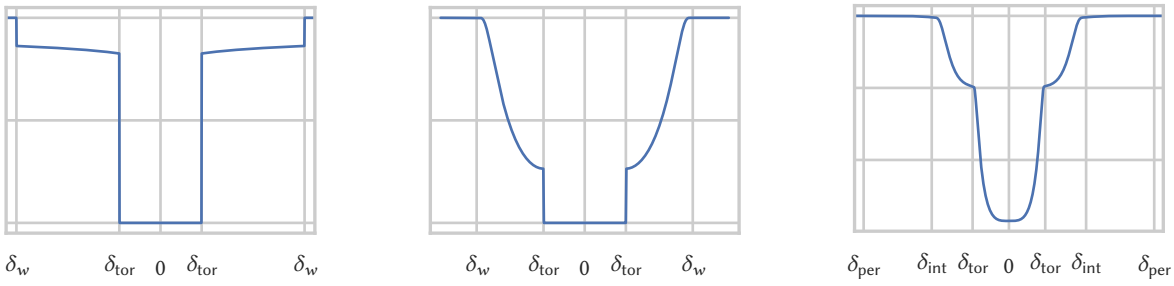
(2) The *local* obstacle utility $u_\mathcal{W}$ ensures that agents stay away from obstacles.

(3) The *local* pedestrian/agent utility $u_{\mathcal{A},j}$ ensures that agents stay away from agent j.

The net utility u_l for an agent l at $\mathbf{y}$ is defined by

$$u_l(\mathbf{y}) = u_\Gamma(\mathbf{y}) + u_\mathcal{W}(\mathbf{y}) + \sum_{\substack{j \in \mathcal{A} \\ j \neq l}} u_{\mathcal{A},j}(\mathbf{y}). \tag{2.19}$$

Pedestrian/agent and obstacle utilities are local, that is, agents far enough away from other agents and obstacles navigate solely based on the destination utility. Therefore, the destination utility u_Γ is responsible for the long- and medium-range navigation. It shares many similarities with *floor fields* of cellular automata. I refer to it as a *(continuous) navigation field* to distinguish it from *floor fields*. For example, to model evacuations, modelers use a navigation field that encodes the negated shortest travel time from an exit $\Gamma \subset \mathbb{R}^2$ to any position in the simulation domain Ω. Even though efficient algorithms for its calculation exist, its computation cost affects large-scale simulations. In Chapter 3, I discuss navigation fields as a modeling tool, and in Chapters 6, 8 and 9, I present my contribution to accelerate continuous navigation field computation for



(i) (ii) (iii)

Figure 2.12: The obstacle and agent utilities introduced by Seitz and Köster [257] (i), by Seitz et al. [260] (ii) and by von Sivers and Köster [300] (iii).

pedestrian dynamics. However, in the following, I continue with the presentation of different types of obstacle and pedestrian utilities introduced in [257, 300, 260].

To avoid collisions with other agents or (static) obstacles, repulsive functions based on the Euclidean distance are introduced. The utility of agent j affecting other agents at position $\mathbf{x}$ is given by $u_{\mathcal{A},j}(\mathbf{x})$. Seitz et al. [257, 260] propose the following form

$$u_{\mathcal{A},j}(\mathbf{x}) = \begin{cases} -\mu_{\mathcal{A}}, & \text{if } 0 \leq d_j(\mathbf{x}) \leq \delta_{\text{tor}} \\ -h_{\mathcal{A}} \cdot g_{\mathcal{A}}(\mathbf{x}), & \text{if } \delta_{\text{tor}} < d_j(\mathbf{x}) < \delta_w \\ 0 & \text{else,} \end{cases} \tag{2.20}$$

where $d_j(\mathbf{x})$ is the Euclidean distance from $\mathbf{x}$ to the center of agent j and δ_{tor} is the agents' torso diameter. The obstacle utility $u_{\mathcal{W}}$ is defined in a similar way with the difference that it depends on $d_\Omega(\mathbf{x})$, that is, the distance from $\mathbf{x}$ to the closest obstacle. Seitz and Köster [257] introduced the utility depicted in Fig. 2.12i, which uses $g_{\mathcal{A}} = g^1_{\mathcal{A}}$ with

$$g^1_{\mathcal{A}}(\mathbf{x}) = \exp(-a_{\mathcal{A}} \cdot d_l(\mathbf{x})^{b_{\mathcal{A}}}). \tag{2.21}$$

Later, in [260] the authors used compact support utility function defined by

$$g^2_{\mathcal{A}}(\mathbf{x}) = \exp\left(\frac{1}{\left(\left[d_j(\mathbf{x}) - \delta_{\text{tor}}\right]/\delta_w\right)^2 - 1}\right) \tag{2.22}$$

instead. Positions closer than the diameter of the agent's torso δ_{tor} are of low utility, preventing agents from overlapping. The utility depicted in Fig. 2.12ii increases more significantly for $\delta_{\text{tor}} < d_j(\mathbf{x})$. It also introduces infinite differentiability for all $\mathbf{x}$ with $\delta_{\text{tor}} < d_j(\mathbf{x})$. Furthermore, $g^2_{\mathcal{A}}$ vanishes outside of δ_w, such that numerical cut-off errors can be avoided. Another advantage of function $g^2_{\mathcal{A}}$ is that $u_{\mathcal{A},j}$ introduces only two new parameters $h_{\mathcal{A}}$ and δ_w, which simplifies calibration. While δ_w can be estimated by using experimental data, $h_{\mathcal{A}}$ is difficult to measure and difficult to interpret.

In [300], von Sivers et al. introduced more complex utilities that translate the theory of (inter-) personal space [112] into mathematical formulas. Each space is modeled as a disc around the

agent. The *contact space* ($d_j(\mathbf{x}) < \delta_{\text{tor}}$) defines the area for which pedestrian bodies physically touch. This space does not originate from Hall [112] but is a straightforward extension of the concept of personal spaces. Hall described the *intimate space* ($d_j(\mathbf{x}) < \delta_{\text{int}}$) to be the area where the sensory inputs (body head, smell, sound etc.) of another person become notable, and body contact is nearly unavoidable [300]. If the distance between agents increases further, that is if $d_j(\mathbf{x}) < \delta_{\text{per}}$, they reach the *personal space*. Pedestrians more than δ_{per} meters away from each other step into the *social space* of one another and if they are far apart they are in the others *public space*, respectively. Von Sivers et al. did not model Hall's social and public space because their influence on pedestrian dynamics is insignificant [300]. For each of the other spaces, the authors introduce a function of compact support, that is, $u_{j,\text{con}}$, $u_{j,\text{int}}$, and $u_{j,\text{per}}$, respectively. They are of similar form as the function of Eq. (2.22). The agent utility of an agent j affecting others at position $\mathbf{x}$ is defined by

$$u_{\mathcal{A},j}(\mathbf{x}) = \begin{cases} -u_{j,\text{con}}(\mathbf{x}) - u_{j,\text{int}}(\mathbf{x}) - u_{j,\text{per}}(\mathbf{x}) & \text{if } d_j(\mathbf{x}) < \delta_{\text{tor}} \\ -u_{j,\text{int}}(\mathbf{x}) - u_{j,\text{per}}(\mathbf{x}) & \text{if } \delta_{\text{tor}} \leq d_j(\mathbf{x}) < \delta_{\text{int}} \\ -u_{j,\text{per}}(\mathbf{x}) & \text{if } \delta_{\text{int}} \leq d_j(\mathbf{x}) < \delta_{\text{per}} \\ 0 & \text{else.} \end{cases} \tag{2.23}$$

Aside from a different resulting behavior, the advantage of these utilities is that the parameters δ_{tor}, δ_{int} and δ_{per} can be observed in the real world. They are difficult to measure and certainly different for each pedestrian, but they are based on a well-established social theory. All additionally introduced parameters of $u_{j,\text{con}}$, $u_{j,\text{int}}$, and $u_{j,\text{per}}$ (compare [300]), are as difficult to measure as the parameter $h_{\mathcal{A}}$ of the previously discussed utilities. I also want to stress that the effect of the pedestrian, obstacle, and destination utilities can only be analyzed in combination because of the superposition principle. For example, doubling the destination utility has the same effect as multiplying all other utilities by one-half. Increasing or decreasing the gradient of the destination utility ∇u_Γ strongly influences the outcome of the simulation as well. On the other side, adding or subtracting some constant from u_Γ has no effect at all. Therefore, especially the heights of utilities controlled by, for example, $h_{\mathcal{A}}$, can only be calibrated as a whole. If utilities are defined unreasonably, agents can get stuck. For example, we can imagine an agent attempting to walk through a corridor, but inside the corridor, the obstacle utility decreases faster than the destination utility increases. Consequently, the agent will stop moving. One interpretation is that the claustrophobic feeling imposed by the corridor outweighs the benefit of moving closer towards the destination.

Optimal steps models for large-scale simulations

Even though optimal steps models are continuous models, they share many similarities with cellular automata. They assume pedestrians optimize their positions in space according to a balance of goals: reaching a destination, avoiding obstacles, and other pedestrians. They reject force-based navigation and introduce an optimization method that operates on the superposition of multiple scalar utility fields instead. Moreover, they reject continuous movement and introduce discrete *footsteps*, similar to the cell hopping mechanism of cellular automata. However, in contrast to cellular automata, time is discretized not by a global ticking clock but by individual

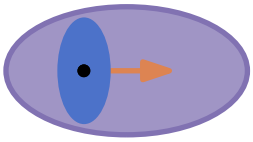


Figure 2.13: Elliptic pedestrian utility and body shape: the blue shape represents the pedestrian body, while the purple area indicates its utility function.

footstep events. Furthermore, optimal steps models use a continuous space, or the underlying grid is dynamically changing, according to the agent's position and speed. Consequently, optimal steps models do not suffer from the same unwanted artifacts and can reproduce high-density situations.

Despite the introduction of multiple utility functions, there is much room for improvement. For example, as suggested in [49] it is reasonable to assume that pedestrians are more affected by pedestrians in front than by others at their side. This asymmetric intensity can be modeled by using an elliptical-shaped pedestrian utility function. Another extension that could be introduced is to model the agent body by another elliptical utility. The idea is shown in Fig. 2.13. Finding suitable pedestrian and obstacle utilities is challenging, but many of their parameters are observable in the real world. Measuring the distance agents tend to keep to each other in certain situations is difficult, but measuring the social forces they impose on each other is impossible.

The lack of acceleration tends to lead to a too optimal stepping behavior, in the sense that agents competitively overtake others instead of embracing a follower behavior. As a consequence, agents in high-density situations walk faster than real-world pedestrians [339]. I suggest introducing *discrete acceleration* by imposing additional restrictions on the set of possible next positions P_l. Changing the shape of the step disc to a dynamic ring, ensures that the step length falls in some interval. Depending on the agent's velocity, the ring grows (acceleration) and shrinks (deceleration). Compare Fig. 2.14. Consequently, the velocity during one footstep can not change too much. This strategy could lead to a stepping behavior observed by [306].

Integrating dynamics into the *navigation field* $u_{l,\Gamma}$ is one way to model long- and medium-range interactions. A first *continuous dynamic navigation field* was proposed by [291, 174]. Köster and Zönnchen [165, 166] introduced a *continuous dynamic navigation field* to optimal steps models. Since the space of the spatial domain is continuous, dynamic navigation fields are computationally expensive.

Excluding the computation cost for the *navigation field*, the rest of the computational burden

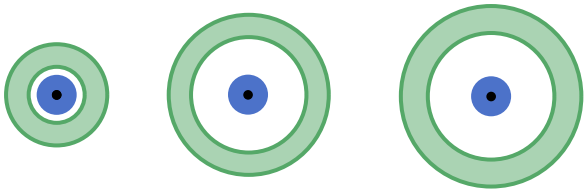


Figure 2.14: The concept of discrete acceleration: the stepping ring restricts the agent's step length to be in a specified interval. While an agent accelerates, this interval shifts to higher values. The ring can be even more restricted to allow only small turning angles for large velocities.

consists of evaluating local agent and obstacle utility functions. On the downside, multiple exponential functions have to be evaluated – dependent on the utility definition, the set of possible next positions P_l, and the used optimization method. Furthermore, the distance between some position $\mathbf{y}$ and nearby agents, as well as the closest obstacle, is needed. However, these computations are also required using a force-based or velocity-based model. Another algorithmic problem comes into play if the event-driven update scheme is used. Since it imposes a strict order on the performed footsteps, parallelization becomes difficult. On the upside, the *optimal step* principle allows large (individualized) time steps. In dense situations, the evaluation of the agent utilities becomes more expensive, because the number of agents influencing each other increases, but the time step length stays the same. Especially the execution time of force-based models suffers from the requirement of small time steps in dense situations. Therefore, optimal steps models are computationally more expensive than cellular automata but have the potential to outperform social force models and ODE-based velocity models such as the Gradient Navigation Model.

In conclusion, from my modeler's perspective, optimal steps models are attractive. Their decision-making process is not based on Newtonian mechanics, car traffic models, or velocity obstacles, but on social and psychological findings. Besides, the stepping behavior they model is a simplified version of the biomechanic stepping progress of real-world pedestrians. I think optimal steps models will further influence the discourse in the pedestrian dynamics community. Moreover, since there is still room for improvement, new optimal steps models will be introduced in the future. From my computer science perspective, they are attractive as well. Compared to ODE-based models, large (individualized) time steps lead to less computational work, and the locality of pedestrian and obstacle utilities are beneficial for parallelization. For all these reasons, I decided to focus on optimal steps models for large-scale simulations.

2.3 Summary

In this chapter, I gave an overview of the diverse landscape of microscopic pedestrian simulation models.

I discussed the introduction and refinement of the hierarchic model approach. It structures the pedestrian behavior into multiple levels and is the basis for most well-known models.

In Section 2.2 of this chapter, I reviewed different types of locomotion models. To this day, cognitive heuristics (Section 2.2.5) alone can not define a valid locomotion model since the *operational layer* is missing. However, they should be considered for the decision-making process. Cellular automata are computational efficient since they rely on highly regular data structures and often support a parallel update scheme (Section 2.2.1). However, their regularity imposes restrictions that lead to unwanted artifacts and a limited range of reproducible densities. Force-based models operate in continuous space and time (Section 2.2.2). They are motivated by Newtonian mechanics. Social forces accelerate agents towards their destination and away from other agents. They can model many situations but the lack of distinction between thinking and acting, that is, the confusion of the mental process with the physical motion of the pedestrian's body, leads to many problems. Many extensions mitigate those problems, but I think they can never be fully resolved if we do not change the model's underlying basis. Velocity-based models avoid acceleration by forces (Section 2.2.3). Combined with a decision-making strategy, they are a promising

alternative. In my opinion, optimal steps models fall into this category (Section 2.2.6). They model the decision-making process of the homo economicus. Furthermore, optimal steps models are motivated by the biomechanical process and findings from psychology rather than physical laws of moving particles. I also argued that from an efficiency perspective, optimal steps models are attractive. They are more computationally expensive than cellular automata but impose less computational costs than force-based models. In Chapter 5, I develop parallel algorithms for optimal steps models. Therefore, I gave a detailed and preparing description. In Section 2.2.4, I shortly discussed another path to build a model: data-driven modeling. Data-driven models learn the pedestrians' behavior by data, mainly video footage. Since there is a lot of process in the development of learning algorithms, they will influence the pedestrian dynamics community in the future. However, to this day, they are not yet competitive compared to classical model approaches since they cannot deal with the variety of scenarios that classic models successfully capture.

CHAPTER 3

Navigation fields

"The world is not comprehensible, but it is embraceable: through the embracing of one of its beings."

– Martin Buber

Researchers often use the terms *navigation field*, *floor field*, *scalar field*, and *potential field* synonymously. In the discrete ($\Omega \subset \mathbb{Z}^2$, $\Gamma \subset \Omega$) as well as in the continuous (non-discrete connected) space ($\Omega \subset \mathbb{R}^2$, $\Gamma \subset \Omega$) these fields are scalar fields

$$u_\Gamma : \Omega \to \mathbb{R} \tag{3.1}$$

that assign to each position $\mathbf{x} \in \Omega$ a real value $u_\Gamma(\mathbf{x})$, which is sometimes interpreted as utility, potential, or probability. Ω is the spatial simulation domain and Γ the spatial destination of interest. I distinguish between *floor fields* of cellular automata and continuous *navigation fields*. Although there are other methods to generate a navigation field, such as flood fill methods [174], I focus on solving the *eikonal equation* because it is widely and successfully employed. Therefore, if I refer to *navigation fields*, I mean solutions of this equation, which I introduce in the following section.

3.1 Large-scale human navigation

Regardless of their name, these fields "guide" agents from any position to a spatial destination Γ. They combine attraction and repulsion. The idea of navigation guided by floor fields was first introduced in the field of robotics [146, 147, 180]. However, in robotics, the environment is usually unknown to the robot, while in pedestrian dynamics, modelers often assume that pedestrians have comprehensive knowledge about the spatial structure of their environment [173]. Navigation fields model multiple aspects of the wayfinding process and are based on the psychological concept of the so-called *cognitive map*, firstly mentioned by [287], and the visual perception of pedestrians.

"Cognitive maps are the internal representation of experienced external environments, including spatial relations among features and objects." – Golledgel et al. [102]

Neuroscientists know that the hippocampal formation, which is part of the limbic system of the human brain, is mainly responsible for storing and retrieving spatial memories [213]. In their book *The hippocampus as a cognitive map*, O'Keefe and Nadel connect the hippocampus with the mental representation of space and context-dependent memory. They argue that

> *"the hippocampus is the core of a neural memory system providing an objective spatial framework within which the items and events of an organism's experience are located and interrelated."* – O'Keefe and Nadel et al. [213]

In [199], Moser et al. discovered *place cells* and *grid cells* in rats' brains involved in the formation of the *cognitive map*. Ekstorm et al. [78] made similar discoveries. They observed *place cells* that increase their firing rates when the animal traverses specific regions of its surroundings, providing a context-dependent map of the environment. The ongoing research indicates that similar cells contribute to the wayfinding decision-making process of humans. It is unclear if the cognitive map takes a cartographic form. There is more evidence for randomly distributed place cells [102]. Golledge et al. [102] explored the connection between environmental learning and cognitive maps in the context of learning a route. They compared blind, vision-impaired, and sighted volunteers and concluded that blind and vision-impaired participants require additional trials. However, after enough learning, the wayfinding abilities of the three groups were equivalent. The authors suggest that the lack of sight interfered with putting knowledge into action [102]. Apart from learning the environments, humans classify their environment and assume particular properties of it. For example, after visiting multiple underground stations of a city, humans can infer how an unknown station will be structured. Therefore, pedestrians might use the cognitive map of some environments to navigate through another unknown environment. There is still much research required to understand the structure and impact of the cognitive map. However, there is strong evidence that it helps us to find our way in many situations, especially in environments visited multiple times before. Nonetheless, it is also known that people get lost in several situations because of human errors, that is, inaccurate, incomplete, and wrong cognitive maps [79].

3.2 Optimal path navigation

Following the notion of the *homo economicus*, we can argue that pedestrians are motivated to choose the optimal path based on some metric. One of the most suggested and applied metrics is the shortest and quickest path. In their contribution from 2011, Kretz et al. [174] summarized:

> *"Usually, models of pedestrian dynamics are (implicitly) built on the assumption that pedestrians walk along the shortest path. [...] There are, however, situations in which travel time matters a lot for pedestrians, which is why they must base their movement decisions on the criterion which direction at some given point in time appears to promise the smallest remaining travel time."* – Kretz et al. [174]

However, there are more criteria a pedestrian might consider, and further research is required. For example, pedestrians might follow the path of fewest turns, minimal effort, or the most familiar,

most secure, and most aesthetic path. Despite this research gap, we could model other criteria by using the notion of the quickest path. For example, we can artificially reduce the travel speed for insecure streets. Nonetheless, there are criteria, such as the path of fewest turn, that are difficult to model by navigation fields – a topic of another thesis.

The simplest ansatz to encode proximity is to calculate the Euclidean distance from a destination to each point in the domain. The closer an agent is to the destination, the better its position. However, this approach is not suitable for large-scale simulations since obstacles such as walls or buildings might be in the pedestrian's line of sight. Therefore, modelers suggest using an approximation of the geodesic distance.

Aside from the walking distance, the time required to reach a destination is another important metric for the *homo economicus*. Therefore, many modelers such as [291, 156, 115, 174, 165] suggest that navigation fields and floor fields should encode the time instead of the distance required to reach a destination.

The field is constructed for a specific destination Γ. For multiple destinations, multiple fields have to be computed. Note that Γ can be the set of multiple disconnected sub-regions. Modelers introduced different approaches in order to construct a *floor field*. They use Dijkstra's algorithm [71] on a visibility graph [211, 169], the FloodFillMethod [173], RayCasting [173] or the FastMarchingMethod [130, 291, 115] introduced by Tsitsiklis [292] and later by Kimmel and Sethian [151]. Even though *navigation fields* are continuous, they are usually constructed by solving a discrete problem on a discretization of the domain Ω. Therefore, partly the same algorithms are used to compute *floor fields* and *navigation fields*.

3.3 The eikonal equation

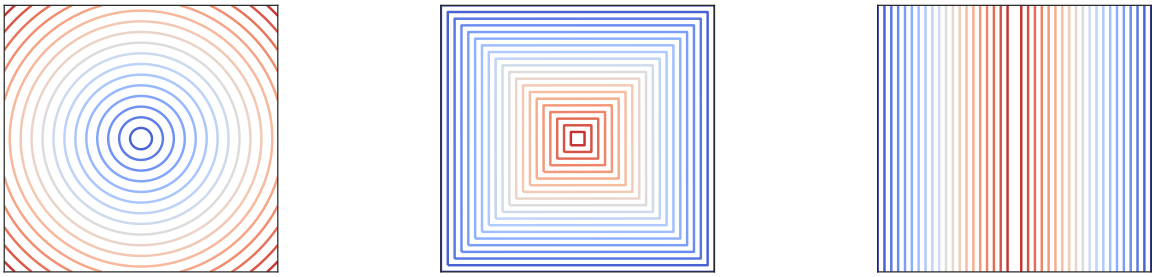
Most navigation field based models rely on the solution of the eikonal equation

$$\begin{aligned} \|\nabla \Phi_\Gamma(\mathbf{x})\| &= f(\mathbf{x})^{-1}, \quad \mathbf{x} \in \Omega \\ \Phi_\Gamma(\mathbf{x}) &= 0, \qquad \mathbf{x} \in \Gamma \\ f(\mathbf{x}) &\geq 0, \qquad \mathbf{x} \in \Omega, \end{aligned} \tag{3.2}$$

It is a non-linear boundary value problem for a partial differential equation. In my interpretation, $f : \Omega \to \mathbb{R}^+$ is a scalar field of the traveling speed for the propagating wave. The wavefront propagation starts at $\mathbf{x} \in \Gamma$ at $\Phi_0(\mathbf{x})$ seconds. Equation (3.2) states that the rate of change of the travel time (in space) is equal to the inverse travel speed. The eikonal equation's solution gives us the minimum time-of-arrival $\Phi_\Gamma(\mathbf{x})$ from $\mathbf{x}$ to Γ. In one dimension, Eq. (3.2) translates to

$$\frac{dt}{dx} = f(x)^{-1} \Rightarrow \frac{dx}{dt} = f(x). \tag{3.3}$$

The wavefront propagates information from the boundary Γ along the characteristics. Because of non-linearity, characteristics may intersect which results in the formation of shocks. The viscosity solution is continuous but may not be differentiable everywhere, compare Fig. 3.1. The existence and uniqueness of the viscosity solution are shown in [54]. In Chapter 9, I discuss the equation in more detail and show how it is solved.



(i) (ii) (iii)

Figure 3.1: Solutions of the eikonal equation for $f = 1, \Phi_0 = 0$: Ω is defined by a 2×2 square centered at $(0, 0)$: there are no shocks present for $\Gamma = \{(0, 0)\}$ and the solution is equal to the Euclidean distance (i). However, even for a simple example $\Gamma = \partial\Omega$ (ii) or $\Gamma = \{(x, y) \mid x = 1 \vee x = -1\}$ (iii), Φ is not differentiable at positions $\mathbf{x} = (x, y)$, where $\mathbf{x} = \mathbf{0}$ (ii) and $x = y$ (iii), respectively.

The equation has many applications in optimal control, computer vision, medicine, geometric optics, and path planning. Especially the special case of a constant travel speed $f = 1$, that gives us the geodesic distance $\Phi_\Gamma(\mathbf{x})$ from $\mathbf{x}$ to Γ, is solved to compute distance fields and curvatures of three-dimensional surfaces in the field of computer vision. For optimal steps models, the solution of the eikonal equation is equal to the negated destination utility, that is,

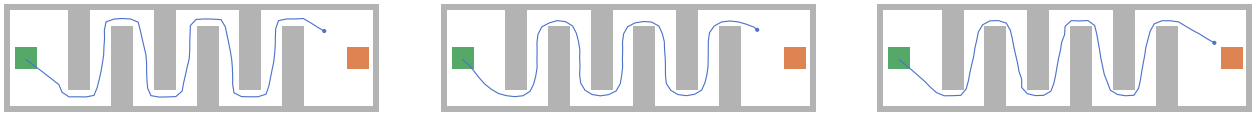
$$u_\Gamma(\mathbf{x}) = -\Phi_\Gamma(\mathbf{x}). \tag{3.4}$$

Each microscopic continuous space model I discussed so far, assumes that the direction towards the destination is known. Especially in early publications, modelers simplify the problem to the gradient of the Euclidean distance. Compare, for example, the first publication of the Social Force Model [118]. For large-scale simulations, the geometry is complex and optimal paths are no longer straight. Using the negated and normalized gradient of the travel time to be the destination direction $\mathbf{n}_\Gamma$ of an agent, that is,

$$\mathbf{n}_\Gamma(\mathbf{x}) = -\frac{\nabla\Phi_\Gamma(\mathbf{x})}{\|\nabla\Phi_\Gamma(\mathbf{x})\|} \tag{3.5}$$

is a robust robust technique. Note that in Eq. (3.5) we assume the agent wants to move towards Γ and is positioned at $\mathbf{x}$. This definition is used by, for example, [291, 174, 257, 68, 165, 293].

All continuous space models discussed in Section 2.2 rely on either Φ_Γ or some specific destination direction vector. Looking at the introduction of new models, for example, [118, 326, 222, 190, 261, 320], there is a lack of discussion on how to define and compute the destination direction. In the end it fundamentally influences the motion of all agents and is therefore of great importance. Models are calibrated and validated for a small-scale setting such that $\mathbf{n}_\Gamma$ can be computed by using the Euclidean metric. If obstacles disrupt the line of sight, modelers tend to introduce artificial intermediate destinations [49] which leads us to graph-based models on the tactical level. However, the location and shape of intermediate destinations is a new non-trivial problem one has to solve. Note that even graph-based models can use $\mathbf{n}_\Gamma(\mathbf{x})$ as the negated gradient of the traveling time [160]. If we instead use navigation fields by solving the eikonal equation, we follow the logic of the *homo economicus* for all sorts of geometries because optimal paths follow



(i) (ii) (iii)

Figure 3.2: The trajectory of a single agent walking from a source (green) to a destination (orange) using an optimal steps model with the default obstacle utility and f_{unit} (i) and without an obstacle utility but the travel speed function f_ρ (ii) and f_{lin} (iii), respectively.

the gradient of Φ_Γ [150]. Therefore, navigation fields are a modeling technique integrated into microscopic continuous space models and a robust method to compute $\mathbf{n}_\Gamma$ in general.

3.4 Static navigation fields

Static navigation fields result from the eikonal equation for which the traveling speed function f does not change during the simulation. Therefore, the traveling speed function does not depend on any dynamics. Consequently, we have to solve Φ_Γ only once. The equation is commonly used to compute the distance in the Euclidean metric under consideration of obstacles, that is, the geodesic distance. Let us assume $\Omega \subset \mathbb{R}^2$ is the simulation domain where we exclude the space covered by obstacles. To compute the geodesic distance, we choose $\Phi_0 = 0$ and a traveling speed that is equal to 1 everywhere except at the domain boundary, that is,

$$f_{\text{unit}}(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \partial\Omega, \\ 1 & \text{else.} \end{cases} \tag{3.6}$$

Agents following the gradient of $u_\Gamma = -\Phi_\Gamma$ use the shortest path towards their destination Γ. If there are certain areas where pedestrians slow down or speed up, for example, if they move uphill, on stairs, or if they tend to avoid regions such as positions very close to walls, this can and is modeled by choosing a different traveling speed function f. For example, in [165] I propose to use

$$f_\rho(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \partial\Omega, \\ 1/(1 + h_W \cdot \rho_W(\mathbf{x})) & \text{else,} \end{cases} \tag{3.7}$$

where $\rho_W(\mathbf{x})$ is the obstacle density defined in [257] and $h_W > 0$ is some constant that we have to calibrate. By using Eq. (3.7) or Eq. (3.8) we lift the obstacle avoidance from a local mechanism to a global one. If obstacle avoidance is a local mechanism, agents adjust their path locally which leads to a sub-optimal path. If avoidance is a global mechanism, the path is shorter. Figure 3.2 illustrates this effect.

To compare the Behavioral Heuristics Model (BHM) presented by Seitz et al. [261] and one of the optimal steps models (see [339]), I chose the vector of Eq. (3.5) to be the destination direction of the BHM. Furthermore, I integrated obstacle avoidance into the Behavioral Heuristics Model

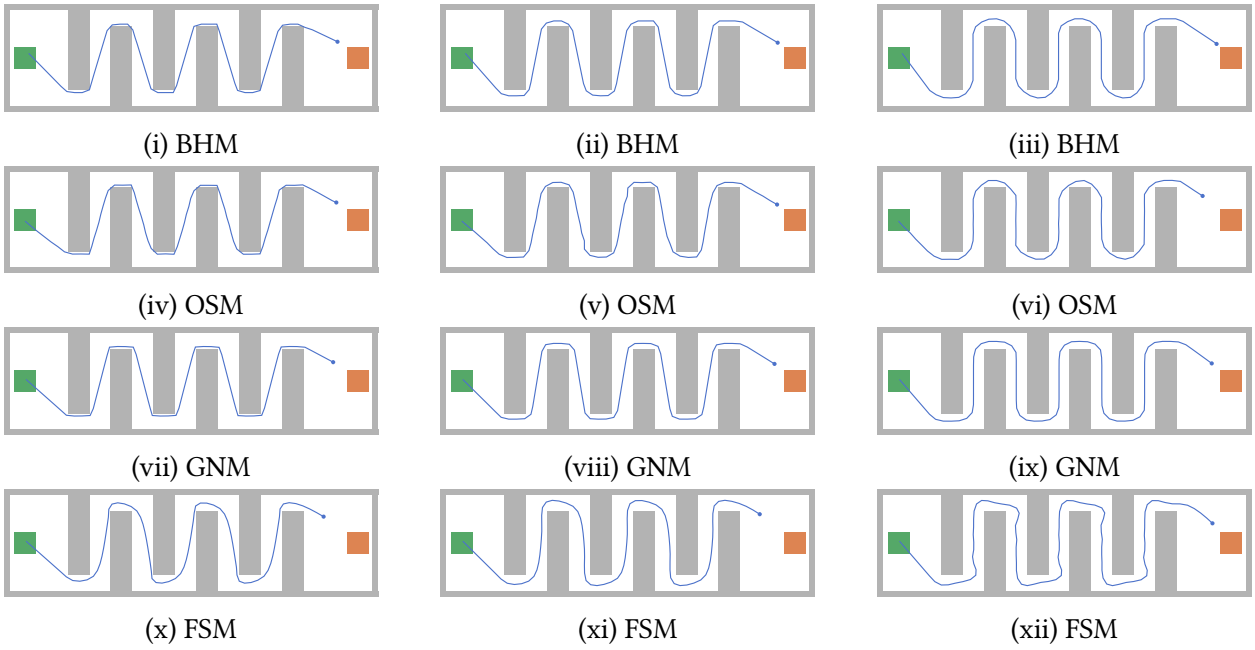


Figure 3.3: The trajectory of a single agent walking from a source (green) to a destination (orange) using the Behavioral Heuristics Model with $\mathbf{n}_\Gamma = \nabla\Phi_\Gamma(\mathbf{x})/\|\nabla\Phi_\Gamma(\mathbf{x})\|$ (i to iii), a optimal steps model with $u_\mathcal{W} = 0$ (iv to vi), the Gradient Navigation Model (vii to ix) and the Social Force Model without obstacle forces (x to xii). I chose f_{lin} (see Eq. (3.8)) with $h_\mathcal{W} = 1.0$. For the first column $\delta_\mathcal{W} = 0.2$, for the second $\delta_\mathcal{W} = 0.5$, and for the third $\delta_\mathcal{W} = 1.0$. Regardless of using the BHM, OSM or GNM, the distance the agent keeps from obstacles is approximately the same. In the case of the SFM, we can observe the effect of acceleration. For $\delta_\mathcal{W} = 1.0$ the agent oscillates.

by using

$$f_{\text{lin}}(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \partial\Omega, \\ 1/(1 + h_\mathcal{W} \cdot [1 - (d(\mathbf{x})/\delta_\mathcal{W})]) & \text{if } d(\mathbf{x}) < \delta_\mathcal{W}, \mathbf{x} \in \Omega \setminus \partial\Omega \\ 1 & \text{else.} \end{cases} \tag{3.8}$$

A similar speed function was independently suggested by [104] to add obstacle avoidance into social force models. Figure 3.3 shows different trajectories for different travel speed functions and different microscopic pedestrian models.

In summary, even though the model definitions do not include $\mathbf{n}_\Gamma$ defined as the gradient of the travel time Φ_Γ, I think many existing and future models can benefit from it.

3.5 Dynamic navigation fields

In [116], Hartmann et al. differentiate between *short-*, *medium-*, and *large-scale* interactions. The authors did not define these terms formally but connected them to the hierarchical structure of the modeled decision making process of Section 2.1. Following this notion, short- and medium-range interactions belong to the operational level while some tactical model realizes long-range

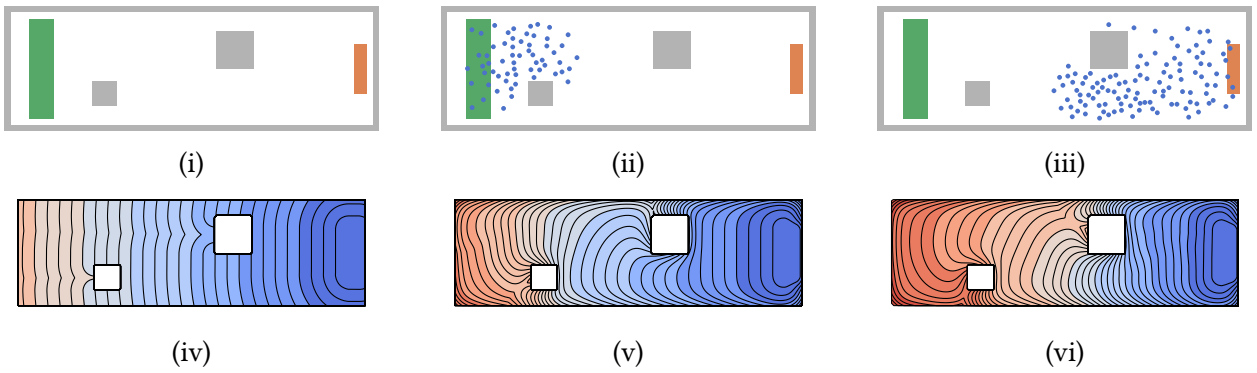


Figure 3.4: Simulation using the dynamic navigation field of [165]: the simulation at 0 (i), 6 (ii), and 32 (iii) seconds and the respective dynamic navigation field (iv, v, vi) of the orange destination.

interactions. I define the *interaction range* to be the geodesic distance within which the agent is influenced by other agents. For example, short-range interactions are modeled by repulsion and attraction forces [118], utility functions [257] or simple heuristics [261]. Short ranges are usually limited to a couple of meters. I think of the medium-range as the range within pedestrians can perceive others. Therefore, the term *perception-range* might be more appropriate. Since humans can anticipate, as well as inter- and extrapolate, the *perception-range* goes beyond what pedestrians can see.

Modelers use dynamic navigation fields to model interaction within the perception-range of agents. They rely on a dynamic recalculation of the navigation field that takes other pedestrians into account and yields significantly more realistic simulation results [116]. The idea, first introduced by Treuille et al. [291], is to define a travel speed function f that depends on local properties such as the pedestrian density, flow, or speed. For example,

$$f(\mathbf{x}) = \frac{1}{1 + h_{\mathcal{A}} \cdot \rho_{\mathcal{A}}(\mathbf{x})}, \tag{3.9}$$

where $\rho_{\mathcal{A}}$ is a measure of the local density of pedestrians, and $h_{\mathcal{A}} > 0$ is a constant that has to be calibrated, generates significant more realistic behavior [116].

Hartmann et al. extended this to

$$f_\Gamma(\mathbf{x}) = \frac{1}{1 + h_{\mathcal{A},\Gamma} \cdot \rho_{\mathcal{A},\Gamma}(\mathbf{x}) + h_{\mathcal{A},\neq\Gamma} \cdot \rho_{\mathcal{A},\neq\Gamma}(\mathbf{x})}, \tag{3.10}$$

where $\rho_{\mathcal{A},\Gamma}$ is the density with respect to all agents of the current destination equal to Γ and $\rho_{\mathcal{A},\neq\Gamma}$ is the density with respect to all other agents. Using $h_{\mathcal{A},\neq\Gamma} > h_{\mathcal{A},\Gamma}$ they assume that following pedestrians with a different destination is significantly less attractive than following pedestrians with the same destination.

In [165], Köster and I extended this binary categorization to a fluent one by introducing a weight for each agent l. The weight depends on the speed $v_{l,\Gamma}$ towards the destination of the dynamic navigation field. To compute $v_{l,\Gamma}$ we use a static navigation field, that is,

$$v_{l,\Gamma} = \frac{\Phi_\Gamma(\mathbf{x}_{l,k-1}) - \Phi_\Gamma(\mathbf{x}_{l,k})}{t_{k-1} - t_k}, \tag{3.11}$$

where $\mathbf{x}_{l,k}$ is the position of agent l at t_k. If the agent moves away from Γ, $v_{l,\Gamma}$ is negative and it is regarded as influential. If it is positive, the agent moves towards Γ and its influence is negligible.

Kretz et al. [174] also used the eikonal equation as a mathematical framework to extend social force models. The authors observed that the travel time is heavily influenced by the distribution of all the other participants of traffic, be it vehicular or pedestrian traffic. They argue that, in contrast to traffic simulation, in pedestrian dynamics, researchers are much less aware of this issue, because pedestrians do not move on a network as vehicles do, but freely in two spatial dimensions [174]. They solve the eikonal equation on a regular grid of side length between 15 cm and 20 cm. In order to integrate dynamics they manipulate f by the agent velocity $\mathbf{v}$:

$$f(\mathbf{x}) = \frac{1}{1 + \max\left\{0, h_{\mathcal{A}} \cdot \left(1 + h_\theta \cdot \frac{\mathbf{v}}{v_0} \cdot \frac{\nabla\Phi_\Gamma(\mathbf{x})}{\|\nabla\Phi_\Gamma(\mathbf{x})\|}\right)\right\}}, \tag{3.12}$$

where v_0 is the free-flow velocity (of all agents), $\mathbf{v}$ is the current velocity of the agent that occupies the grid point and $h_{\mathcal{A}}$, h_θ are free parameters of the method. $h_{\mathcal{A}}$ controls the impact of an agent in general while h_θ controls the impact of its moving direction. If the grid point is not occupied, the travel speed stays unchanged, that is, $f = 1$. The idea is similar to the previous approach: agents that move towards Γ have a smaller impact than agents that walk in the opposite direction. In the former case

$$\frac{\mathbf{v}}{v_0} \cdot \frac{\nabla\Phi_\Gamma(\mathbf{x})}{\|\nabla\Phi_\Gamma(\mathbf{x})\|} = -\frac{\mathbf{v}}{v_0} \tag{3.13}$$

holds and in the latter

$$\frac{\mathbf{v}}{v_0} \cdot \frac{\nabla\Phi_\Gamma(\mathbf{x})}{\|\nabla\Phi_\Gamma(\mathbf{x})\|} = +\frac{\mathbf{v}}{v_0}. \tag{3.14}$$

At this point, the reader might think that connecting the agent speed directly with the travel speed (defined on the grid), for example, by using

$$f(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \partial\Omega \\ \|\mathbf{v}_l\| & \text{if there is an agent } l \text{ present at the grid point} \\ v_0 & \text{else,} \end{cases} \tag{3.15}$$

might be the most straightforward approach. However, as Kretz et al. [174] point out, if $\|\mathbf{v}_l\|$ is 0, the wavefront stops at the respective grid point. In front of bottlenecks, waiting agents would completely stop the wavefront from moving through it and motion would come to an end. Therefore, $f(\mathbf{x})$ cannot be equal to the actual agent speed at $\mathbf{x}$.

Aside from reducing the travel speed at occupied areas, Köster and I used the opposite effect to model queuing in front of bottlenecks and a follower behavior [165]. To achieve a global attraction towards other agents and repulsion of areas close to obstacles we suggested

$$f_{\text{queue}}(\mathbf{x}) = \frac{1}{1 - \min\{h_{\mathcal{A}} \cdot \rho_{\mathcal{A}}(\mathbf{x}), 1 - \epsilon\} + h_{\mathcal{W}} \cdot \rho_{\mathcal{W}}(\mathbf{x})}, \tag{3.16}$$

where ρ_{ob} is the obstacle density defined in [257], $\epsilon > 0$ is a small threshold and $h_{\mathcal{A}}$, $h_{\mathcal{W}}$ are free parameters of the technique. Ignoring the obstacle density we get

$$f \to \frac{1}{\epsilon} \text{ for } (h_{\mathcal{A}} \cdot \rho_{\mathcal{A}}) \to 1. \tag{3.17}$$

The wavefront moves with rapid speed over occupied areas. Therefore, optimal paths lead through these areas, and agents follow other agents. If $h_{\mathcal{A}}$ is large enough, they even stop overtaking others [165].

3.6 Summary

In this chapter, I described the origin and application of navigation fields in microscopic pedestrian simulation. I started by presenting the neuroscientific background of large-scale human navigation called *cognitive maps*. Furthermore, I argued that *navigation fields* model multiple aspects of the wayfinding process, including *cognitive maps*.

In Section 3.2, I discussed the term "optimal path" under the assumption of the *homo economicus*. What is optimal is debatable and depends on the individual, its needs and the situation it is exposed to. First, modelers considered the length of the path. But soon after, the travel time was introduced as another essential criterion for the *homo economicus*.

In Section 3.3, I introduced the eikonal equation – the mathematical framework of *navigation fields*. I stressed the often forgotten importance of the destination direction $\mathbf{n}_\Gamma$. It can, and often is derived from the eikonal equation's solution and fundamentally influences simulation results for all discussed models.

In the following two sections (Sections 3.4 and 3.5), I discussed different definitions for $\mathbf{n}_\Gamma$ by presenting different travel speed functions f. First, I presented static travel speed functions. They do not change over time. Consequently, static navigation fields do not change during the whole simulation run. Researchers of the listed publications argue that if medium-scale navigation is required, dynamic navigation fields achieve realistic results. In this case, f depends on the dynamics of the simulation and changes over time. Therefore, navigation fields must be recomputed, usually, after some time step Δt. I presented different dynamic travel speed functions from the literature.

PART II

Large-scale microscopic locomotion

CHAPTER 4

Large-scale simulations

> *"Our life always expresses the result of our dominant thoughts."*
>
> – Søren Kierkegaard

The term *large-scale*, in large-scale microscopic simulations, refers to two properties of a simulation: (1) the size and complexity of the simulation domain Ω, and (2) the number of simulated agents $n_{\mathcal{A}} = |\mathcal{A}|$. To enable real-time large-scale simulations one has to deal with both. The same is true for many other problems from other areas, such as the simulation of car traffic, molecules, and galaxies. However, if one asks me what the difference between simulating a large number of pedestrians and a galaxy system is, I always answer:

> *"In pedestrian dynamics we deal with complex systems consisting of intelligent inhomogeneous subjects, while in physics, chemistry, and other areas, the entities we are looking at are lifeless objects."*

Like the n-body problem [226], we are dealing with many components that behave similarly on the operational level. But in addition, they behave differently and make complicated decisions on higher levels. Therefore, it is no surprise that physical analogies inspire pedestrian models on the operational level, but psychology and sociology play an important role as well, especially at higher levels. It is also no coincidence that one of the most important conferences for pedestrian dynamics, the Conference on Traffic and Granular Flow, covers topics from granular flow.

Pedestrian dynamics is a complicated problem: by the term complicated, I refer to the level of difficulty, the degree of difference in decision-making, and the inhomogeneity of pedestrians. Models or model parts on the strategic and tactical level describe a complicated process. Finding the way to a particular location, deciding in which order one wants to visit specific intermediate destinations, or when to change one's behavior from competitive to cooperative is complicated. The different decisions an agent can make is revealed if we analyze strategic or tactical models. One can argue that the purpose of the strategic and tactical level is to increase the set of possible behaviors of agents. The decision-making process on that level is some artificial intelligence. Most models use a reasoning system to generate conclusions from available knowledge utilizing

logic or heuristics. For example, Seitz [256] describes heuristics to decide if an agent should wait, step forward, or evade sideways. Kleinmeier et al. [158] introduce a simple reasoning system to determine if agents should start cooperating to resolve a situation where they got stuck. In [86], fuzzy inference systems resemble human reasoning and determine the agent's movement direction. Kielar et al. [149] introduce a state machine to model the strategic level. Later, Kielar et al. [148] presented a pedestrian destination choice model that implements the decision making for picking the next destination of an agent.

Thinking and decision making in itself is a stepwise sequential process which is reflected by artificial intelligence code – it is often branch-heavy and, therefore, unsuitable for parallelization. Consequently, for a set of agents, many different execution paths are traversed. Exploiting vectorization or single instruction multiple data (SIMD) architectures, such as GPUs, require a low execution path divergence. Therefore, branch-heavy code should be avoided. Besides, the evaluation of each condition is, in many cases, computationally inexpensive. Often it consists of picking a specific next target from a list of targets by some condition. For example, in [304] von Sivers et al. studied different search strategies which they implemented on the tactical level of an optimal steps model. The strategies require $O(m \cdot n_{\mathcal{A}})$ time for the whole simulation, where m is the number of doors inside a building and $n_{\mathcal{A}}$ the number of agents. If we compare this to the optimization which requires $O(|P_l|)$ time for each footstep of a single agent l, the computational cost is small. Most of the computational burden is part of the operational level. Additionally, the execution path divergence weakens the benefit of parallel execution.

In conclusion, if we want to accelerate simulations to enable real-time large-scale pedestrian simulations, I suggest to introduce parallelism first and foremost to the operational level. Additionally, code optimization should start at the operational level as well.

Pedestrian dynamics is a complex problem: by complex, I refer to the number of components (agents) in the simulation and their homogeneity with respect to the operational level. At the operational level, the next intermediate spatial destination of all agents is known, and computing the following agents' position is similar for every agent. In some sense, agents behave more like lifeless particles after making their decisions on the higher levels. Some authors even refer to them as "particles" [245]. Therefore, models on the operational level are more closely related to models investigating physical phenomena such as fluid and molecular dynamics.

Concrete implementations reveal that the processing units execute the same code for each agent. In a relevant real-world scenario, pedestrians spend most of their time with walking. Coincidentally, most of the simulation run time is spent on the operational level, and its high computational costs are critical. For example, for the Social Forces Model [118] (see Section 2.2.2), we compute the next position by applying a force F_l to each agent l. In the case of the Optimal Steps Model [257, 298] (see Section 2.2.6) we compute the next position by solving an optimization problem for each agent. Cellular automata are inherently regular and adapt the state of each cell for each update phase. In each case, some operation such as integration, optimization or evaluation of a state transition function is executed for all agents for many points in time resulting in an overall heavy workload.

4.1 The parallel nature of pedestrian dynamics

If we assume the model implementation is highly optimized, the only way to reduce computation times is to add more and more computation power to the hardware system in use. However, with the breakdown of Dennard scaling, clock frequencies of single central processing units (CPUs) no longer increase significantly. As a consequence, manufacturers turned their attention towards multi-core processors. Consequently, parallel implementations of microscopic simulation models are required to enable large-scale simulations. I argue that parallelism is a natural property of pedestrian dynamics that should be exploited by developers of model implementations.

Watching people walking through streets, an airport, or other facilities reveals a simple truth: pedestrians think, make decisions, and move in parallel. We can think of each person as a physically movable processing unit, which computes at every point in time its next position – consciously or unconsciously. Another observation is that pedestrians act independently from one another if (1) they are unaware of each other or (2) their interest do not conflict. In my eyes, the key to successfully simulating large-scale scenarios using a microscopic model is to exploit these two observations.

To simulate thousands of individuals, the model should be computationally inexpensive, but more importantly, it should scale well. Two factors are essential:

(1) How well does it scale with an increasing spatial domain / simulation area?

(2) How well does it scale with an increasing number of agents?

Even for computationally expensive models, scalability makes large-scale simulations possible by adding more hardware, especially processing units. Assuming p is the portion of the simulation that can be parallelized, and $1 - p$ is the fraction that can not, then Amdahl's Law [237]

$$\frac{1}{(p-1) + \frac{p}{n_P}} \tag{4.1}$$

gives us the maximal possible speedup using n_P processors. If $n_P \to \infty$, Amdahl's Law reveals an upper bound of the speedup equal to $1/(1-p)$. Amdahl's Law makes some strong assumptions, such that either one or n_P processors execute the code or that the overhead of thread creation is negligible. In some high-performance computation settings, these assumptions are valid but not in general. Therefore, Amdahl's Law can only serve as a benchmark.

4.2 Algorithmic structure

Operational models

Looking at many operational models, one can break down their computation into several parts. In the initialization phase, domain-specific data structures are constructed, and agents are initialized. For example, the static floor field of cellular automata, the navigation field of optimal steps models, and navigation graphs are initialized. Then the simulation loop, which basically consists of two sub-routines, begins. The fist method StrategyAndTactic realizes the strategic

Algorithm 1: SimulationRun

```
1 Initialization();
2 while simulation is running do
3     StrategyAndTactic(Δt);
4     Locomotion(Δt);
```

and tactical decision-making of the agent. It includes the update of dynamic navigation or floor fields. Then Locomotion updates all agents' positions. Usually, a global clock controls how often the strategic and tactical models can interfere. Compare Algorithm 1. There are other methods that, for example, initialize new agents or delete agents from the simulation. Often agents spawn during a simulation run. However, for the sake of simplicity, they are not listed. How often the tactical and strategic levels interfere depends on the model. For example, a dynamic floor field requires a small Δt, and for each call, computation is necessary. This is also true for dynamic navigation fields. If a purely graph-based approach is used, Δt can be large, and for many calls, there is nothing to compute. Note that Δt is not the time step used to solve ODE-based models – it is usually larger.

Locomotion

For Locomotion, it is crucial which update scheme is used. Can we update each agent in parallel, or does the model impose some specific order? As argued, it seems reasonable that for large-scale simulations, there is some space for parallel motion and decision making. However, discrete-time models, such as cellular automata and optimal steps models, often impose an order to avoid collisions, that is, overlapping agents. In Chapter 5, I show how to introduce parallelism without violating these restrictions for optimal steps models. If discrete-time models use a parallel update scheme, they introduce some conflict resolution mechanism. For example, if two or more agents compete for the same cell of a cellular automaton, all but one randomly chosen agent have to step back to their last cell. In the case of ODE-based models such as social force models and the Gradient Navigation Model, integration is a parallel update scheme using a tiny time step [167]. Therefore, parallelization for these models is straightforward. Because there are no restrictions, they scale well. In summary, ODE-based models highly benefit from parallelism at the cost of a tiny time step. Especially in dense situations, the time step size has to be small to avoid inaccurate results and unrealistic behavior. In contrast, cellular automata and optimal steps models use larger time steps of approximately the duration of a footstep ($\Delta t = 0.5$ seconds). Cellular automata often impose a parallel update scheme while parallelization of optimal steps models requires more sophisticated strategies.

4.3 Review of parallel locomotion

The parallelization of the locomotion model enables large-scale simulations. Therefore, the LOCOMOTION method in Algorithm 1 should be realized by multiple processing units. In general, a partition of independent agents is required such that a set of agents can be updated in parallel. Since it is reasonable to assume that spatially far apart agents behave independently, spatial *domain decompositions* generate these partitions. There are two types of domain decomposition. *Static domain decompositions* use a fixed decomposition of the simulation domain, while *dynamic decompositions* change during the simulation. In the dynamic case, *load balancing strategies* distribute the work evenly to all processors such that the speedup is maximized. Besides the computational cost of the load balancing algorithm itself, higher communication costs between processors are introduced. Therefore, one has to balance costs against gains. The goal of most load balancing strategies is to decompose the space such that each processing unit updates approximately the same number of agents. The strategy assumes that the computational load per entity is the same. In the case of large-scale pedestrian simulation, this is not necessarily the case. If the density is high, the computation cost per entity is higher than if agents are far apart.

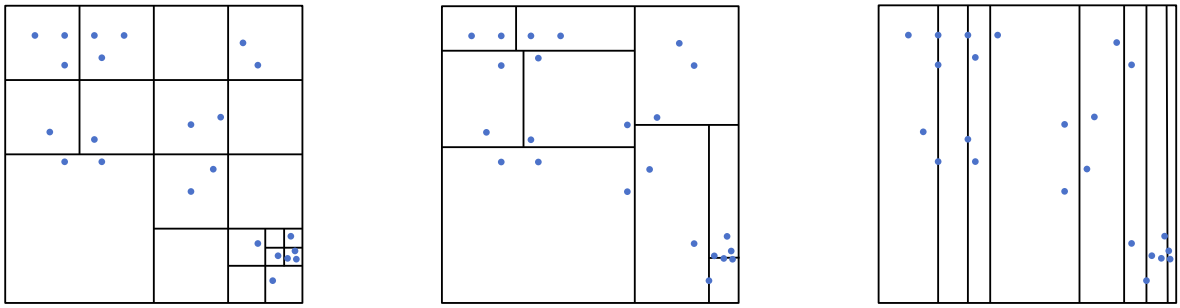
Regardless of the implemented decomposition and load balancing strategy, the most suitable parallelization technique for an application is usually coupled to the underlying hardware architecture. Researchers implemented their models for different hardware architectures from shared and distributed memory clusters, GPUs to specialized hardware like the PLAYSTATION 3. Even though hardware specifics, such as GPUs' memory structure, influence the implementation, we can still extract abstract techniques from concrete implementations.

In the following, I extract these techniques by looking at the contribution of the past. Since modelers are more interested in simulation outcomes than execution times and performance, I look beyond results from pedestrian dynamics. Especially in computer graphics and animations, researchers are less concerned with the correctness of the model and more concerned with realistic looking and fast simulations.

4.3.1 Large-scale agent-based animation

Steed et al. [279] compared different density-dependent domain decompositions for the animation of a large virtual urban environment. They experimented with a quadtree partitioning, k-d trees, and a growing region approach, a variation of an image partitioning algorithm. For the quadtree and k-d tree partitioning, the space covered by a set does not align with the domain, compare Figs. 4.1i and 4.1ii. If the pedestrian density in a set of the partition exceeds a certain maximum, it is split into multiple sets. Their growing region approach divides the domain into small, equally sized cells. Then it chooses some seed cells. For each seed cell, it adds adjacent cells until a threshold is reached. The authors found that the region growing approach works best. They hypothesized that the growing region approach tends to follow the likely connection between dense areas by the nature of its construction mechanism. The crowd navigation is based on the model introduced by Tecchia [282], which does not model realistic pedestrian behavior.

Zhou et al. [335] used a dynamic column-wise block-striped decomposition to partition the simulated space. They divide the domain at one dimension and adjust the width of the constructed



(i) (ii) (iii)

Figure 4.1: Three domain decomposition techniques: the quadtree approach splits each set for which the maximum pedestrian density is reached into four equally sized regions (i). The kd-tree approach allows a more irregular subdivision (ii). For the column-wise block-striped decomposition, a processor must communicate to at most two neighbors (iii). For each example, the regions are subdivided if they contain more than three pedestrians.

columns dynamically, compare Fig. 4.1iii.

Reynolds [234], introduced a model for autonomous agents. Two years later, he used the PLAYSTATION 3 to simulate 15,000 individuals and achieved a performance of 60 frames per second. He developed a static domain decomposition that divides the three-dimensional space into equally sized buckets. Each agent inside a specific bucket is updated by the same Synergistic Processor Units (SPUs). There are many more buckets than processors, such that no dynamic load balancing is required.

Cosenza et al. [51] used the flocking model, introduced by Reynolds [233], to simulate flocks of birds. They split the domain along one specific dimension. The authors tried to balance the size of each partition set and the number of agents in the buffer zone. They report being able to simulate up to one million birds.

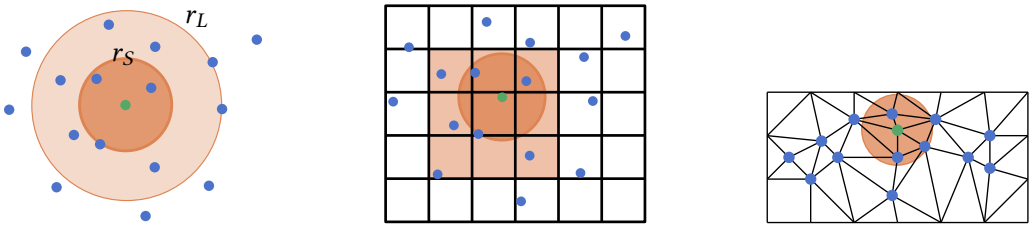
Guy et al. [110] present a collision avoidance algorithm based on velocity obstacles. The collision avoidance is realized by solving multiple optimization problems. Using parallel computation, the authors report that the algorithm is highly efficient.

Richmond et al. [235] introduced a high-performance framework for agent-based pedestrian dynamics and animation on the GPU. Agent variables are stored in the OpenGL texture. A linked cell approach decomposes the domain. It is split into several equally sized cells. The authors use a combination of the steering model [234] and the Social Force Model.

Another linked cell-based fish simulation implementation was introduced by Erra et al. [80]. To achieve fast execution times, they implement their model in CUDA. A similar mixture of models was used in [143]. Karmakharm et al. [143] use the Flexible Large-scale Agent Modeling Environment (FLAME) [10], a GPU framework to simulate up to 10,000 agents.

In [325], Yilmaz gave insights into the potential of using GPU for crowd simulation. Their fuzzy logic approach implemented with CUDA can be used to simulate one million agents in real-time. They report a speedup of simulation runs by 100 using a GPU.

In [228], Rahman et al. accelerated the well-known agent-based simulator OpenSteer by using CUDA. OpenSteer is a C++ library to help construct steering behaviors for autonomous characters in games and animation [6].



(i) (ii) (iii)

Figure 4.2: Nearest-neighbor data structures: the verlet list (i) stores for each agent all neighbors inside the circle defined by r_L, that is larger than the influence radius r_S. After some simulation time, the list is updated. The linked cell data structure (ii) consists of equally sized cells. Each cell contains the agents of the space it represents. Finding all neighbors of an agent is solved by iterating over neighboring cells in the breadth-first order. Another data structure that supports the nearest-neighbor search is the Delaunay triangulation (iii). To find the nearest-neighbors closer than some threshold, one can use a breath-first search strategy.

4.3.2 Large-scale force-based simulation

Interestingly, one of the first large-scale implementations of a pedestrian model was already presented by Quinn et al. [227] at the second International Conference in Pedestrian and Evacuation Dynamics in 2003. The authors introduced a parallel distributed memory implementation of the Social Force Model [118], making use of the well-known Message Passing Interface (MPI). Quinn et al. parallelized the Social Force Model by a primary-and-secondary-approach. The primary process reads in the domain and population and sends this information to the appropriate secondary processes. The primary process gathers data from the secondary processes at each time step and passes this information to the rendering engine. The whole spatial domain is split into squared sub-domains. Secondary processors are interconnected on a two-dimensional virtual grid. As long as agents move inside the sub-domain controlled by one processor and are not inside any ghost area near the sub-domain boundary, no communication to other worker processes is necessary. If agents reach the domain boundary, it is necessary to communicate details to the neighboring processes. If an agent moves to another sub-domain, its processor must hand over that agent to the processor of this sub-domain. To update an agent inside a sub-domain by its processor, the authors cut all long-range forces. That is, they only consider social forces from agents inside the same or neighboring square cells. The authors report a simulation of 10,000 pedestrians, which can be updated 50 times per second. They use a Linux-based multicomputer at Oregon State University consisting of 2.4 GHz Intel Xeon CPUs connected by a gigabit Ethernet switch.

In 2014, Mroz and Was [204] used a similar decomposition for a GPU implementation of the Social Force Model. They divided the domain into 15 m times 15 m sub-regions. For 100,000 agents, they measured a speedup of 5 compared to their CPU implementation. Their hardware setup consisted of a GeForce GTS 250 with 512 MB RAM and a Dual-Core AMD Athlon II 250 (3.00 GHz), 4 GB of RAM.

Kemloh [144] presented another large-scale implementation of a force-based model. He parallelized the Generalized Centrifugal Force Model by developing a distributed memory imple-

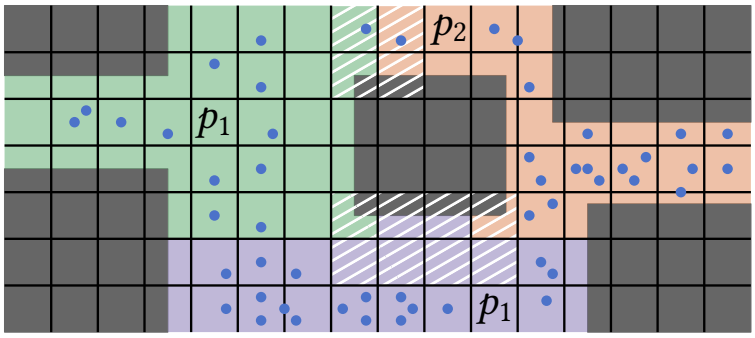


Figure 4.3: Linked cell-based domain decomposition: gray cells represent the linked cell data structure distributed among three processors. The different colors indicate the set of cells of each processor. Ghost cells are highlighted in white.

mentation based on the linked cell algorithm. The linked cell data structure is combined with a verlet list, compare Figs. 4.2i and 4.2ii. Both data structures and similar techniques are commonly used in molecular dynamics [280, 12]. To enable domain decomposition, Kemloh also assumes some cut-off radius of the obstacle and pedestrian forces. He suggested using a cut-off radius of 2 m. Therefore, the entire spatial domain is divided into cells of 2 m times 2 m. Each processor is responsible for a set of connected cells. If a neighboring cell belongs to another computing node, the cell is part of the ghost area. Ghost areas are copied to multiple processors and require communication to be consistent. To achieve real-time simulations, the Message Passing Interface is used to exploit multiple computing nodes. The simulation code on each node is run in a multi-threaded environment using Open Multi-Processing. Kemloh reported a speedup of 9 using 12 processors simulating a scenario of 10,000 uniformly distributed agents in a 50 m times 50 m room.

Löhner et al. [192] introduced a parallel implementation of his PEDFLOW model [190], another force-based model. In their publication we find an interesting strategy to compute the nearest-neighbors relation different from the linked cell approach. Given the set of agents $\mathcal{A}$, Löhner et al. suggest constructing the constrained Delaunay triangulation of $n_{\mathcal{A}} = |\mathcal{A}|$ positions. They use the triangulation as the basis of the nearest neighbors search, compare Fig. 4.2iii. Dickerson and Drysdale [65, 67, 66] introduced the idea of searching for nearest neighbors using the Delaunay triangulation. Löhner et al. [192] also presented a technique to store spatial domain-specific and dynamic information on a background mesh. The mesh can be partitioned and distributed among multiple processors. Shared and distributed memory parallelism is introduced with Open Multi-Processing and the Message Passing Interface, respectively. At first, the authors implemented a static domain decomposition. They report being able to compute the movement of a million pedestrians in real-time. Later, Löhner et al. [193] added a simple dynamic load balancing strategy.

Algorithm 2: ParallelCASimulationRun

```
1 Initialization();
2 while simulation is running do
3     foreach cell j of the grid of cells in parallel do
4         compute diffusion and decay of j for Δt;
5     foreach agent l ∈ 𝒜 in parallel do
6         compute next cell of l for Δt;
7     foreach agent l ∈ 𝒜 in parallel do
8         if there is no conflict for l then
9             move l;
```

4.3.3 Large-scale cellular automata simulation

Since one can use much larger time steps, cellular automata are usually more efficient than any other type of microscopic pedestrian model. Therefore, it is no surprise that researchers use cellular automata for large-scale simulations. If the cellular automaton uses a parallel update scheme and the floor field technique described in Section 2.2.1, the simulation can be parallelized on a cell level. Algorithm 1 can be rewritten as Algorithm 2.

In 2009, Kretz [170] used a cellular automaton to simulate up to 182,000 agents in real-time on an eight-core machine (2 Xeon E5320 quadcore processors and 20 GB RAM). The model [171] developed by him and Schreckenberg is based on a parallel update scheme. Agents can only move to their next position if their path does not intersect with any other agent.

Since cellular automata are based on the grid of cells, an inherently regular matrix-like data structure, GPUs are suitable for accelerate cellular automata. They offer thousands of processing cores at an affordable price.

In [205, 204], Mróz et al. discussed the potential of GPUs and introduced a CUDA implementation of their Social Dinstance Model [307]. They initialize their floor fields, geometry and other objects using the CPU. In the second step, data is transferred to the GPU's global memory. The update phase of Algorithm 2 runs entirely on the GPU. Their GPU implementation outperforms a CPU implementation of the Social Force Model. In a follow-up work [309], the authors included some hardware-specific topics such as avoiding branch instructions.

In 2018, Kłusek et al. [159] extended their implementation to support multiple GPUs using a domain decomposition similar to the one depicted in Fig. 4.3. Instead of cells of a linked cell data structure, the decomposition is based on the cellular automaton cells. Ghost cells require communication across multiple GPUs. In addition to the data parallelism, the author introduced task parallelism using a specific GPU that computes the floor field.

Lately, Renc et al. [232] adapted the Social Dinstance Model to integrate it into the Xinuk framework. Xinuk is a high-performance computing framework with desynchronized information propagation for large-scale simulations [38].

4.4 Summary

In this chapter, I discussed the pedestrian dynamics as an algorithmic problem. First, I established pedestrian dynamics as a complicated and complex problem. I argued that on higher levels, with respect to the hierarchical model approach discussed in Section 4.1, problems get more complicated. In contrast, on the operational level, we deal with a complex system, consisting of many but homogeneous entities. Based on my differentiation of complex and complicated parts, it follows that operational models offer the most parallel potential and computational workload.

In Section 4.2, I emphasized the importance of parallelism to enable large-scale real-time simulations. Furthermore, I argued that one can find natural parallelism in pedestrian dynamics. My claim relies on independent acting pedestrians that is similar to data independence required for parallelization. I gave an abstraction of implementations of operational models and a more concrete assessment of how we can find and exploit parallelism for a specific model types.

In Section 4.3, I became even more concrete by presenting existing large-scale implementations. I looked into agent-based animation to find data structures that one can use in pedestrian simulation. There is also a lot of work with respect to large-scale force-based simulations and large-scale cellular automata simulation. I highlighted data structures and algorithms that establish data independence, because they play the most crucial role to establish parallelism.

CHAPTER 5

Parallel optimal steps models

"Play is the exultation of the possible."

– Martin Buber

In this chapter, I use the parallel nature of pedestrian behavior and introduce parallelism to optimal steps models to make acceleration possible. In Section 4.3, I discussed parallelization techniques introduced by numerous researchers. Parallelism was brought to cellular automata, ODE-based and other models. I argued that computational costs of optimal steps models are in between cellular automata and force-based models. And that they offer the potential for fast large-scale simulations. The *optimal steps* principle of optimal steps models gifted us large (individualized) time steps. But finding the next optimal position via optimization requires multiple evaluations of exponential functions. Additionally, optimal steps models operate inherently sequential since they rely on an event-driven update scheme.

If model implementations enforce a sequential update of agents, they do not scale with the number of agents, and we eventually run into performance issues. Today, clock frequencies of single central processing units no longer increase significantly. Therefore, the computational burden has to be divided amongst multiple CPUs, and parallelism becomes mandatory. Without it, increasing the number of simulated pedestrians leads eventually to lengthy unbearable simulation times even for high-performance computing (HPC) hardware systems.

In the following, I show that even the inherently sequential update order of optimal steps models feature parallelism. The key problem of parallelism is to reduce data dependencies in order to be able to perform computations on independent computation units. In the case of optimal steps models we have to deal with dependencies between different *footstep events*. *Footsteps* that are processed spatially distant but temporally close are likely to be independent of one another. I introduce (Section 5.2) and analyze (Section 5.3) a parallel version of the event-driven update scheme called ParallelEventDrivenUpdate. To enable future implementations by other researchers and developers, the algorithm description is technical. The algorithm's underlying idea is to identify independent events without introducing noticeable additional computational costs. I designed the algorithm to run on single-instruction multiple date hardware systems. In Section 5.4, I examine its potential using an OpenCL implementation executed on a graphics processing unit (GPU). Before introducing the parallelism, I shed light on why the update order of agents matters (Section 5.1).

5.1 Update schemes

5.1.1 The event-driven update scheme

Let us first look at the event-driven update scheme more closely in order to understand what restrictions it imposes and what behavior it evokes. The scheme processes events in their natural order, that is, the way they occur. Let t be the simulation time. Each agent l performs a series of *footstep events*

$$((t_{l,0}, \mathbf{x}_{l,0}), (t_{l,1}, \mathbf{x}_{l,1}))_{l,0}, \ldots, ((t_{l,k}, \mathbf{x}_{l,k}), (t_{l,k+1}, \mathbf{x}_{l,k+1}))_{l,k}. \tag{5.1}$$

Definition 5.1 (footstep)**.** Let l be an agent of a simulation using a optimal steps model. Then

$$(e_{l,i}, e_{l,i+1})_{l,i} = ((t_{l,i}, \mathbf{x}_{l,i}), (t_{l,i+1}, \mathbf{x}_{l,i+1}))_{l,i} \in \mathbb{R}^+ \times \mathbb{R}^2 \times \mathbb{R}^+ \times \mathbb{R}^2 \tag{5.2}$$

is the i-th *footstep event* of agent l, if and only if it moves at $t_{l,i}$ seconds from $\mathbf{x}_{l,i}$ to $\mathbf{x}_{l,i+1}$ and is able to move again at $t_{l,i+1}$ seconds (simulation time). The *footstep event* occurs at time $t_{l,i}$ and position $\mathbf{x}_{l,i}$. I call $e_{l,i} = (t_{l,i}, \mathbf{x}_{l,i})$ the *event start* and $t_{l,i}$ *event (start) time*.

Optimal steps models ensure that for the choice of the next *footstep* at t, all *footstep events* that start at $t_{l,i} < t$ have already been processed. In the simulation software framework Vadere [157, 294] the order is enforced by a priority event queue Q that contains the next *event start* for each agent. Whenever a new agent spawns its first *event start* is added to Q. The queue is sorted by the *event (start) time*. After executing one *footstep*, the *start* of the next one is known, compare Algorithm 3. Events are executed in an instant, one after another. If agents do not adjust their desired speed, their stepping frequency remains the same for the whole simulation thus we can compute the *event (start) time* for future *footsteps* in advance. However, we only know the time. The place, on the other hand, is determined by the dynamics of the simulation. The event time line of an example for three agents is depicted in Fig. 5.1. Note that, because of the time discretization and update scheme, optimal steps models are, in fact, discrete event simulation (DES) models.

From a modeling perspective, the natural order combined with instantaneous updates leads to deliberate anticipation – agents anticipate currently processed *footsteps* of others nearby, because the agent utility $u_{\mathcal{A},l}$ depends on the agents' position of the very near future. On the one hand, the anticipation of others is an essential property of the model. On the other hand, the strict event order imposes a significant obstacle to parallelism.

Algorithm 3: SequentialEventDrivenUpdate

```
while Q.min() < t + Δt do
    e_{l,i} = (t_{l,i}, x_{l,i})_{l,i} ← Q.min();
    e_{l,i+1} ← Execute(e_{l,i});
    Q ← Q \ {e_{l,i}};
    Q ← Q ∪ {e_{l,i+1}};
t ← t + Δt;
```

5.1.2 The parallel update scheme

One way to introduce parallelism is to violate the order of the event-driven update scheme. Seitz et al. [258] experimented with such a violation by implementing and evaluating a parallel update scheme. The globally synchronizing clock mainly determines its *footstep event oder*, compare Algorithm 1. After an increase of the clock by a fixed time step Δt, the parallel update processes all footstep events within $[t; t + \Delta t)$ in parallel. Anticipation is reduced, and, for large enough Δt, it vanishes completely. Agents are blind for the other's behavior during $[t; t + \Delta t)$. Consequently, one has to deal with collisions since multiple agents might compete for the same area. To

Algorithm 4: ParallelUpdate

```
1 do
2 |   E_t ← Seek(A) for each agent in parallel;
3 |   A ← Move(A,E_t) for each agent in parallel;
4 |   t ← t + Δt;
5 while E_t ≠ ∅;
```

resolve this conflict, the parallel update scheme (Algorithm 4) consists of the following steps: Seek computes the next desired position for all agents in parallel. In the second step, Move moves the agent if its event time is the smallest among all competing agents. Agents are competing if and only if their bodies overlap. Seek and Move are called as long as there are competing agents.

5.1.3 The loss of anticipation

The parallel update scheme

> *"has the objective of improving computational performance and should not influence the model's behavior and macroscopic simulation outcome, such as evacuation times."*
> – Seitz et al. [258]

It produces the same result as the event-driven update scheme if Move only affects one agent, that is, if Δt is sufficiently small. However, in that case, it has no computational advantages over

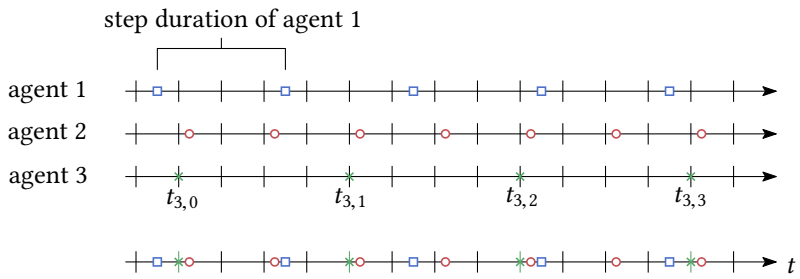


Figure 5.1: Event processing time line: at the top three individual timelines, one for each agent, are displayed. At the bottom, the overall event processing time line for the simulation is illustrated.

the event-driven update scheme. In [258], the authors showed that even for a small time step ($\Delta t = 0.2$ seconds), the parallel update scheme produces significantly larger evacuation times.

> *"Therefore, update schemes change the model in a way that has an impact on macroscopic measures."* – Seitz et al. [258]

Agents tend to require more time to walk from one place to the other. This indicates that they use sub-optimal paths or interrupt their motion because they lose some of their ability to anticipate others' behavior. At first glance, it seems that the combination of Seek and Move imposes similar anticipation since the *event* with the earliest *event time* is processed first. However, non-competing agents are partially blind to others. They might move far more close to others than intended, and consequently, disrupt personal spaces of others. This blind spot leads to unwanted oscillations that were not mentioned in [258]: at time t an agent l might move far too close to another agent l'. For its next *footstep*, agent l is pushed away from l' since it is no longer blind for l' – its utility function has changed accordingly. This combination of actions is repeated. I observed a back and forth motion for a significant portion of agents for different scenarios, especially in crowded areas.

Seitz et al. suggests that

> *"one can use the parallel update scheme for scenarios with high performance requirements, [... but] it might be necessary to further calibrate the model to obtain the same results as for the event-driven update. [... One should use] either the event-driven scheme (for accuracy) or the parallel scheme (for computational efficiency) depending on the requirements."* – Seitz et al. [258]

In my opinion, this suggestion is a valid assessment. Still, we have to keep in mind that agent behavior does not necessarily converge towards the event-driven update's motion if we use smaller and smaller Δt. Even a few agents updated in parallel can impact microscopic simulation outcomes, that is, the overall set of processed *footstep events*. For many scenarios, this would not lead to significant changes in macroscopic outcomes. However, the observation made in [258] does not lead to a generalization of this assumption. The anticipation imposed by optimal steps models is a vital model property – anticipation was discovered in [74] for pedestrians moving towards a bottleneck. Dismantling it transforms optimal steps models into something else. Therefore, it is not apparent that calibration will fix the gap between parallel and event-driven update schemes for all possible scenarios.

5.2 Algorithm design

Instead of relying on a parallel update scheme, I propose to parallelize the event-driven update scheme. We published parts of this contribution in [337]. My approach relies on the critical observation that the influence range of agents is limited. Aside from the medium-scale navigation realized by dynamic navigation fields, agents are influenced by others due to short-range agent utilities $u_{\mathcal{A},l}$, compare Eq. (2.23) in Section 2.2.6. If two agents are distant from each other, they do not affect each other locally. Consequently, it is most likely but not certain that many agents

in a large-scale simulation act independently from another, primarily if they are spread evenly across the domain Ω. The challenge is to identify independent agents efficiently to update them in parallel without introducing any new significant computational bottlenecks. Therefore, the identification has to be a cheap parallel process.

```
class Agent {
  float x;
  float y;
  float eventTime;
  float speed;
  float strideLength;
}
```

Listing 5.1: Entry of the array of structures.

```
class Agents {
  float[] x;
  float[] y;
  float[] eventTime;
  float[] speed;
  float[] strideLength;
}
```

Listing 5.2: The agent's structure of arrays.

5.2.1 The parallel linked cell data structure

Imagine the next event of agent l' occurs after the event of some other agent l. We have to process agent l in advance of l' if the update of agent l changes the agent utility $u_{\mathcal{A},l'}$ within the step circle or disc of agent l'. Let v_l be the maximum desired speed of agent l, $\mathcal{A}$ be the set of all agents, and let $t_{l,i+1} - t_{l,i}$ be its step duration. Then

$$\gamma_{l,\max} = v_l \cdot (t_{l,i+1} - t_{l,i}) \cdot \lceil \Delta t / (t_{l,i+1} - t_{l,i}) \rceil \tag{5.3}$$

is the maximum distance $\gamma_{l,\max}$ agent l can move by executing events occurring between t and $t + \Delta t$. Furthermore, let

$$\gamma_{\max} = \max_{l \in \mathcal{A}} \gamma_{l,\max} \tag{5.4}$$

be the maximum distance any agent can be moved within a time step. And let δ_w be the width of the utility function of our optimal steps model. If two agents are more than

$$\delta_c = (2 \cdot \gamma_{\max} + \delta_w), \tag{5.5}$$

meters apart, they do not influence each others behavior locally. I multiply $\gamma_{\max}$ by two because agents might move directly towards each other.

To decide if agents act independently, I test if nearby agents move earlier. This requires distance comparison of nearby agents. To avoid a quadratic time complexity of $O(|\mathcal{A}|^2)$, I decided to base my implementation on the linked cell data structure. It is one of the data structure mentioned in Section 4.3 and enables efficient nearest neighbor requests [313]. It is highly regular, thus suitable for single instruction multiple data (SIMD) hardware architectures. Let w_Ω, h_Ω be the width and height of a tight bounding rectangle enclosing the whole simulation domain Ω and let δ_c be the cell size of the linked cell data structure. I divide the space into

$$n_c = \lceil w_\Omega / \delta_c \rceil \cdot \lceil h_\Omega / \delta_c \rceil = w_c \cdot h_c \tag{5.6}$$

cells, uniquely numbered from 0 to $n_c - 1$. I choose δ_c so that for a given cell, it suffices to consider only agents in its Moore neighborhood to compute the next position of any agent within the cell. If a cell size

$$\delta_c = (2 \cdot \gamma_{\max} + \delta_w) \tag{5.7}$$

is used and we update the data structure every Δt seconds, we can identify dependent agents of agent l by only looking at the Moore neighborhood of the cell containing agent l. In the worst case, two agents move directly towards each other, closing a gap of at most $2 \cdot \gamma_{\max}$ meters. If they were still at least δ_w meters apart, their moves would not be influenced by each other locally.

Constructing the linked cell data structure on SIMD hardware architectures requires more sophisticated techniques than just managing an array of lists since one relies on highly regular, that is, indexable structures. I implemented the technique presented in [80, 105]: let $n_{\mathcal{A}} = |\mathcal{A}|$ be the number of agents. Multiple arrays realize the linked cell data structure. An integer array $\mathbf{I}$ of size $n_{\mathcal{A}}$ containing all agent ids $0, \ldots, n_{\mathcal{A}} - 1$. Ids are sorted according to the agent's cell. For example, ids of agents contained in cell 0 are at the front of the array, while at its tail, we find ids of agents contained in the last cell. A second array of structures (AoS) contains the actual agents' data. Later I will discuss a GPU implementation that uses a structure of arrays (SoA) instead, compare Listings 5.1 and 5.2. Each array of the structure is sorted in the same way. Accessing all agents contained in a cell requires two additional arrays $\mathbf{C}_{\text{start}}$, $\mathbf{C}_{\text{end}}$ of size n_c. $\mathbf{C}_{\text{start}}[i]$ is the smallest index of $\mathbf{I}$, such that $\mathbf{I}[\mathbf{C}_{\text{start}}[i]]$ is part of cell i. Similarly $\mathbf{C}_{\text{end}}[i] - 1$ is the largest index for that the agent, identified by $\mathbf{I}[\mathbf{C}_{\text{start}}[i]]$, is part of cell i. To access all of the $\mathbf{C}_{\text{end}}[i] - \mathbf{C}_{\text{start}}[i]$ agents in cell i, I access $\mathbf{I}[\mathbf{C}_{\text{start}}[i]], \ldots, \mathbf{I}[\mathbf{C}_{\text{end}}[i] - 1]$. I set $\mathbf{C}_{\text{start}}[i]$ and $\mathbf{C}_{\text{end}}[i]$ to -1, if cell i is empty. To construct and sort $\mathbf{I}$, $\mathbf{C}_{\text{start}}$, $\mathbf{C}_{\text{end}}$ and either the array of structures or structure of arrays, the following steps are necessary:

HASH: computes the cell id $w_c \cdot \lfloor y/\delta_c \rfloor + \lfloor x/\delta_c \rfloor$ for each agent positioned at $\mathbf{x} = (x, y)$ in parallel and stores it in $\mathbf{C}$,

SORT: sorts cell ids $\mathbf{C}$ applying the parallel bitonic sort algorithm. At the same time, agent ids $\mathbf{I}$ are sorted in the same way,

ORDERING: rearranges agent properties, that is, the AoS or SoA according to $\mathbf{I}$ in parallel,

FIND: constructs $\mathbf{C}_{\text{start}}$ and $\mathbf{C}_{\text{end}}$ by detecting unequal consecutive cell ids in $\mathbf{C}$ in parallel.

The construction is depicted in Fig. 5.2. The reordering of agent properties does not only simplify the access to nearby agents but additionally increases the cache hit rate during the following computation steps of the cycle.

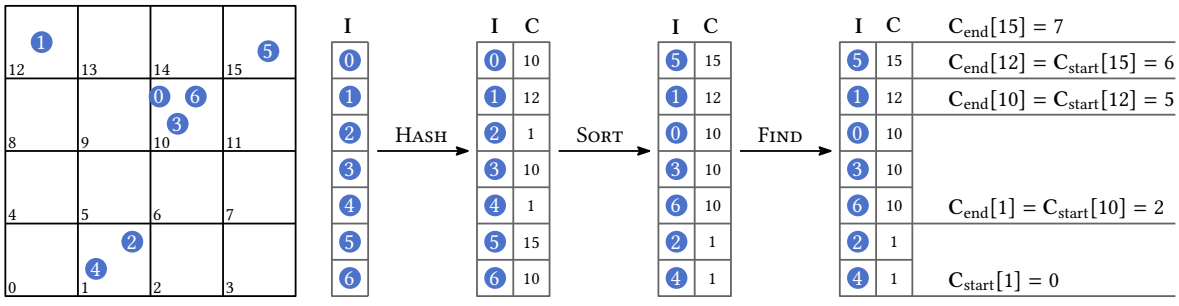


Figure 5.2: Construction of the linked cell data structure with 4×4 cells and $n_{\mathcal{A}} = 7$ agents: the square on the left covers the spatial domain. Agents numbered from 0 to 6 are depicted in blue. After HASH computes cell ids based on the agent's position, SORT sorts them. Finally, FIND detects consecutive changing cell numbers in $\mathbf{C}$ and constructs $\mathbf{C}_{\text{start}}$ and $\mathbf{C}_{\text{end}}$.

Constructing the linked cell data structure sequentially by using an array of dynamic lists requires $O(n_{\mathcal{A}})$ time. Using only arrays and primitive data types as presented requires sorted agent ids. The sorting is the dominant factor and requires $O(n_{\mathcal{A}} \cdot \log(n_{\mathcal{A}}))$ sequential time but only $O(\log^2(n_{\mathcal{A}}))$ parallel time. Therefore, the described construction requires overall $O(\log^2(n_{\mathcal{A}}))$ parallel time.

5.2.2 Identification of independent events

After the linked cell data structure is constructed, I use it to identify independent footstep events applying a two-phase parallel filtering technique.

Cellfilter

First, I invoke CELLFILTER for each cell of the linked cell data structure. It iterates over all agents of a specific cell and filters the agent with the earliest event time $t_{l,i} \leq t + \Delta t$. Its id is written into an array $\mathbf{E}'$ of size n_c. If no agent was found, which happens if the cell is empty, -1 is written instead, compare Fig. 5.3.

CELLFILTER requires $O(n_{\mathcal{A},c})$ parallel time where $n_{\mathcal{A},c}$ is the number of agents of the most populated cell. $n_{\mathcal{A},c}$ is bounded by some constant since optimal steps models avoid overlaps of agent bodies. Therefore, the parallel time complexity of CELLFILTER is theoretically independent of the overall number of agents $n_{\mathcal{A}}$. It is known that the density of a maximum packed arrangement of agent bodies (circles) is

$$\frac{\pi\sqrt{3}}{6} \approx 0.9069. \tag{5.8}$$

If we multiply the area of a cell by this number, we get the space that can be occupied by an agent body. Consequently,

$$n_{\mathcal{A},c} \leq \frac{\text{maximal space occupied by agents}}{\text{area of the agent body}} = \frac{\sqrt{3}\pi \cdot (\delta_c + \delta_{\text{tor}}/2) \cdot (\delta_c + \delta_{\text{tor}}/2)}{6\pi(\delta_{\text{tor}}/2)^2} \tag{5.9}$$

holds, where δ_c is the cell size (width and height of a cell) and δ_{tor} the diameter of the agent's torso. For example, if one uses the potential function proposed in [260] with a width of $\delta_w = 0.5$, the torso diameter of $\delta_{\text{tor}} = 0.4$ and a maximum step length of $\gamma_{\max} = 1.1$ (compare [257]) we get $\delta_c = (2 \cdot 1.1 + 0.5) = 2.7$. Then the number of agents contained in a cell is bounded by

$$\frac{\sqrt{3}\pi \cdot (2.7 + 0.2) \cdot (2.7 + 0.2)}{6\pi 0.2^2} \approx 60.69, \tag{5.10}$$

if Δt is smaller than the step duration of any agent. In that case, the pedestrian density would be approximately 8.33 agents per square meter.

Gridfilter

After applying CELLFILTER, I invoke the second filter, called GRIDFILTER, for each cell in parallel. It filters left-over conflicting footstep events. It replaces the agent ids in $\mathbf{E}'$ by -1 if there is an agent in the Moore neighborhood with a smaller *event (start) time*.

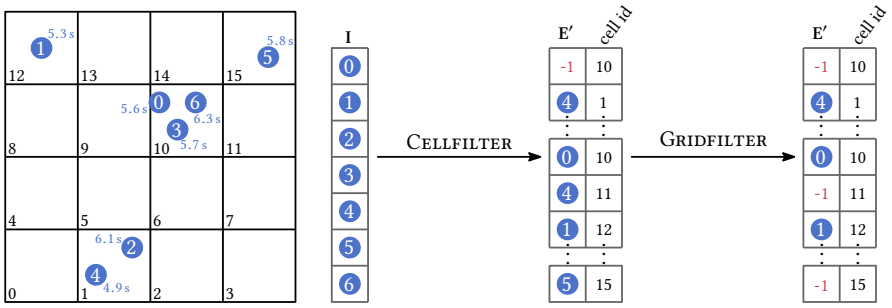


Figure 5.3: Construction of $\mathbf{E}'$ using the situation depicted in Fig. 5.2 by invoking CELLFILTER and GRIDFILTER consecutively: white highlighted numbers represent the agent ids. The first array is constructed by CELLFILTER. For each cell, the agent with the earliest event time (blue) is written into the array. In this example, GRIDFILTER filters agent 5 because the event of agent 0 starts earlier.

Due to the first filter phase, each thread has to test 8 footstep events. Thus its parallel time complexity is $O(1)$.

Align

After applying CELLFILTER and GRIDFILTER, a large integer array $\mathbf{E}'$ of size n_c contains the agent ids for independent agents and many entries are equal to -1. The ALIGN procedure computes an array $\mathbf{E}$ that only contains ids of agents that can be updated in parallel.

To compute the array size $|\mathbf{E}|$, I use a modified prefix sum adapting the parallel algorithm presented by Harris et al. [114]. Instead of summing everything up, I only add 1 if the array value is non-negative. Additionally, I compute a second prefix sum array $\mathbf{E}^-$ ignoring all positive values. Since all agent ids are non-negative and all other array entries are set to -1, the integer $-\mathbf{E}^-[j]$ is equal to the number of cells with an id smaller than j that do not participate in any movement update. It follows that $j + \mathbf{E}^-[j]$ is equal to the number of cells with an id smaller than j which are affected by changes. Let i be the id of some of the n_c threads. Then by executing the following assignment for all threads in parallel generates the desired array $\mathbf{E}$:

$$\mathbf{E}[i + \mathbf{E}^-[i]] \leftarrow \mathbf{E}'[i], \text{ if } \mathbf{E}'[i] \geq 0, \tag{5.11}$$

compare Fig. 5.4.

The parallel time complexity of ALIGN is dominated by the computation of prefix sums and thus equal to $O(\log(n_c))$.

5.2.3 Update of independent agents

After having identified independent agents, we can solve the optimization stated in Eq. (2.12) for each agent in $\mathbf{E}$ in parallel. However, to increase parallelism further, I propose a simple brute force strategy that is especially suitable for SIMD hardware. It relies on a finite set of possible next positions. Therefore, if the optimal steps model optimizes on a connected infinite set of points (for example a disc), the set is approximated. In case of a disc, it is approximated by multiple step

Algorithm 5: ParallelBruteForceOptimization

```
   /* evaluate utilities                                          */
1  j ← thread group id (and event id);
2  i ← thread id (and possible next position id);
3  l ← E[i] // agent id
4  X[j][i] ← x_l + P[i] · r_l;
5  U[j][i] ← u_l(X[j][i]);
6  thread group barrier // wait for all |P| threads
   /* find maxima                                                 */
7  X[j][0] ← ReduceMax(X[j], U[j]) // executed by |P|/2 threads
8  if i = 0 then
       /* update the agent position and event time               */
9      UpdateAgent(A[l], X[j][0], E[j])
```

circles: let R be the number of step circles and K be the number of points we want to use for the largest circle step circle. Then in case of a disc,

$$P = \left\{ r' \cdot (\cos(\theta), \sin(\theta)) \;\middle|\; r' = 0, \frac{1}{R}, \ldots, 1, k = 0, \ldots, \lceil R/(r' \cdot K) \rceil, \theta = \frac{2\pi}{k}, \right\}, \tag{5.12}$$

is the *general set of possible positions*, compare Fig. 2.9ii in Section 2.2.6. Let r_l be the radius of the step circle of agent l and $\mathbf{x}_l$ its current position, then I define

$$P_l = \mathbf{x}_l + P \cdot r_l = \{\mathbf{x}_l + \mathbf{y} \cdot r_l : \mathbf{y} \in P\} \tag{5.13}$$

to be the *set of next possible positions* of agent l. Let $\mathbf{P}$ be an array containing all positions of P, then Algorithm 5 solves the optimization problem and executes all $|\mathbf{E}|$ events. The pseudo-code gives the perspective of one particular thread. The thread computes one utility value and takes part in finding the maximum. ParallelBruteForceOptimization is executed by $|\mathbf{E}| \cdot |P|$ threads grouped into $|\mathbf{E}|$ thread groups. In line Line 7, the parallel reduction ReduceMax($\mathbf{X}[j], \mathbf{U}[j]$) returns the position for that the utility is maximal. It requires $\mathcal{O}(\log(|P|))$ parallel time which is the dominant factor. Therefore, the overall parallel time complexity to solve the optimization is also $\mathcal{O}(\log(|P|))$.

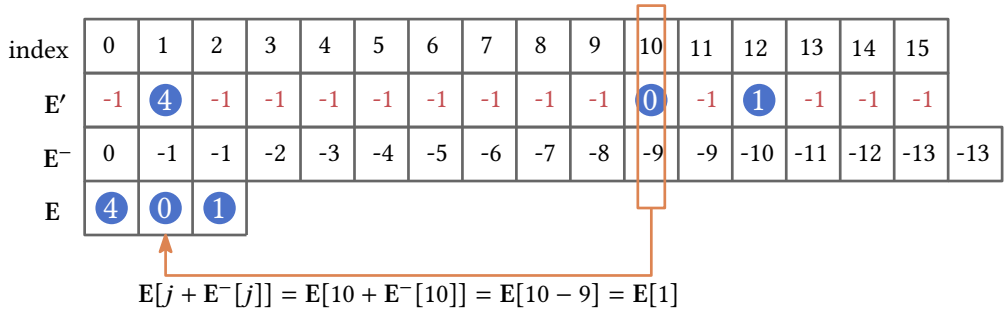


Figure 5.4: Construction of $\mathbf{E}$ using the situation depicted in Fig. 5.3 by invoking Align after the kernel function Gridfilter has finished: blue highlighted numbers represent agent ids.

Algorithm 6: ParallelEventDrivenUpdate

```
1  C ← Hash(𝒜, I) for agent in parallel;
2  C, I ← Sort(C, I) in parallel;
3  𝒜 ← Ordering(𝒜, I) in parallel;
4  C_start, C_end ← Find(I) for agent in parallel;
5  do
6      E′ ← Cellfilter(𝒜, t + Δt) for each cell in parallel;
7      E′ ← Gridfilter(𝒜, E′) for each cell in parallel;
8      E ← Align(E′) for each cell in parallel;
9      𝒜 ← ParallelBruteForceOptimization(𝒜, E, P) in parallel;
10 while E ≠ ∅;
11 t ← t + Δt
```

To exploit fast shared memory, ParallelBruteForceOptimization is executed by $|\mathbf{E}| \cdot |P|$ threads grouped into $|\mathbf{E}|$ (distributed) processors (thread groups). The i-th thread of the j-th thread group computes $u_{\mathbf{E}[j]}(\mathbf{x}_i)$ where $\mathbf{x}_i$ is the i-th possible next position. All intermediate results are saved into shared memory. Therefore, each thread group requires $(|P| \cdot 3 \cdot 4)$ bytes of local memory: $(2 \cdot 4)$ bytes for each point in P and four bytes to save each utility evaluation. After all threads completed their task, the final next position is computed by a parallel reduction using $\lceil |P|/2 \rceil$ threads (Line 7 of Algorithm 5) that finally solves the optimization defined in Eq. (2.12).

The identification and update of independent *footstep events* are repeated until $\mathbf{E}$ is empty. The ParallelEventDrivenUpdate is depicted in Algorithm 6.

5.3 Algorithm analysis

Let E be *footstep events* occurring within $[t; t + \Delta t)$ and let $E_j \subseteq E$ be the events ParallelEventDrivenUpdate updates in parallel for one *update cycle* (Line 6 to Line 9 in Algorithm 6) $j = 1, \ldots k$. If $|E| > 0$,

$$1 \leq |E_j| \leq n_c/4 \tag{5.14}$$

holds for all j, where n_c is the number of cells. We can easily construct an example of agent and event arrangement for that the parallelized event-driven update scheme falls back to a sequential update. One such example is a queue where each agent steps forward after the person in front moved ahead. The upper bound of $n_c/4$ parallel executable events is shown by the optimal tessellation pattern in Fig. 5.5. How much work can be done in parallel depends on the distribution of agents (spatially) and event times (temporally). Large and sparsely populated scenarios offer the most parallel potential, and dense areas reduce it since more agents conflict with one another. In the following, I estimate the expected parallelism for the proposed algorithm.

5.3.1 Theoretical considerations

In the following, I build a model to estimate how many agents can be updated in parallel for a disadvantageous crowded situation. More precisely, I compute the expected number of updates and reveal the proposed algorithm's parallel potential.

Let us look at the simulation state at some point in a simulation run. Let us assume all *start event times* are uniformly distributed, and each cell contains m events (agents) such that

$$|E| = m \cdot w_c \cdot h_c = m \cdot n_c, \tag{5.15}$$

where n_c is the number of cells, w_c is the number of cells in x-direction (columns) and h_c the number of cells in y-direction (rows). Additionally, let us ignore cells at the boundary – I assume each cell has eight neighbors in its Moore neighborhood. Let

$$A_{x,y,j} = \{\text{an event at cell } (x, y) \text{ is processed in } \textit{cycle } j\} \tag{5.16}$$

be a random event, $I_{A_{x,y,j}}$ be its indicator function, and $X_{x,y,j}$ be the random variable for the earliest event time for cell (x, y). The probability that an event in cell (x, y) is executed in the first cycle

$$\mathbf{Pr}\left(A_{x,y,1}\right) = \mathbf{Pr}\left(\bigcap_{\substack{\Delta x=-1,0,1,\\ \Delta y=-1,0,1}} \left\{X_{x,y,1} \leq X_{x+\Delta x,y+\Delta y,1}\right\}\right) = \frac{1}{9} \tag{5.17}$$

is independent of x, y. For this model, the expected number of events that we can update in parallel for the first *cycle* is

$$\sum_{\substack{x=0,\ldots,w_c-1,\\ y=0,\ldots,h_c-1}} \mathbf{Ex}\left(I_{A_{x,y,1}}\right) = \sum_{\substack{x=0,\ldots,w_c-1,\\ y=0,\ldots,h_c-1}} \mathbf{Pr}\left(I_{A_{x,y,1}}\right) = \frac{n_c}{9}. \tag{5.18}$$

For the upcoming *cycles* ($j > 1$) probabilities change because there are no longer m events in each cell. I compute these probabilities by sampling the situation for different numbers of events m and cells n_c. Figure 5.6 illustrates the simulation results. I repeated the experiment 100 times and averaged over the results. The number of events processed in each *cycle* on average is independent of the number of events m and the number of cells n_c. For $n_c = 10 \times 10$, on average 8.3 events

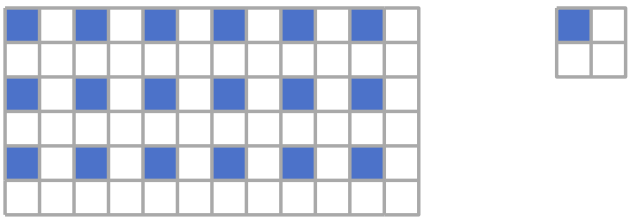
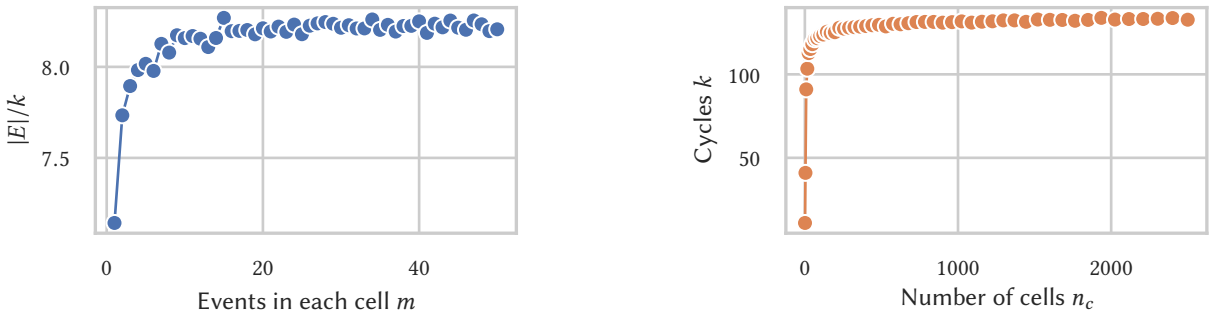


Figure 5.5: A grid tessellated (left) by following the optimal tessellated pattern where one out of four cells take part in an update *cycle*. Blue cells take part in the update, while white cells are excluded because of dependency restrictions.



(i) (ii)

Figure 5.6: Parallel processed events: the average number of events processed in parallel for any *cycle* stays between 8 and 9 for $w_c = h_c = \sqrt{n_c} = 10$ cells. For a constant number of events $m = 10$, and a changing number of cells $n_c = w_c \cdot h_c$, with $w_c = h_c$, the required number of *cycles* k stays below 150 (ii).

are processed for each *cycle*, that is, approximately 8.3 % of cells participate, compare Fig. 5.6i. For a fixed number of events ($m = 10$) and an increasing number of cells, Fig. 5.6ii reveals that the number of *cycles* k stays below 150, that is, approximately $m/k = 10/150 \approx 6.7\,\%$ cells are updated.

In summary, the overall parallel time complexity of ParallelEventDrivenUpdate is

$$O(\log^2(n_{\mathcal{A}}) + k \cdot (n_{\mathcal{A},c} + \log(n_c) + \log(|P|)), \tag{5.19}$$

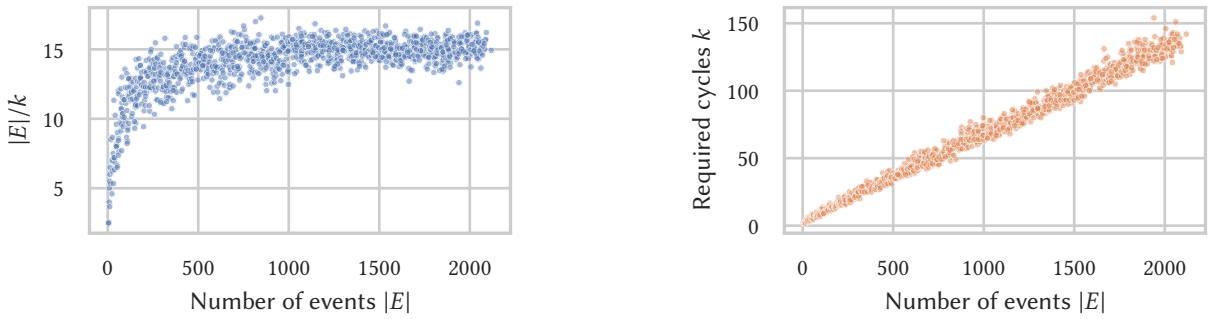
where k is the number of *cycles*, $n_{\mathcal{A}}$ the number of simulated agents, $n_{\mathcal{A},c}$ the maximal number of agents within a cell and P the discrete set of possible next positions. Since the percentage of cells that can be updated on average seems to be constant for my pessimistic model, k depends only on the maximum number of agents within a cell $n_{\mathcal{A},c}$, which is bounded by another constant (see Section 5.2.2). In theory, this leads to

$$O(\log^2(n_{\mathcal{A}}) + \log(n_c) + \log(|P|)). \tag{5.20}$$

However, the hidden constant can be large and depends on the agent utility definition of the optimal steps model.

5.3.2 Experimental observations

In the following, I conduct two crowd simulations to test the parallelism using the optimal steps model presented in [260]. The first benchmark scenario is a 10×100 square meter corridor. Every $\Delta t \cdot 2 = 0.8$ seconds, 5 new agents spawn at a random position. They walk from left to right until they get teleported back to the left. Therefore, I emulate a cyclic corridor. Over time, the corridor becomes more and more crowded. Agents are distributed uniformly over the complete domain. The resulting cell size is $\delta_c = 2.61$ meter. Consequently, the linked data structure consists of $n_c = 10 \times 100/(2.61 \times 2.61) = 146$ cells. As expected, the number of *cycles* k required linearly



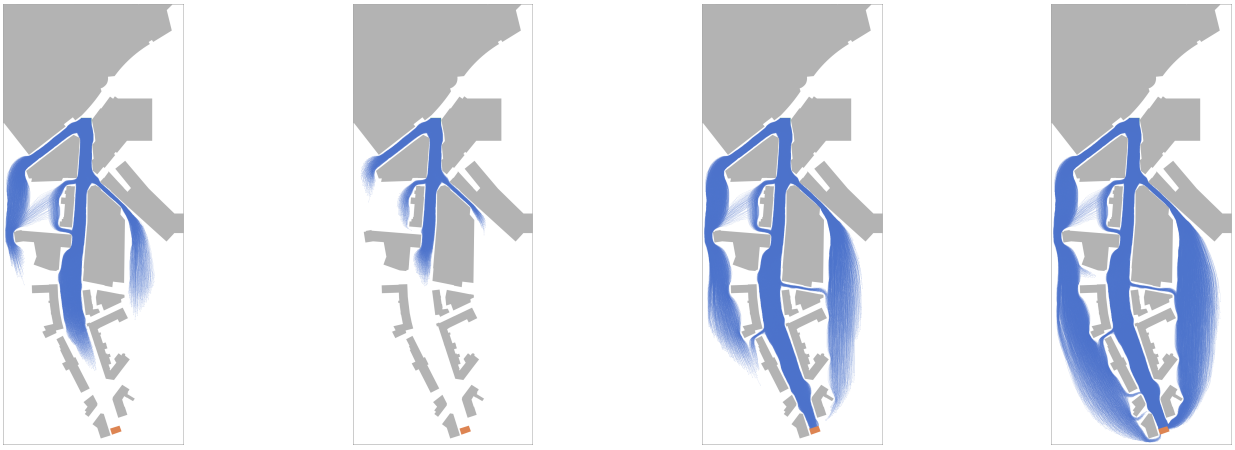
(i) (ii)

Figure 5.7: Parallel processed events (corridor): the number of *cycles* k required to process all events E increases linearly with the number of events $|E|$ occurred within Δt (ii). The average number of events processed for each *cycle* fluctuates between 12 and 17 (i). Therefore, on average between 8 % and 11, 6 % of cells can be updated in parallel in each *cycle*.

increases with the number of events $|E|$. The average number of events processed for each *cycle* fluctuates between 12 and 17. Importantly, there is no decreasing trend recognizable. On average more than 8 % of all cells and $100/1500 \cdot 100 = 6,67\,\%$ of all events E are updated in parallel. Furthermore, $1 \leq k \leq 150$. Consequently, for a proper hardware setup, a speedup of an order of magnitude for LOCOMOTION(Δt) can be expected.

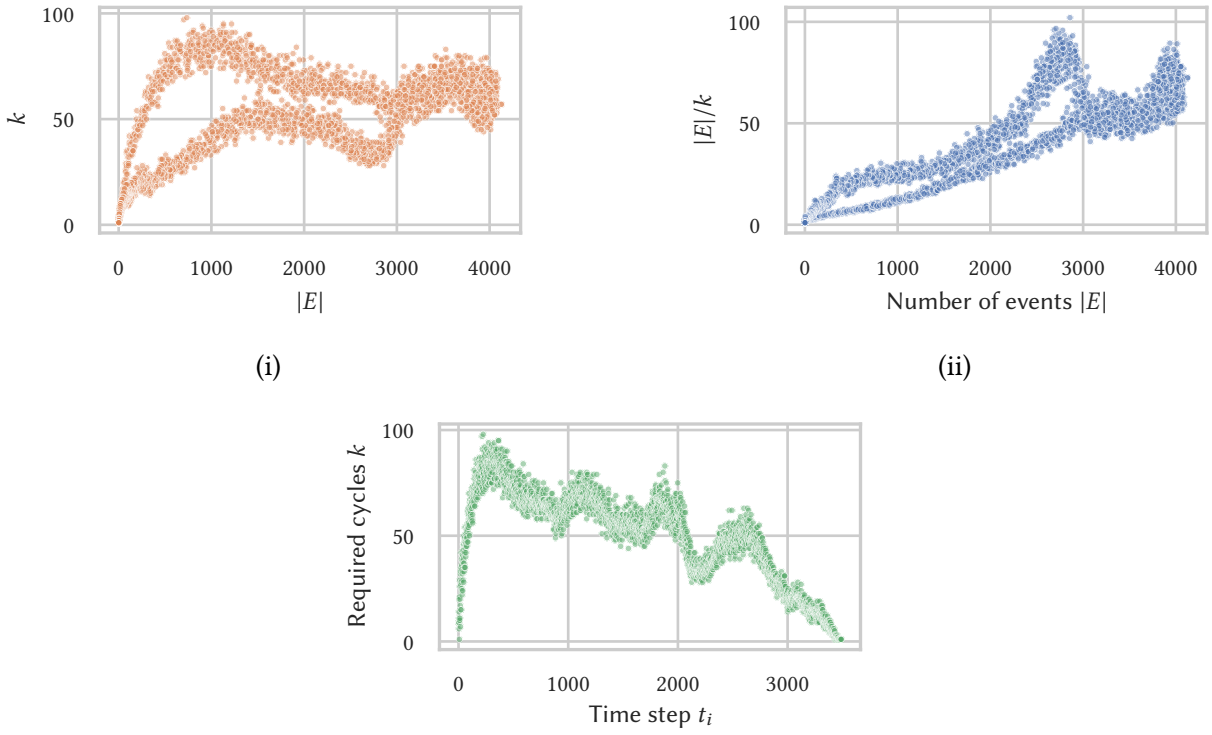
The second scenario is the Richard-Wagner-Straße in Kaiserslautern flooded with 10,000 agents walking from top to the bottom, compare Fig. 5.8. The domain is 228×560 square meters large. This scenario is inspired by demonstration marches that often take place in this part of Kaiserslautern. Every second, 20 agents spawn at random positions inside the green rectangle. Each agent walks towards the orange rectangle Γ at the bottom. The simulation reveals that the parallelism of PARALLELEVENTDRIVENUPDATE depends on agents' spatial distribution, including the number of non-empty cells. At the start of the simulation, agents only occupy a tiny region. Therefore, the number of required *cycles* k is large, compare Fig. 5.9. As expected, as soon as agents are more spread out more events can be processed in parallel. The portion of parallel processed events lies between 1 % and 3.3 % (if 10,000 agents are present). Therefore, $30 \leq k \leq 100$ follows, compare Fig. 5.9ii. Consequently, for a proper hardware setup, a speedup of an order of magnitude for LOCOMOTION(Δt) can be expected.

Figures 5.7i and 5.9ii illustrate the number of events that can be processed averaging over the number of required *cycles* k. To estimate the number of threads suitable for the problem, I compare the number of events that can be processed in each *cycle*. Instead of averaging over the cycles of the time step, I average over the time steps with respect to each *cycle* number. Figure 5.10 shows the absolute and cumulative portion of events processed in a *cycle*. For both scenarios, parallelism decreases with the *cycle* number $j = 1, \ldots, k$. If agents are distributed evenly, the only reason for this effect is that the number of remaining unprocessed events decreases with each *cycle*. Otherwise the slope in Fig. 5.10iii would be constant. For the Richard-Wagner-Straße scenario 50 % of events are processed after approximately 10 *cycles*. For the corridor scenario, it takes ruffly 43 *cycles*. The difference reflects the differences in spatial agent distributions. The



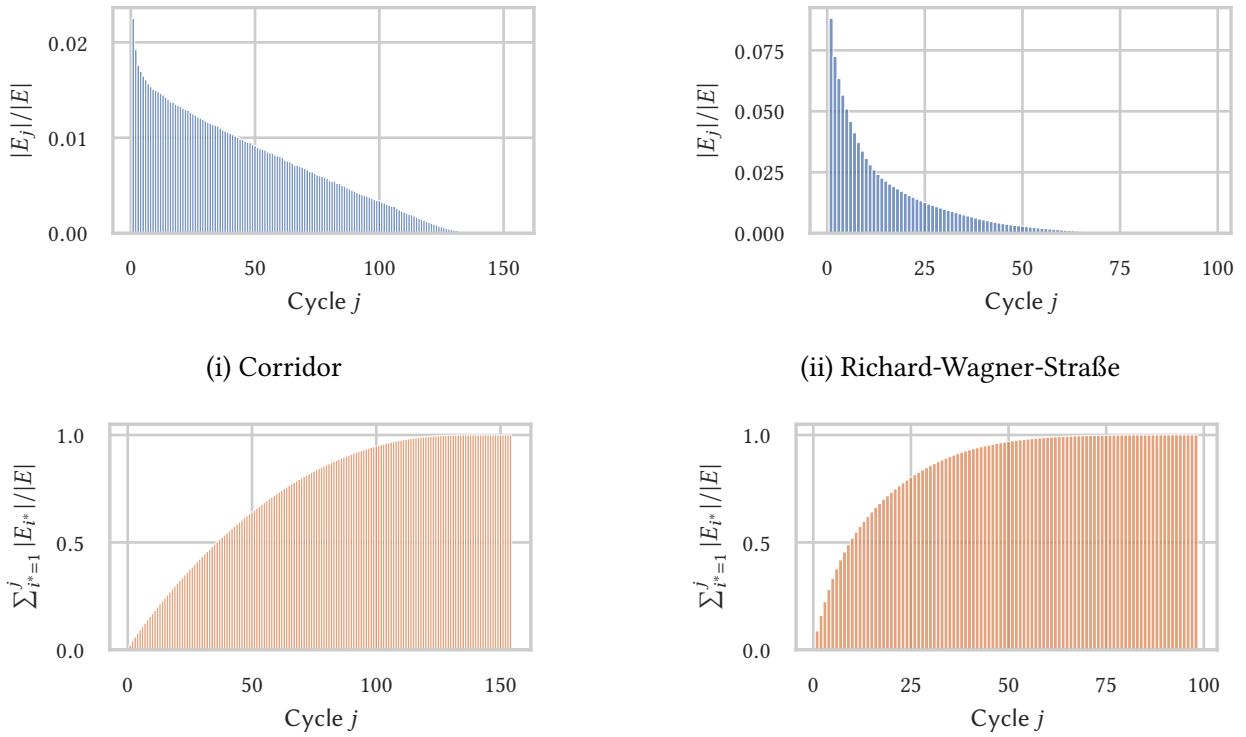
(i) (ii) (iii) (iv)

Figure 5.8: Snapshots of agent trajectories 125 s (i), 196 s (ii), 280 s (iii), and 363 s (iv) into the simulation of the Richard-Wagner-Straße scenario using a dynamic navigation field.



(i) (ii) (iii)

Figure 5.9: Parallel processed events (Richard-Wagner-Straße): the number of *cycles* k required to process all events fluctuates but stays below 100 (i). The same is true for the number of events processed on average in each *cycle* (ii). For any *cycle*, at least 1 % of events can be processed in parallel. At the start of the simulation parallelism decreases until approximately time step $t_i = 500$. Then it increases since agents cover more and more cells (iii).



(i) Corridor

(ii) Richard-Wagner-Straße

(iii) Corridor

(iv) Richard-Wagner-Straße

Figure 5.10: Average portion of events executed in parallel in *cycle* $j = 1, \ldots, k$ for the corridor (i) and Richard-Wagner-Straße (ii) and the cumulative portion below. I average over all time steps of the simulation.

corridor is evenly populated, while for the Richard-Wagner-Straße we have crowded and low-density areas. Early *cycles* update all agents in those low-density areas while late *cycles* deal with events occurring in crowded areas. This explains the different slopes of the cumulative number of processed events illustrated in Fig. 5.10. If we would use $0.025 \cdot 4000 = 100$ threads for the Richard-Wagner-Straße simulation and each thread would be responsible for the computation of only one event (in each *cycle*), then after on average the 12^{th} *cycle* some threads would idle and after the 25^{th} *cycle* ruffly 50 % of threads would idle, compare Fig. 5.9ii. Therefore, it is important that more than |**E**| threads contribute work in ParallelBruteForceOptimization (Line 9 in Algorithm 6). More precisely, the computation of one *footstep event* must be parallelized as well. Otherwise, we can not utilize computing resources properly.

5.4 GPGPU implementations

In contrast to CPUs, the hardware architecture of graphics processing units (GPU) is designed for massive parallelism. Since GPUs are part of many current and upcoming supercomputers, efficient exploitation of GPUs has become essential in scientific computing. Additionally, GPUs offer thousands of cores inside affordable off-the-shelf workstations making general-purpose graphics

processing units (GPGPUs) a source of cheap and efficient computational power. Consequently, I consider how to exploit GPUs for large-scale crowd simulations. I use a GPU to demonstrate acceleration achieved by the introduced parallelism experimentally.

I base my implementations on OpenCL to support a broad range of hardware accelerators. I integrated it into our Java-based open-source framework Vadere [157]. To call the OpenCL kernels within Java, I use the Lightweight Java Game Library 3.2.3 [195]. One important factor to successfully transform an algorithm into a GPU implementation is to use the fast local memory.

Since I want to show the performance effect of the dependencies enforced by the event-driven update, I also developed an OpenCL implementation of the parallel update scheme. After all, the parallel update scheme (ParallelUpdate) might still be a valid alternative for many scenarios. I use the measured run time of ParallelUpdate as a baseline for ParallelEventDrivenUpdate – we can not expect it to be faster.

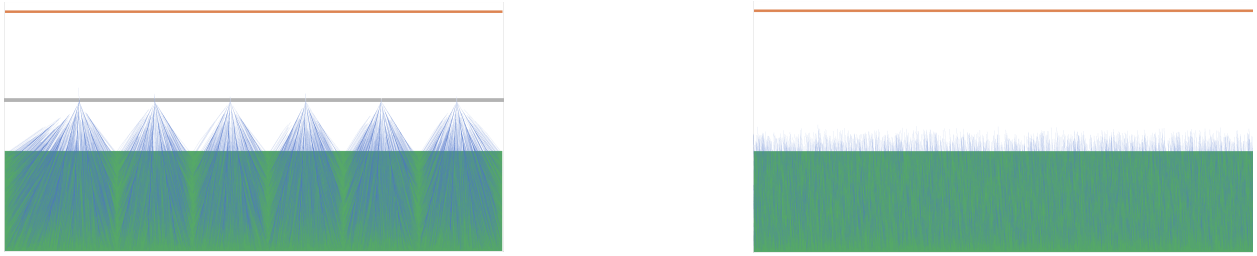
5.4.1 Navigation field transfer

In my implementation, navigation fields are computed by the CPU and I copy them onto the GPU. If the navigation field is dynamic, I would have to copy a lot of data after each Δt simulated seconds. This leads to a significant performance drop. Enabling simulations based on dynamic navigation fields carried out by the GPU requires future work. In Section 9.3, I will show that multiple GPU-based eikonal solvers exist. Integrating them or transforming my solver (Section 9.4.4) into a GPU-version goes beyond this thesis. However, to show achievable performance improvements for Locomotion(Δt) (Algorithm 1) in practice it is reasonable to ignore navigation field computation.

To compute target and obstacle utilities on the GPU, solutions of the eikonal equations Φ_{Γ_i} and the distance function d_Ω are required. d_Ω is approximated by another solution of an eikonal equation, that is, $\Phi_{\partial\Omega}$, see Section 8.7. Each eikonal equation is solved on the host device (CPU). Before the simulation starts, I sample the solutions on regular grids and transfer each from the host to the *global memory* of the GPU. Values in between grid points are bi-linearly interpolated. Furthermore, an approximation of the possible next footstep positions P (defined by Eq. (5.12)) is computed and transferred to the GPU. Therefore, at the start of the simulation, one massive memory transfer is necessary.

5.4.2 The parallel update scheme

Implementing the parallel update scheme (Algorithm 4) for the GPU is straightforward. OpenCL kernel methods realize Seek and Move. Agent properties such as the start and end position of its next position are saved in structures of arrays (SoA). To test for conflicts, I use the linked data structure described in Section 5.2.1. Therefore, before updating the agents' position for one time step, I construct the linked cell data structure beforehand. Instead of threads and thread groups, the GPU uses OpenCL work-items and work-groups. After the linked cell date structure is constructed, Seek computes the agents' next possible positions in parallel. Since there is enough work to do, each agent is assigned to a different work-item (thread) executing the Seek kernel. Thereby I increase the amount of work for each work-item compared to the event-driven update implementation. If the agents' next footstep happens within $[t; t + \Delta t)$, the next possible



(i) Bottleneck scneario.

(ii) Open space scenario.

Figure 5.11: Snapshots of benchmark scenarios: all agents are uniformly distributed inside the green 400×1900 square meter rectangle at $t = 0$ seconds. They walk towards their orange destination Γ at the top. The blue trajectories reveal the agents' movement through the 6 bottlenecks (i). They move towards each other. Without bottlenecks, agents stay away from each other during the whole simulation (ii).

best position is computed. Work-items reduces all possible positions to the best one by solving Eq. (2.12). Finally, the resulting position is saved to *global memory*. For each agent, the MOVE kernel is executed by a different work-item (thread). This kernel tests if there are any collisions for the possible next positions (calculated by the SEEK kernel) agents within the Moore neighborhood of the linked cell data structure. If there is none, I update the agent's event time and position accordingly. Otherwise, I mark the cycle as conflicted. SEEK and MOVE calls are repeated until there is no collision detected and all event times are greater than $t + \Delta t$. For efficiency reasons, I choose $|P| = 2^z$ discretization points for step disc, where $z \in \mathbb{N}$, because the number of processing units in GPU-groups is equalt to 2^y for some $y \in \mathbb{N}$.

5.4.3 The event-driven update scheme

Since I developed PARALLELEVENTDRIVENUPDATE for SIMD hardware an OpenCL implementation is obtained by transforming each procedure in Algorithm 6 into a kernel function. Because at most $n_c/4$ agents can be updated in parallel (see Section 5.3) and because parallelism decreases with each cycle and depends on the spatial distribution of agents, I use multiple work-items for computing the next position of a specific agent. Whenever some task is assigned to a thread group (an OpenCL work-group), intermediate results are stored in fast *local memory*. For example, all utility evaluations of one specific agent or the event times required for the CELLFILTER kernel. Prefix sums and the bitonic sort algorithm also use fast *local memory* whenever possible.

5.4.4 Comparison of Computation Times

To compare computation times, I carry out a series of tests. The analysis above showed that PARLLELEVENTDRIVENUPDATE performs best for evenly distributed and well-separated agents because, in this case, footstep events are likely to be independent of each other. It performs worst if cells are either empty or highly populated. To control the spatial distribution of agents, I use two artificial benchmark scenarios.

The first multi-bottleneck scenario consists of 6 bottlenecks. The domain size is 1000×2000

Number of agents $\lvert\mathcal{A}\rvert$	ParallelUpdate			ParallelEvent-DrivenUpdate		EventDrivenUpdate
	OpenCL (GPU)	OpenCL (CPU	Java (CPU)	OpenCL (GPU)	OpenCL (CPU)	Java (CPU)
$10 \cdot 10^3$	3	15	80	20	99	140
$100 \cdot 10^3$	13	119	1,190	60	546	2,100
$500 \cdot 10^3$	74	1,348	12,137	160	2,875	30,096

Table 5.1: Average computation time in milliseconds of $\Delta t = 0.4$ seconds simulation time of the open-space scenario for $10 \cdot 10^3$, $100 \cdot 10^3$ and $500 \cdot 10^3$ agents.

square meters. Agents spawn at $t = 0$ seconds at random positions in a large 400×1900 squaremeter area. They all pass through the 6 bottlenecks. Even though the multi-bottleneck scenario is simple, it imitates more complex geometries and situations by generating a wide range of densities. The simulation starts with low densities until congestions begin to occur - the density increases because all agents have to move to 6 specific spots. The second open-space scenario is equal to the first one but without bottlenecks. Both scenarios are depicted in Fig. 5.11.

For all tests $|P|$ is approximated by 32 points and Δt is set to 0.4 seconds. My OpenCL implementation uses single precision and the existing (single-threaded) Java implementation double precision. Tests were carried out on the following hardware platform: Intel i5-7400 Quad-Core (3.50 GHz), 8 GB DDR4 SDRAM, and a graphics card NVIDIA GeForce GTX 1050 Ti / 4 GB GDDR5 VRAM.

In open space, using GPGPU computation over the existing Java implementation speeds up the simulation by multiple orders of magnitude – the simulation runs more than 100 times faster. However, this comparison is not very meaningful since I compare run times based on implementations realized by different programming languages and executed on different machines. To eliminate at least one factor, I compare the OpenCL implementation using the CPU and the GPU. Running the same OpenCL code on the CPU is 5 up to 18 times slower than on the GPU.

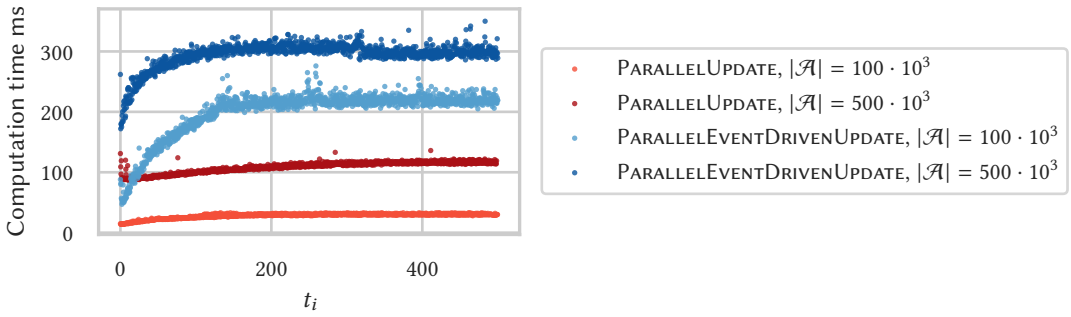


Figure 5.12: Comparison of computation times over a simulation run of the multi-bottleneck scenario for 100,000 and 500,000 agents $\mathcal{A}$ using ParallelUpdate and ParallelEventDrivenUpdate. The computation time is required to simulate $\Delta t = 0.4$ seconds.

Furthermore, the GPU scales better for a growing number of agents, compare table 5.1. During the simulation, the computation times do not significantly fluctuate.

The multi-bottleneck scenario benchmark reveals that computation times fluctuate during the simulation run if the ParallelEventDrivenUpdate is used. As expected, the computation slows down when agents approach the bottlenecks and thus move closer together. After approximately 100 simulated seconds, the computation time reaches a plateau. Figure 5.12 illustrated this phenomenon.

5.5 Source code

The source code realizing the OpenCL implementation of ParallelEventDrivenUpdate and ParallelUpdate is part of the open source simulation framework Vadere [294, 157]. Together the following files belong to the implementation of ParallelEventDrivenUpdate:

- `CLAbstractOSM.java`,
- `ICLOptimalStepsModel.java`,
- `CLParallelEventDrivenOSM.java`,
- `CLUpdateSchemeEventDriven.java`, and
- `ParallelEventDrivenOSM.cl`.

And

- `CLAbstractOSM.java`,
- `ICLOptimalStepsModel.java`,
- `CLUpdateSchemeParallel.java`,
- `CLParallelOSM.java`, and
- `ParallelOSM.cl`.

belong to the implementation of ParallelUpdate.

5.6 Summary

In this chapter, I introduced parallelism to optimal steps models. In Section 5.1, I discussed the importance of preserving the intended update order of models. By changing the order, one creates a new model. In the modeling process, we should not limit ourselves by computational restrictions. Instead, proper algorithms should free us from those restrictions.

Therefore, I presented ParallelEventDrivenUpdate in Section 5.2. It introduces parallelism without changing the model. I gave a detailed and technical description that focuses on single-instruction multiple data architectures such that other researchers and developers can implement ParallelEventDrivenUpdate for their hardware setup. Extensions to support distributed memory systems or to make use of other (hardware) accelerators belongs to future works.

Because the event-driven update scheme imposes event dependencies that reduce parallelism, I gave a theoretical and experimental analysis of the parallelism achieved by ParallelEventDrivenUpdate in Section 5.3. The average portion of events that can be processed in parallel is relatively small if cells are either empty or highly populated. However, if we simulate thousands of agents, even a small portion is enough to utilize multiple processors efficiently. The reason for that is that finding independent events is not time-consuming if there are enough processors available.

In the last part of this chapter (Section 5.4), I compared execution times for a GPU. To get a sense of how much performance we lose by the imposed restrictions, I use the ParallelUpdate as a baseline. ParallelUpdate outperformed ParallelEventDrivenUpdate but in case of half a million agents it was only 3 times faster. I was able to simulate up to half a million agents in real-time without changing the model(s). I also compared the parallel execution of the CPU and GPU. Simulations on the GPU ran up to 18 times faster. The single-threaded Java implementation could not keep up at all.

The described parallel linked cell technique can be carried over to other models if the agents' influence remains local, a valid property for many models. It opens the door for microscopic pedestrian simulation on the GPU. And as I showed, GPGPU can be beneficial for pedestrian dynamics beyond CA models or ODE-based models.

PART III

Large-scale navigation fields

Introduction

Navigation fields realize robust wayfinding to facilitate simulations with complex and large geometries, compare Chapter 3. Static navigation fields do not change during a simulation run. They are computed in the Initialization of Algorithm 1 before the simulation loop starts. Dynamic navigation fields have to be recomputed frequently. This computation is part of StrategicAndTactic. In the past, frequent computation of the eikonal equation's solution imposed high computational costs that made real-time simulations for large-scale scenarios impossible.

For general spatial domains $\Omega \subset \mathbb{R}^2$, destinations $\Gamma \subset \Omega$ and travel speed functions f the eikonal equation Eq. (3.2) is solved by numerical methods such as the FastMarchingMethod. The development of new numerical methods was driven by many authors outside of the pedestrian dynamics community. Over the decade, they designed efficient and parallel methods to achieve accurate results while keeping the computation time at a minimum. I give an extensive review in Section 9.3. For each method, a particular space discretization $\mathcal{T}$ is part of its input, and they compute an approximation of the travel time $\Phi_\Gamma(\mathbf{v})$ for a finite number of vertices $\mathbf{v}$. A typical discretization, also used in pedestrian dynamics, is the *(structured) Cartesian grid*. In that case, elements of $\mathcal{T}$ are unit squares. Each grid point has at most four neighbors at a distance of h meters. Using small elements, *Cartesian grids* are a save option to achieve accurate approximations. However, they consist of a large number of vertices. And since the execution time of numerical methods depends on the number of grid points, *Cartesian grids* lead to a heavy workload.

I identify two strategies to reduce the computation time required to compute navigation fields: the first one is to design a novel method that outperforms state-of-the-art solvers – a rather ambitious project, because there already exists efficient solvers. The second way to combat the

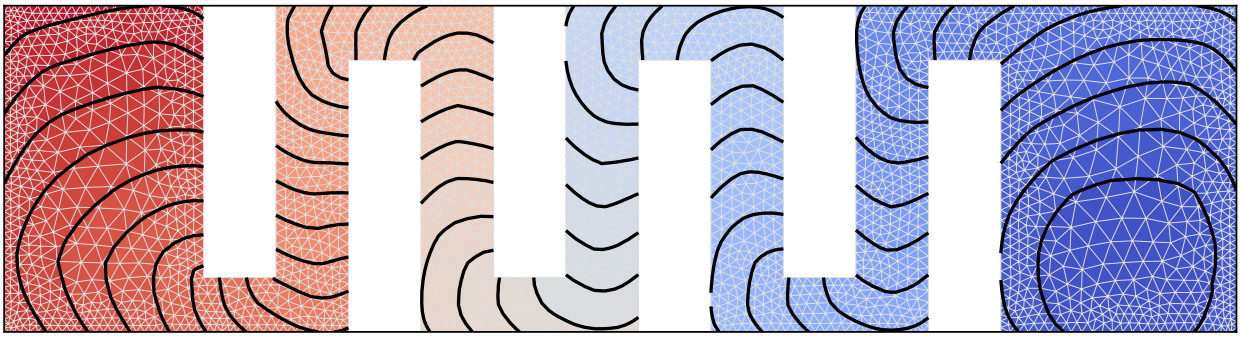


Figure 5.13: An example of a navigation field where the destination Γ is on the right.

problem is to reduce the number of discretization points without losing too much accuracy.

In Part III, I follow both strategies. In Section 9.5, I design a new eikonal solver (for unstructured meshes and Cartesian grids) to compute dynamic navigation fields. My approach exploits the fact that the dynamic navigation field does not change too much over time. Therefore, the method outperforms other methods if multiple 'similar' eikonal equations had to be solved.

Secondly, I change the discretization from *Cartesian grids* to *unstructured meshes*. I develop EikMesh (Chapter 8) an adaptation of the meshing algorithm DistMesh (Chapter 7) to generate high-quality adaptive unstructured triangular meshes for pedestrian dynamics. Meshes generated by EikMesh conform to the geometry and adapt to localized resolutions – important properties for accuracy (of simulation results) and the reduction of discretization points to improve performance.

In Chapter 6, I introduce mesh generation for pedestrian dynamics. I discuss the state-of-art-techniques and algorithms with a focus on two-dimensional unstructured meshes. Furthermore, I establish a metric to evaluate the mesh quality and discuss point location algorithms that are important for using unstructured meshes in a pedestrian simulation. In Chapter 7, I give a description of DistMesh that is important to understand the EikMesh algorithm. EikMesh's description follows in Chapter 8. Finally, in Chapter 9, I discuss the computation of navigation fields for unstructured meshes. I start by an algorithmic description of numerical methods (Sections 9.1 to 9.3). The consecutive section, Section 9.4, focuses on the adaptive aspect of unstructured meshes – the section belongs to the second way. In the subsequent section, I introduce the InformedFastIterativeMethod – a novel eikonal solver specialized for dynamic navigation field computation.

From the Delaunay triangulation, walks on triangulations, Ruppert's algorithm to DistMesh, a lot of work from other authors in the field of computational geometry went into EikMesh. To facilitate a deep understanding, I describe the required algorithms in detail. Therefore, the reader familiar with unstructured triangular mesh generation might want to skip Chapters 6 to 8.

CHAPTER 6

Mesh generation for pedestrian dynamics

> *"The only way to deal with an unfree world is to become so absolutely free that your very existence is an act of rebellion."*
>
> – Albert Camus

Mesh generation and (adaptive) refinement have their application in many different areas ranging from computer graphics, animation, and complex numerical simulations of physical phenomena, such as fluid dynamics to robotics and, of course, pedestrian dynamics. In each area, many computational techniques are based on a discrete representation of the underlying geometry. Objects and domains of interest are tessellated into a mesh of simple elements. Quoting Joe F. Thompson:

> *"An essential element of the numerical solution of partial differential equations (PDEs) on general regions is the construction of a grid (mesh) on which to represent the equations in fine form. At present it can take orders of magnitude more man-hours to construct the grid than it does to perform and analyze the PDE solution on the grid. This is essentially true now that PDE codes of wide applicability are becoming available, and grid generation has been cited repeatedly as being a major pacing item. The PDE codes now available typically require much less esoteric expertise of the knowledgeable user than do the grid generation codes."* – Thompson in 1992

Two decades later, meshes are still a recurring bottleneck [275].

Everything started in the 70's decade when researchers and engineers were interested in the application of the finite-element method. Rapidly more and more researchers became involved, among other computer scientists. Around 1975, they established a new discipline called computational geometry, which was and still is interested in the computational complexity of geometric problems. Research in mesh generation is concerned with developing efficient, robust, and automatic algorithms to construct, adapt, and maintain high-quality meshes for complex geometries and objects. Meshes are expected to conform to several, sometimes contradictory, requirements. They must accurately represent the geometry, that is, adhere to complex geometrical features, and support a high spatial resolution in areas of interest while maximizing sparsity elsewhere to reduce computational costs. Additionally, meshes should consist of high-quality elements,

promote local, adaptive, and incremental refinement techniques, and they should be efficiently computable.

While applications such as pedestrian dynamics use meshes as an ingredient, computational geometry is concerned with mesh generation itself, especially the amount of time and space it takes to generate a mesh and the properties it has. These properties range from lower and upper bounds of the largest θ_∞ and smallest θ_0 angle to quality bounds based on a particular measurement ρ. I establish quality measures for this task in Section 6.6.

In this chapter, I lay the foundation for EikMesh, a new mesh generator for unstructured two-dimensional meshes. Its detailed description will follow in Chapter 8. The discourse starts with mesh requirements for microscopic navigation field-based pedestrian simulation. I introduce and discuss different mesh types and mesh generation methods. Furthermore, I give a more detailed description of all algorithms and data structured EikMesh is based on, including the DistMesh algorithm.

6.1 Requirements for pedestrian dynamics

Even if pedestrians move in a three-dimensional space, pedestrian simulators model their spatial domain as a two dimensional abstraction of the real world, ignoring the z-coordinate, and with it the height of objects. This simplification disregards obstacles, which are barely passable like fences, desks, chairs and architectural features such as stairs, escalators and elevators. Models, which include the influence of such 'soft obstacles', avoid to model their three-dimensional body. For example, optimal steps models decrease the attractiveness of areas close to obstacles [257].

Undoubtedly, the motion and path planning of pedestrians is fundamentally influenced by the given architectural environment. Furthermore, small changes in the geometry can have a great impact on the behavior of pedestrians. For example, a passage just a little too narrow might only be passable by one pedestrian at a time. A wider passage might double the flow of pedestrians. Realistic models reproduce these effects. In Fig. 6.1 we can observe the effect of small geometrical changes. I deduce that the first important core mesh requirement is an accurate representation of the given two-dimensional geometry.

Pedestrians do not only navigate around obstacles but keep a certain distance to them, compare the experiment conducted in [260]. Modelers introduce this natural obstacle avoidance by different techniques that depend on the distance $d_{\mathcal{W}}$ to obstacles $\mathcal{W}$. In case of force-based models [118, 48, 49], a force pointing away from obstacles has a repulsive effect. Optimal steps models [257, 299, 300, 305] achieve a similar effect through a reduced utility for positions close

(i) (ii)

Figure 6.1: Effect of small geometrical changes: for a 10 cm wider corridor (ii) the flow of a simulation increases significantly. For both simulations I used the Gradient Navigation Model [68] and, except for the corridor width, the exact same configuration.

to obstacles. From an implementation perspective, the computation of $d_{\mathcal{W}}$ can be expensive. In Section 8.7 I propose to use a so called background mesh to save computation time. Before the simulation starts I construct a coarse unstructured background mesh and compute $d_{\mathcal{W}}$ for each vertex. During the simulation the distance $d_{\mathcal{W}}(\mathbf{x})$ is approximated by interpolating based on the vertices and values of a triangle τ containing $\mathbf{x}$. The approximation has to be accurate, especially for small values of $d_{\mathcal{W}}$ because important phenomena such as congestion occur at bottlenecks, that is, positions close to obstacles. In Section 9.4.4 I explain why the underlying mesh of navigation fields should be fine at these areas. Intuitively, at these bottlenecks most important parts of the dynamics take place.

In this work, I focus on pedestrian simulation models for which the cognitive map of a person is represented by a navigation field – the solution of one (static) or multiple (dynamic) eikonal equations. In section Section 9.4, I discuss different solvers and their extensions. For all these solvers, the computation time depends on the number of mesh points. Using a Cartesian grid for space discretization causes either large errors or long computation times. For large spatial domains the computation time becomes prohibitive, especially for dynamic navigation fields for which frequent re-computations are necessary. This is one of the big computational bottlenecks we have to overcome to enable real-time large-scale simulations. To do so, I propose to reduce the number of mesh elements drastically. For this reason I designed EikMesh in order to generate conforming meshes with localized resolutions.

I conclude that meshes that represent the underlying geometry in large-scale navigation field-based pedestrian simulation have to

(1) adhere to complex geometrical features,

(2) support high spatial resolution where needed, notably near obstacles, and

(3) have a low spatial resolution everywhere else.

6.2 Mesh types

In the following section I give an introduction to meshing, the different mesh types, and what property they impose. Most definitions and terms are listed in [275, Chapter 1]. A *mesh* (grid) is a collection of simple geometrical objects that tessellate the spatial domain and can be represented by a planar undirected graph. Based on their topology, meshes can be classified in several ways. Like many meshing algorithms, EikMesh constructs meshes that are

(i) *unstructured*,

(ii) *conforming*,

(iii) *homogeneous* and

(iv) *triangular*.

Properties can be arbitrarily combined, but some combinations make more sense than others. For example, *unstructured* meshes (Section 6.2.1) are often *triangular*. *Structured* meshes that do not adhere to the domain boundary are mostly *conforming* (Section 6.2.2), whereas *rectangular structured* meshes which align with the boundary are not.

6.2.1 Regularity

First of all, a mesh is either *structured* or *unstructured*. These terms refer to the regularity of the mesh. *Structured* means highly regular. The topology of *structured* meshes is usually given implicitly. For example, accessing a triangle containing a given point $\mathbf{x}$ is trivial for a 2-d structured triangular mesh. It requires only a few simple arithmetic operations. One of the most regular 2-d *structured* mesh is the so-called *structured rectangular grid*. I use the term *grid* instead of *mesh* to emphasize its regularity. If the *structured rectangular grid* consists of squares, I call it a *Cartesian grid*. For a Cartesian grid with side length h and its origin at $\mathbf{0}$

$$(i, j) = (\lfloor x/h \rfloor, \lfloor y/h \rfloor) \tag{6.1}$$

is the column and row of the cell containing $\mathbf{x} = (x, y)$. Consequently, a simple array suffices to store data for each vertex of the Cartesian grid.

Unstructured meshes, on the other hand, require explicit storage of their topology information. Therefore, a more sophisticated and memory-hungry data structure (see Section 6.8) is required to represent the mesh. Furthermore, *point locations* (Section 6.7) are non-trivial and require more computational effort – point location is the problem of finding the (one) triangle τ containing some given point $\mathbf{x}$. However, unstructured meshes are much more versatile because of their ability to combine good element shapes with odd domain shapes and element sizes that grade from very small to very large [275, p. 3].

Comparing *unstructured* and *structured* meshes in general, is difficult and goes beyond my work. However, we can identify some trends. For an equal mesh size, *structured* meshes require only a fraction of the memory of their counterparts [27, p. 300]. As showcased by the example above, their regularity enables fast index-like access to arbitrary mesh elements. Consecutive accesses to neighboring elements can often be implemented in a cache-friendly manner. Additionally, global refinement and coarsening can be easily implemented. However, the regularity requirement comes at a high price: dealing with complex spatial domains is difficult, because vertices do not necessarily align with the boundary or any other geometrical constraints. Furthermore, many local refinement and coarsening techniques introduce *irregularity*. Therefore, *structured* grids are suitable primarily for domains that have tractable geometries and do not require a strongly graded mesh [275, p. 3]. Despite of a computational more expensive *point location*, *unstructured* meshes offer more geometrical flexibility. This leads to fewer elements and the additional computational and memory costs per element may still result in overall computational savings.

An attempt to get the best from both types is to build so-called *hybrid* or *block-structured* meshes. As their name suggests, meshing algorithms of that type build an *unstructured* mesh and refine its faces in a structural manner. The generated meshes are the result of merging multiple *structured* grids in an *unstructured* way together. Unfortunately, their generation is rarely fully automatic [27].

6.2.2 Conformity

EikMesh constructs *conforming* meshes. Hereby I am not referring to the adherence to geometry but to the conformity with respect to the mesh topology. Adjacent elements of *conforming* meshes must intersect along a common edge or face – there are no hanging nodes or overlapping faces allowed. This lack of fractional element connectivities is desirable because most efficient numerical methods, including many solvers for the eikonal equation (Section 9.2), rely on it. Hanging nodes introduce special cases handled by special code, often leading to a performance drop. *Non-conforming* meshes allow hanging nodes and gain additional geometrical flexibility, but they exhibit edges and faces, that do not match perfectly between neighboring elements [189].

6.2.3 Element type

Structured grids are mostly *homogeneous* – they consist of a single element type. The most common types are *triangles*, *quadrilaterals* for 2-d domains and *tetrahedra*, *hexahedrons* for 3-d domains. *Triangles* and *tetrahedra* are the simplest of all ordinary convex polygons and polyhedra, respectively. *Quadrilaterals* and *hexahedrons* are typically used for *structural* meshing, while the simplest objects are used for *unstructured* meshing.

6.3 Meshing algorithms

There exists a vast range of meshing algorithms, and some are a composition or conglomerate of others. For example, a coarse background mesh is a common construct that supports meshing algorithms. Some of them use the constructed background mesh as an initial mesh that is refined and improved.

This section organizes meshing algorithms into the four categories under which all modern meshing algorithms for unstructured meshes fall: *grid-overlay methods*, *advancing-front methods*, *Delaunay-based algorithms* and *mesh-improvement approaches*. An excellent source for many aspects of mesh generation not covered in my work, for example generation of *structured* grids, is the *Handbook of Grid Generation* by Thompson et al. [286] and *Grid Generation* by Löhner [189].

(i) (ii)

Figure 6.2: Depiction of a conforming (i) and non-conforming (ii) triangular mesh: red vertices indicate non-conformity.

6.3.1 Grid-overlay techniques

Grid-overlay techniques are based on *structured* grids, but they introduce irregularity and sometimes non-conformity to adhere to the geometrical features. Some meshing algorithms use a *Cartesian grid*, but more sophisticated methods use *quadtrees* [28, 194] in 2-d and *octrees* [198, 177, 133] in 3-d. An axis-aligned square covers the complete boundary Ω. Each square is split recursively into four new squares until some element size condition is satisfied. After the *quadtree* is constructed, squares are warped and cut to conform to the boundary. Finally, a triangulation based on the *quadtree* is constructed.

The strength of *grid-overlay techniques* is their simplicity and efficiency. On the downside, their close relation to *structured grids* translates to high regularity. This lack of geometrical flexibility causes problems near geometrical constraints. Elements in those areas are often poorly shaped. Combating the problem by refinement schemes often leads to *over-refinement* [186]. Furthermore, mesh edges tend to align in a few preferred directions, which may influence results of numerical computations [275, p. 9].

Interestingly, the first algorithm, which achieved some quality guarantees, was a quadtree grid-overlay technique introduced by [28]. Excluding angles defined solely by the domain boundary, the algorithm designed by Bern et al. guarantees

$$18.4^\circ \leq \theta_0 \ \wedge \ \theta_\infty \leq 153.2^\circ \tag{6.2}$$

for a connected planar region bounded by a union of disjoint polygons [28].

6.3.2 Advancing-front methods

The *advancing-front method* starts from the domain boundary. It initializes a front that marches towards the interior. One-by-one, new elements are introduced whenever the front moves forward. This process terminates when the region is filled. A more detailed description is given by [286, Chapter 17]. New elements are created by carefully positioning new vertices that are adjacent to the triangles in the front. The front is updated accordingly.

Especially in 2-d, the method generates high-quality meshes. Since the front starts at the boundary, the resulting mesh adheres to complex geometrical features. The freedom of choice at the start of the march leads to high-quality elements close to the boundary. However, this freedom and with it the element quality drops at regions where advancing-fronts merge. Despite its practical success, conventional advancing-front methods are heuristic in nature, therefore, no meaningful guarantees have been proven yet [275]. For arbitrary inputs, it is possible that a valid tessellation can not be constructed at some iteration.

Early methods for two- and three-dimensional domains were introduced by [91, Chapter 2] and [191], and extended to surfaces and volumetric problems by [30, 248, 47, 132]. Schreiner et al. [250] developed a re-meshing method for surfaces based on the advancing-front paradigm, which can also be used to re-mesh a portion of the spatial domain.

6.3.3 Delaunay-based approaches

Delaunay-based approaches construct the topology utilizing the *Delaunay-criterion*. The criterion guarantees that no vertex is contained in the circumcircle (2-d) or circumsphere (3-d) of any

triangle (2-d) or tetrahedra (3-d) of the mesh. It is also called empty-circle (2-d) or empty-sphere property (3-d). The Delaunay triangulation [60] was named after Boris Delaunay, who introduced it around 1930. It is the dual graph of the Voronoi Diagram and can also be defined as the set of lines joining a set of points together such that each point is joined to its nearest neighbors. In two-dimensions, a Delaunay triangulation is unique if and only if there are exactly three points on any circumcircle.

There are many different algorithms to construct the Delaunay triangulation directly, ranging from incremental methods [181, 182, 35, 311, 108] to divide and conquer algorithms [264, 183] and computations of the convex hull [276, 41, 253, 136, 254, 37]. Another possibility is to compute the triangulation indirectly by its dual, that is, the Voronoi diagram [85], which is computed beforehand. With respect to mesh generation for which new vertices are introduced one-by-one or moved around, incremental methods give the necessary flexibility.

Delaunay-based approaches [191, 231, 242, 42, 221, 43, 44] consist of two tasks: the first one addresses the placement of vertices, and the second creates the mesh topology defined by the *Delaunay triangulation*. In general, a sequence is established in which these tasks are carried out. The first possibility is to create all vertices before computing the *Delaunay triangulation* in one pass. More prominent incremental methods, such as [45, 242, 241], first compute the Delaunay triangulation for boundary vertices. New internal vertices are inserted incrementally while maintaining the Delaunay criterion for all elements. To enforce geometric constraints, the *Delaunay triangulation* is replaced by the *constrained* or *conforming Delaunay* triangulation. EikMesh is based on the *(constrained) Delaunay triangulation* and exploits its properties. Therefore, I discuss it in Section 6.5 in more detail. The *Delaunay triangulation* has some theoretical properties. In the two-dimensional case it is provable optimal. As a consequence, Delaunay-based mesh generators are in many cases robust and produce provable good meshes [46, 267]. For difficult inputs, heuristic methods, like the advancing-front method, might fail to construct a high-quality mesh. These properties made them popular in the computational geometry community as well as in the field of numerical mathematics. Nonetheless, it is also known that *advancing-front methods* often outperform *Delaunay-based approaches* if the input is sufficiently simple.

6.3.4 Mesh improvement

Mesh improvement manipulates a given mesh to improve its elements. The mesh had to be constructed in the first place. Therefore, mesh improvement relies on other meshing algorithms. The most common mesh improvement methods rely on *flipping* and *smoothing* [27]. In 2-d, flipping exchanges the diagonal of a quadrilateral formed by two neighboring triangles, compare Fig. 6.5.

While flipping changes the mesh connectivity, smoothing adjusts the location of mesh vertices – the topology remains invariant. One of the most commonly used smoothing technique is the *Laplacian smoothing* [27]. It iteratively repositions a vertex $\mathbf{v}_i$ to

$$\overline{\mathbf{v}}_i = \frac{1}{m} \sum_{j=1}^{m} \mathbf{u}_j, \tag{6.3}$$

where m is the number of adjacent vertices $\mathbf{u}_1, \ldots, \mathbf{u}_m$ of $\mathbf{v}_i$. Other types of smoothing algorithms use optimization to determine new vertex positions. Global optimization considers all vertices at

once while the local optimizer adjusts vertex positions one by one. The former is computationally expensive while the computation costs of local optimizations are comparable to the Laplacian smoothing [27].

EikMesh implements a flipping and a force-based improvement technique similar to DistMesh [221]. Strictly speaking, they do not smooth the mesh because topological changes are possible. Furthermore, EikMesh and DistMesh employ both: a mesh generation and a mesh improvement.

6.4 Triangulations

Definitions of this section follow the description in [275, Chapter 1]. Elements of a triangular mesh are vertices, edges, and faces. They are the convex hull of some finite point set $X \in \mathbb{R}^d$.

Definition 6.1 (convex hull)**.** Let $X \subseteq \mathbb{R}^d$ be a finite point set, and $|X| = k$ be the number of points in X then

$$\operatorname{con}(X) = \left\{ \sum_{i=1}^{k} \lambda_i \mathbf{x}_i \;\middle|\; k > 0, \mathbf{x}_i \in X, \lambda_i \in \mathbb{R}^+, \sum_{j=1}^{k} \lambda_i = 1 \right\} \tag{6.4}$$

is the *convex hull* of X.

If we drop the requirement that all weights λ_i have to be positive, we get a superset of $\operatorname{con}(X)$ which is the *affine hull* of X. If three points are not part of one line, their affine hull is the plane that contains them. Consequently, the affine hull of X translated by some point in X is a vector space spanned by X.

Definition 6.2 (affine hull)**.** The *affine hull* $\operatorname{aff}(X)$ of a finite set $X \subseteq \mathbb{R}^d, |X| = k$ is the set of all affine combinations of elements of X, that is,

$$\operatorname{aff}(X) = \left\{ \sum_{i=1}^{k} \lambda_i \mathbf{x}_i \;\middle|\; k > 0, \mathbf{x}_i \in X, \lambda_i \in \mathbb{R}, \sum_{j=1}^{k} \lambda_i = 1 \right\}. \tag{6.5}$$

Furthermore, a point $\mathbf{y}$ is said to be *affine independent* with respect to X if it is not contained in the affine hull of X, that is, it is not an *affine combination* of the points/vectors in X. Going back to the perspective of a displaced vector space, points in X are *affinely independent* if all vectors $\mathbf{v} = \mathbf{x}_i - \mathbf{x}_j$ with $\mathbf{x}_i, \mathbf{x}_j \in X$ and $\mathbf{x}_i \neq \mathbf{x}_j$ are *linear independent*. Therefore, only $n + 1$ points of the n-dimensional space can be affine independent.

Definition 6.3 (k-flat)**.** A *k-flat*, also known as an affine subspace, is the affine hull of $k+1$ affine independent points. A 0-flat is a vertex, a 1-flat is a line, a 2-flat is a plane etc. A $(d-1)$-flat in $\mathbb{R}^d$ is called a hyperplane.

Each mesh element (vertex, edge, triangle) of a triangular mesh is a convex hull and called *k-simplex*.

Definition 6.4 (k-simplex)**.** A *k-simplex* τ is the convex hull of a set of $k+1$ affine independent points in $\mathbb{R}^d$ where $k \leq d$. A vertex is a 0-simplex, edges are 1- and triangles 2-simplices.

Note that a quadrilateral is not a simplex. The boundary of a k-simplex can be decomposed into lower-dimensional simplices. For example, the boundary of a triangle can be decomposed into three edges and three vertices. The boundary of an edge can be further decomposed into two vertices. Regardless of the dimensionality of the simplex, its 0-simplices are its vertices; its 1-simplices are its edges, and so on. Gluing simplices in the right way together gives a triangular mesh called a *homogeneous simplicial complex*.

Definition 6.5 (homogeneous simplicial complex (triangulation) [275])**.** A *homogeneous simplicial complex*, also known as triangulation, $\mathcal{T}$ of $\mathbb{R}^d$ is a finite set of simplices if

(i) for each $\tau \in \mathcal{T}$ all simplices of τ are also contained in $\mathcal{T}$,

(ii) for all pairs of the simplices $\tau_i, \tau_j \in \mathcal{T}$, their intersection $\tau_i \cap \tau_j$ is either empty, or a simplex common to both τ_i and τ_j and

(iii) all simplices of dimension smaller than d are simplices of a d-simplex.

If the intersection of two different simplices $\tau_i \cap \tau_j$ is not empty, they are *adjacent* to each other. In this case, they are also called *neighbors*. Otherwise, they are *disjoint*. The last constraint in Definition 6.5 ensures homogeneity. That is, it enforces a valid triangulation with no hanging or isolated vertices and edges, respectively. To differentiate between elements of the mesh and points which are inside the underlying space of the mesh, I use the following definition:

Definition 6.6 (underlying space of a simplicial complex)**.** Let $\tau \in \mathcal{T}$ be an element of the homogeneous simplicial complex $\mathcal{T}$, then

$$|\tau| = \text{con}(X) \tag{6.6}$$

is the *underlying space* of τ, where X contains all 0-simplices of τ. Furthermore,

$$|\mathcal{T}| = \bigcup_{\tau \in \mathcal{T}} |\tau| \tag{6.7}$$

is the *underlying space* of $\mathcal{T}$.

Furthermore, I use the following notion:

(i) $\tau \in \mathcal{T}$ if and only if τ is an element of the mesh $\mathcal{T}$,

(ii) $\mathbf{x} \in |\tau|$ if and only if $\mathbf{x}$ is contained in the underlying space $|\tau|$.

Let us now define a triangulation of a point set.

Definition 6.7 (triangulation of a point set [275])**.** Let $\mathcal{V}$ be a finite set of points in the plane. A *triangulation* of $\mathcal{V}$ is a simplicial complex $\mathcal{T}$ such that $\mathcal{V}$ is the set of vertices in $\mathcal{T}$, and the union of all the simplices in $\mathcal{T}$ is the convex hull of $\mathcal{V}$, that is, $|\mathcal{T}| = \text{con}(\mathcal{V})$.

In pedestrian dynamics we are mostly dealing with linear geometrical objects. The spatial domains consist of lines and simple polygons. In other words, the geometry can be defined by a *planar straight-line graph* (PSLG).

Definition 6.8 (convex polyhedron)**.** A convex polyhedron is the *convex hull of a finite point set.*

Definition 6.9 (linear cell)**.** A *linear k-cell* is the union of a finite number of convex k-polyhedra, all included in some common k-flat. A linear 0-cell is a vertex, a linear 2-cell is sometimes called a polygon.

Definition 6.10 (2-d planar straight-line graph (PSLG))**.** A *planar straight-line graph* (PSLG) $\mathcal{P}$ is a finite set of linear cells, for which

(i) vertices and edges in $\mathcal{P}$ form a simplicial complex,

(ii) for each polygon $C \in \mathcal{P}$, the boundary of C is a union of edges in $\mathcal{P}$, and

(iii) if two polygons in $\mathcal{P}$ intersect, their intersection is a union of edges and vertices in $\mathcal{P}$.

A PSLG is *segment-bounded* if there is one polygon $C \in \mathcal{P}$ such that $|C|$ contains all vertices, edges and polygons in $\mathcal{P}$.

In Fig. 6.3 a PSLG of a typical urban simulation scenario is illustrated. Finally, we are ready to define the simplicial tessellation of a *planar straight-line graph* (PSLG), in other words, a triangulation of $\mathcal{P}$. In fact, I define a triangulation to *conform* to the spatial domain it tessellates.

Definition 6.11 (Steiner triangulation of a PSLG)**.** Let $\mathcal{P}$ be an arbitrary PSGL, then a simplicial complex $\mathcal{T}$ is called *Steiner triangulation* of $\mathcal{P}$ if

(i) every cell in $\mathcal{P}$ can be written as a union of cells in $\mathcal{T}$ (conformity),

(ii) $|\mathcal{T}| = |\mathcal{P}|$ and

(iii) $\mathcal{T}$ contains all vertices in $\mathcal{P}$.

Additional vertices in $\mathcal{T}$ which are not contained in $\mathcal{P}$ are called *Steiner vertices.*

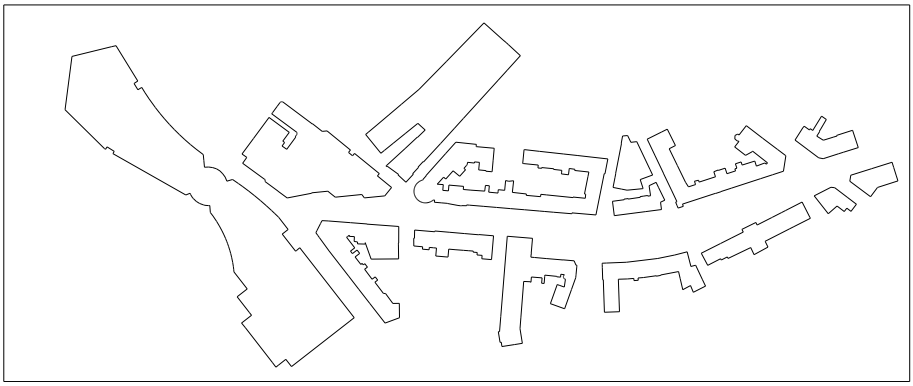


Figure 6.3: The planar straight-line graph of an urban simulation scenario.

I differentiate between a triangulation with and without *Steiner vertices*.

Definition 6.12 (triangulation of a PSLG)**.** Let $\mathcal{P}$ be an arbitrary PSGL, then a *Steiner triangulation* $\mathcal{T}$ of $\mathcal{P}$ is a *triangulation* if there are no *Steiner vertices* in $\mathcal{T}$.

Every PSLG $\mathcal{P}$ supports a triangulation $\mathcal{T}$ (without introducing *Steiner vertices*) – a well-known fact that can not be generalized for higher dimensions. In unstructured mesh generation, the goal is to compute a (conforming) *Steiner triangulation* by carefully chosen and placed *Steiner vertices*. So far I have discussed different meshing methods and established some necessary definitions for the discourse. In the following, I focus on the required algorithms EikMesh is based on. These methods are *Delaunay-based approaches*.

6.5 Triangulation computation

6.5.1 The orientation and in-circle certificate

Mesh algorithms combine combinatorial and numerical computations. Those numeric computations performed on geometric objects are called *certificates*. A certificate reveals specific geometric properties of geometrical structures and is used to decide what combinatorial operation should be executed.

The implementation of EikMesh requires two well-known certificates: the *orientation certificate* Orientation and the *in-circle certificate* InCircle. I use the *in-circle certificate* to test if the given structure represents a Delaunay triangulation. The *orientation certificate* tells us if a point is on the right or left side of a directed line defined by two other points. I use it extensively, not only to test the orientation of a triangle, but also to test for line intersections. Let $\mathbf{v}_0 = (v_{0x}, v_{0y}), \mathbf{v}_1 = (v_{1x}, v_{1y}), \mathbf{v}_2 = (v_{2x}, v_{2y})$ be consecutive vertices of a triangle τ, then the orientation certificate in $\mathbb{R}^2$ is the sign of the determinant

$$\textsc{Orientation}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) = \begin{vmatrix} v_{0x} & v_{0y} & 1 \\ v_{1x} & v_{1y} & 1 \\ v_{2x} & v_{2y} & 1 \end{vmatrix}.$$

$\textsc{Orientation}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$ returns a positive value if the vertices $\mathbf{v}_0, \mathbf{v}_1$, and $\mathbf{v}_2$ are arranged in counterclockwise order, a negative value if points are in clockwise oder, and zero if the points are collinear [275, Section 3.1]. Based on Orientation, I define the following predicates

$$\begin{aligned} \textsc{IsLeft}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{x}) &= \textsc{Orientation}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) > 0 \\ \textsc{IsRight}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{x}) &= \textsc{Orientation}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) < 0. \end{aligned}$$

$\textsc{IsLeft}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{x})$ is true if and only if $\mathbf{x}$ is on the left side of the directed line from $\mathbf{v}_0$ to $\mathbf{v}_1$. From now on, I assume that vertices of triangles are arranged in counterclockwise order.

Definition 6.13. A triangulation $\mathcal{T}$ is *valid* if and only if the Orientation is either negative for all its triangles or positive for all its triangles.

Assuming vertices of triangles are arranged in counterclockwise order, we can test if some point $\mathbf{x}$ is contained in a triangle by using the IsLeft predicate:

$$\textsc{IsContained}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{x}) = \textsc{IsLeft}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{x}) \wedge \textsc{IsLeft}(\mathbf{v}_1, \mathbf{v}_2, \mathbf{x}) \wedge \textsc{IsLeft}(\mathbf{v}_2, \mathbf{v}_0, \mathbf{x}).$$

$\textsc{IsContained}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{x})$ is true if $\mathbf{x}$ is on the left side of each line defined by an edge of the triangle, which is only the case if and only if $\mathbf{x}$ lies inside the triangle. Additionally, I use the *orientation certificate* to test if a line-segment defined by $\{\mathbf{v}_0, \mathbf{v}_1\}$ intersects a line defined by $\{\mathbf{u}_0, \mathbf{u}_1\}$. In that case, $\mathbf{v}_0$ is on the right and $\mathbf{v}_1$ on the left of the line or vise versa:

$$\begin{aligned}\textsc{Intersects}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{u}_0, \mathbf{u}_1) = &[\textsc{IsLeft}(\mathbf{u}_0, \mathbf{u}_1, \mathbf{v}_0) \wedge \textsc{IsRight}(\mathbf{u}_0, \mathbf{u}_1, \mathbf{v}_1)] \vee \\ &[\textsc{IsRight}(\mathbf{u}_0, \mathbf{u}_1, \mathbf{v}_0) \wedge \textsc{IsLeft}(\mathbf{u}_0, \mathbf{u}_1, \mathbf{v}_1)]\,.\end{aligned}$$

Now let $\mathbf{v}_3 = (v_{3x}, v_{3y})$ be some other vertex which is not part of the triangle $\tau = \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$. Then the *empty-circle certificate* is the sign of the determinant

$$\textsc{InCircle}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) = \begin{vmatrix} v_{0x} & v_{0y} & v_{0x}^2 + v_{0y}^2 & 1 \\ v_{1x} & v_{1y} & v_{1x}^2 + v_{1y}^2 & 1 \\ v_{2x} & v_{2y} & v_{2x}^2 + v_{2y}^2 & 1 \\ v_{3x} & v_{3y} & v_{3x}^2 + v_{3y}^2 & 1 \end{vmatrix}. \tag{6.8}$$

If we assume a counterclockwise arrangement of vertices, $\textsc{InCircle}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ is positive if $\mathbf{v}_3$ is not contained in the circumcircle of the triangle τ and negative otherwise. Therefore, if

$$\textsc{InCircle}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \geq 0$$

is true for vertices of any triangle $\tau = \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$ and any other vertex $\mathbf{v}_3 \notin |\tau|$, the triangulation is a Delaunay triangulation.

6.5.2 The Delaunay triangulation

The Delaunay triangulation ($\mathcal{DT}$) is one of the essential structures in computational geometry and the base for a whole set of algorithms that generate unstructured meshes, see Section 6.3.3. It is not my intention to give an extensive analysis of the Delaunay triangulation's properties

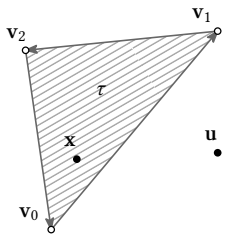


Figure 6.4: The orientation certificate for $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ is positive while for $\mathbf{v}_0, \mathbf{v}_1, \mathbf{u}$ it is negative, thus $\mathbf{u}$ is not contained in τ. $\mathbf{x}$ is contained in τ because it is left of each of the counterclockwise oriented edges. In that case, all three certificates are positive.

and all the algorithms that compute it. For a more detailed discussion, I refer to [275, Chapter 2]. However, EikMesh is a Delaunay-based mesh generator. Later on, I will introduce background meshes computed by Ruppert's algorithm, which is another Delaunay-based approach. Therefore, I shortly introduce the Delaunay triangulation and its extensions: the *constrained* and *conforming Delaunay triangulation*.

Definition 6.14 (Delaunay triangulation)**.** Let $\mathcal{V}$ be a set of vertices in the plane. A triangulation $\mathcal{T}$ is a Delaunay triangulation of $\mathcal{V}$ if for each edge e of $\mathcal{T}$ there exist a circle C with the following properties:

(i) the endpoints of edge e are on the boundary of C, and

(ii) no other vertex of $\mathcal{V}$ is in the interior of C.

I define $\mathcal{DT}(\mathcal{V})$ to be a Delaunay triangulation of $\mathcal{V}$.

In the previous section, I established the criterion the Delaunay triangulation is based on, that is, the *empty-circle certificate*. In other words, a triangulation is a Delaunay triangulation if and only if the *empty-circle certificate* is greater or equals zero for each triangle and any vertex which is not part of the triangle.

One of the most important results concerning the Delaunay triangulation is the Delaunay Lemma, proved by Boris Delaunay himself. It provides an alternative and local characterization of the Delaunay triangulation, that is, locally Delaunay edges:

Definition 6.15 ((locally) Delaunay edge [275])**.** Let e be an edge of a triangulation $\mathcal{T}$ in the plane. Then e is *locally Delaunay* if

(i) it is an edge of only one triangle, or

(ii) the open circumcircle of both triangles neighboring e do not contain the (two) vertices that are opposite of e.

Lemma 6.1 (Delaunay Lemma [60])**.** *A triangulation is a Delaunay triangulation if and only if every edge is locally Delaunay.*

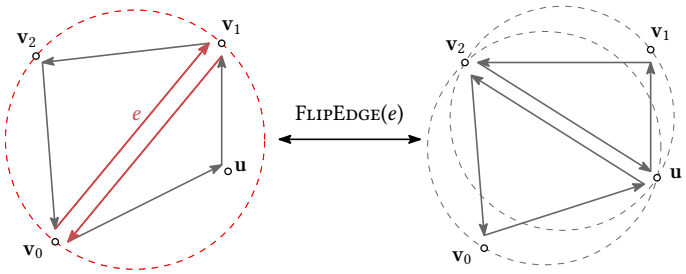


Figure 6.5: A violated empty-circle certificate: the circumcircle of the triangle defined by $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ is not a Delaunay triangle because it contains another vertex $\mathbf{u}$ of the triangulation $\mathcal{T}$. Flipping the edge resolves the problem.

Consequently, to test if a triangulation in the plane is *Delaunay*, one can evaluate the empty-circle certificate for each edge, compare Fig. 6.5.

The Delaunay triangulation is unique if there are no four vertices in $\mathcal{V}$ that are cocircular. It has a unique property in two dimensions: the Delaunay triangulation maximizes the minimum angle amongst all possible triangulations of a fixed set of points [275]. This property is especially nice because it leads to well-shaped triangles, thus to a high element quality. A more extensive discussion of the mesh quality can be found in Section 6.6.

Let d be the dimension of the Euclidean space and let n be the number of points in $\mathbb{R}^d$. Then for $d = 2$, the Delaunay triangulation of those n points can be computed in $O(n \log(n))$ time and $O(n)$ space. Many construction algorithms achieve such an optimal time and space complexity. They can be classified in the following way:

(i) **sweep line methods** [84, 277, 268]: computation of the Voronoi diagram by sweeping through the domain Ω in $O(n \log(n))$ time,

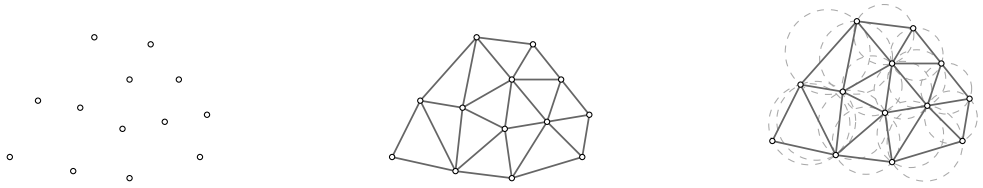
(ii) **duality approaches** [276, 41, 253, 136, 254, 37]: computation of the convex hull of the lifted points into dimension $d + 1$ gives the Delaunay triangulation for dimension d in $O(n \log(n) + n^{\lfloor d/2 \rfloor})$ time,

(iii) **divide-and-conquer algorithms** [264, 183]: computation of the Delaunay triangulation by computing multiple Delaunay triangulations of close points and merging results in overall $O(n \log(n))$ time,

(iv) **flip algorithms** [181, 77, 275]: flipping of edges to make them locally Delaunay until the triangulation is a Delaunay triangulation. Requires $O(n^2)$ time for $d = 2$,

(v) **incremental methods** [62, 34, 108, 107, 183]: computation of the Delaunay triangulation by insertion of one point after another in $O(n^{\lceil d/2 \rceil + 1})$ deterministic and $O(n \log(n) + n^{\lceil d/2 \rceil})$ expected time.

Some methods are appropriate in the *static* setting [84, 277, 268, 264, 183], where vertices are *fixed* and *known* in advance. Some are more suitable for a *dynamic* or *online* setting [62, 34, 108, 107, 183], where vertices are *fixed*, but the triangulation is maintained under vertex insertions and deletions. EikMesh can deal with a *dynamic* setting; additionally, vertices *move* around. In the following, I describe the two algorithms EikMesh uses: FlipAlgorithm and an incremental approach.



(i) point set $\mathcal{V}$ (ii) $\mathcal{DT}(\mathcal{V})$ (iii) Empty circumcircles

Figure 6.6: The Delaunay triangulation (ii) of a point set (i): all circumcircles are empty (iii).

Algorithm 7: FlipAll

Input: a triangulation $\mathcal{T}$
Output: a Delaunay triangulation $\mathcal{T}$

```
while ∃ edge e ∈ T not locally Delaunay do
    T ← FlipEdge(e);
return T;
```

The flip algorithm

The flip algorithm introduced by [181] takes some given triangulation $\mathcal{T}_0$ and transforms it into a Delaunay triangulation by flipping edges that are not locally Delaunay. A flip of an edge might remove the *locally Delaunay* property from any of its four surrounding edges. But a flipped edge, which was not *locally Delaunay*, can never reappear [275]. Therefore, the following algorithm terminates after at most $n(n-1)/2$ flips and generates a Delaunay triangulation.

In Section 8.2, I explain how a slight adaptation of FlipAll converts almost all edges into *locally Delaunay* edges in the improvement phase of EikMesh. The basic idea is that with an increasing number of Delaunay edges, Algorithm 7 terminates after a few flips. In other words, the more 'Delaunay' a triangulation is, the less work has to be done.

Incremental construction

Incremental approaches do not require an initial triangulation of the whole point set but can start from scratch. Let $\mathcal{V}$ with $|\mathcal{V}| = n$ be our point set and $\mathcal{V}_i$ the point set of iteration i. We start with a (virtual) large triangle $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ that contains all points $\mathcal{T}_0 = \mathcal{DT}(\mathcal{V}_0)$, with $\mathcal{V}_0 = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$. I implement Lawson's algorithm [182] to add vertices $\mathbf{v}_i \in \mathcal{V}$ one by one to the triangulation such that

$$\begin{aligned} \mathcal{V}_{i+1} &= \mathcal{V}_i \cup \mathbf{v}_{i+1} \\ \mathcal{T}_{i+1} &= \mathcal{DT}(V_{i+1}). \end{aligned} \tag{6.9}$$

To insert a new vertex $\mathbf{v}_{i+1}$, three steps are necessary: first, I search for the enclosing triangle τ_i with $\mathbf{v}_{i+1} \in |\tau_i|$. I discuss efficient traversal methods to locate τ_i in Section 6.7. In the second step, I insert $\mathbf{v}_{i+1}$ into $\mathcal{T}_i$ by SplitFace(τ_i,$\mathbf{v}_{i+1}$). This local operation splits τ_i into three triangles by connecting $\mathbf{v}_{i+1}$ with each of the vertices of τ_i. Due to the new vertex, multiple edges may no longer be Delaunay edges. To reestablish the Delaunay criterion for all edges, I apply the FlipAll algorithm from the previous section in a breadth-first fashion until all edges are *locally Delaunay*

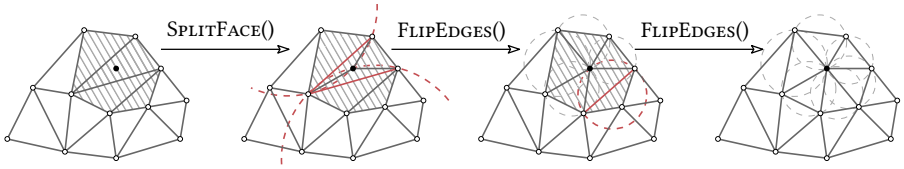


Figure 6.7: Vertex insertion into a given Delaunay triangulation.

– the last two steps are depicted in Fig. 6.7. After all vertices are inserted, I remove $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$, and all neighboring triangles from the triangulation, thus

$$\begin{aligned} \mathcal{V}_{n+1} &= \mathcal{V} = \mathcal{V}_n \setminus \mathcal{V}_0 \\ \mathcal{T}_{n+1} &= \mathcal{DT}(V_{n+1}) = \mathcal{DT}(\mathcal{V}) \end{aligned} \tag{6.10}$$

is the finalized triangulation. Additional points can still be inserted if they lie inside some triangle of the current triangulation.

Guibas et al. [108] proved that using a random insertion order, $O(n)$ connectivity changes are required. In that case, the point location problem requires $O(\log(n))$ expected time such that the construction requires overall $O(n \log(n) + n) = O(n \log(n))$ expected time.

EikMesh relies on a small number of local mesh operations, amongst others

(i) InsertVertex($\mathbf{v}$),

(ii) RemoveVertex($\mathbf{v}$),

(iii) FlipEdge(e) and

(iv) Move($\mathbf{v}, \Delta\mathbf{x}$).

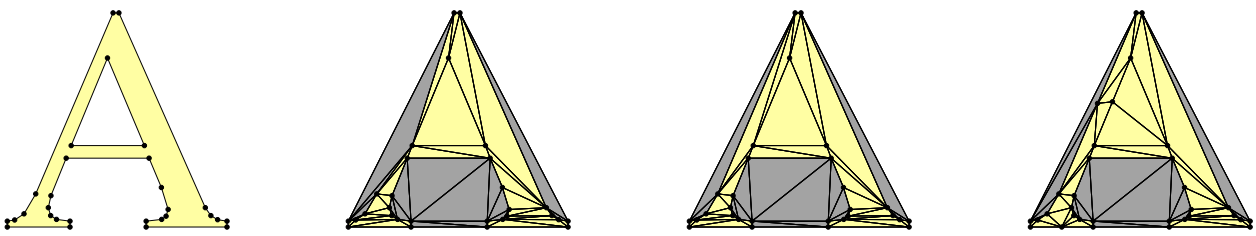
Lawson's algorithm shares this reliability and uses the exact same operations – one can replace Move by combining RemoveVertex and InsertVertex. In addition, the algorithm is suitable for the *dynamic* setting. Therefore, EikMesh uses the incremental method of Lawson [182], also presented in [77].

6.5.3 The constrained Delaunay triangulation

The Delaunay triangulation is a useful geometric structure. Still, without geometric constraints in the form of line-segments, the Delaunay triangulation often fails to conform to planar straight-line graphs (PSLGs). A mesh $\mathcal{T}$ of a planar straight-line graph $\mathcal{P}$ should conform to $\mathcal{P}$, therefore,

(i) all vertices of $\mathcal{P}$ have to be contained in $\mathcal{T}$, and

(ii) each line-segment $l \in \mathcal{P}$ has to be the union of edges e in $\mathcal{T}$.



(i) (ii) (iii) (iv)

Figure 6.8: Different triangulations of a planar straight-line graph (i): its Delaunay triangulation (ii) is not conforming while its constrained Delaunay triangulation (iii) is not a Delaunay triangulation. Only a conforming Delaunay triangulation (iv) is Delaunay as well as conforming.

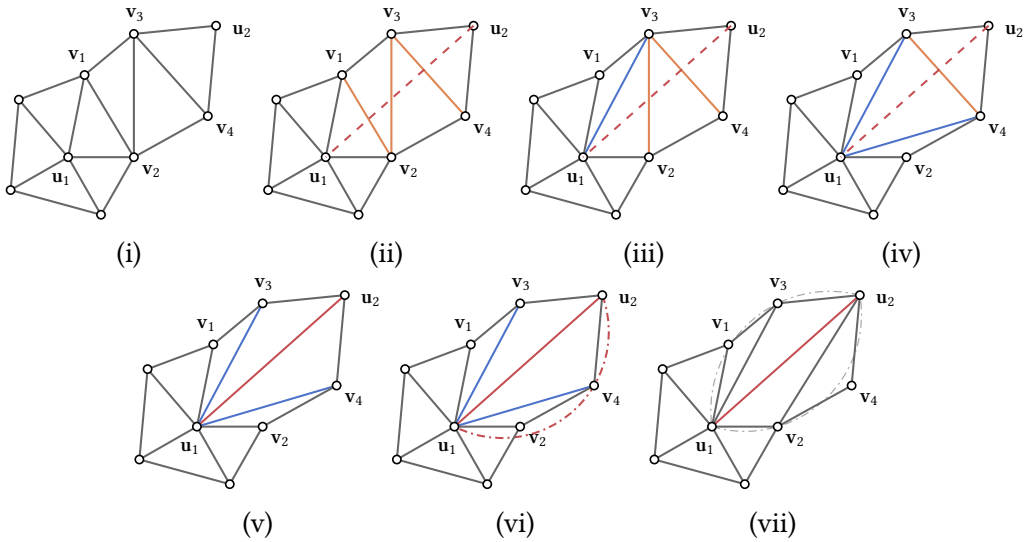


Figure 6.9: Establishment of a constraint of the constrained Delaunay triangulation: the triangulation before the insertion of the constraint (i) and after the insertion has finished (vii). First, all intersecting edges are identified (ii) and flipped (iii)-(v). Illegal edges are flipped again (vi), compare [278].

The *constrained Delaunay triangulation* is a triangulation that is 'almost Delaunay', i. e., the Delaunay criterion is only violated for the inserted line-segments called constraints.

Definition 6.16 (constrained Delaunay triangulation of a planar straight-line graph)**.** Let $\mathcal{P}$ be a planar straight-line graph. A triangulation $\mathcal{T}$ is a *constrained Delaunay triangulation* (CDT) of $\mathcal{P}$ if each line-segment of $\mathcal{P}$ is an edge of $\mathcal{T}$ and for each remaining edge e of $\mathcal{T}$, there exists a circle C with the following properties:

(i) the endpoints of edge e are on the boundary of C, and

(ii) if any vertex $\mathbf{v}$ of $\mathcal{P}$ is in the interior of C, then it cannot be 'seen' from at least one of the endpoints of e, that is, if you draw the line segments from $\mathbf{v}$ to each endpoint of e, then at least one of the line segments crosses an edge of $\mathcal{P}$.

I implemented Sloan's algorithm [278] to construct the constrained Delaunay triangulation for a given planar straight-line graph $\mathcal{P}$. It starts with an initial Delaunay triangulation of the points set $\mathcal{V}$ of $\mathcal{P}$. After $\mathcal{DT}(\mathcal{V})$ has been computed by Lawson's algorithm [182], constraint line-segments are recovered: Sloan's algorithm flips edges that cross line-segments. An edge is only flipped if its two neighboring triangles form a strictly convex quadrilateral. Otherwise, the flip is postponed. After the constraint is established, all flipped edges are flipped again until all of them are *locally Delaunay* ignoring points on the other side of the constraint. The process is depicted in Fig. 6.9. Sloan's algorithm requires $O(n \log(n))$ time, where n is the number of points in $\mathcal{P}$.

Since the constrained Delaunay triangulation is not a Delaunay triangulation, we lose some of its properties. To reduce the size of triangles in $\mathcal{T}$, a Steiner triangulation of $\mathcal{P}$ is usually desirable. However, Steiner points can also be used to transform a constrained Delaunay triangulation into

a *conforming Delaunay triangulation*. This transformation requires edge splits of non-Delaunay edges and is depicted in Fig. 6.10.

Definition 6.17 (conforming Delaunay triangulation of a planar straight-line graph)**.** Let $\mathcal{P}$ be a planar straight-line graph. A triangulation $\mathcal{T}$ is a *conforming Delaunay triangulation* of $\mathcal{P}$ if each line-segment of $\mathcal{P}$ is the union of edges of $\mathcal{T}$ and $\mathcal{T}$ is a Delaunay triangulation.

6.5.4 Ruppert's algorithm

Ruppert's algorithm [241, 242] was the first provable-good Delaunay-based refinement technique. It transforms a PSLG $\mathcal{P}$ into a conforming Delaunay triangulation such that all triangle angles α are larger than some minimum angle α_{min}. In [241], Ruppert showed that his algorithm is guaranteed to terminate if $\alpha_{\text{min}} \leq 20.7°$. If we choose the maximum, i. e., $\alpha_{\text{min}} = 20.7°$, the angles of the triangulation are in between 20.7 and 138.6 degrees. Ruppert's algorithm requires that adjacent edges of the PSLG $\mathcal{P}$ meet at non-acute angles. However, there are multiple techniques, such as protecting discs [229], to deal with acute angles determined by $\mathcal{P}$. Obviously, we can never get rid of small angles defined by the edges of the PSLG.

I use Ruppert's algorithm to generate background meshes to construct an element size function (Section 8.6) and the distance function (Section 8.7) that support the mesh generation of EikMesh.

Definition 6.18 (encroached edges)**.** Let e be an edge of a triangulation $\mathcal{T}$ then e is *encroached* if and only if any vertex $\mathbf{v} \in \mathcal{V}$ with $\mathbf{v} \notin e$ lies within the closed diametric circle of e.

Ruppert's algorithm refines triangles if the triangle's angles do not satisfy the minimum angle condition. Let $\mathbf{c}$ be the circumcenter of the triangle τ, then Ruppert's algorithm refines τ by inserting $\mathbf{c}$ into the Delaunay triangulation if the insertion will not lead to an edge e being *encroached*. Otherwise, e is split at its midpoint, compare Fig. 6.10. During the refinement, the Delaunay criterion is always enforced – after each modification, the result is a Delaunay triangulation. If the minimum angle condition is satisfied for all angles, Ruppert's algorithm terminates. Typically, an element size function h can be used as an additional input to enforce element size constraints. For an extensive description, I refer to [275, Chapter 6].

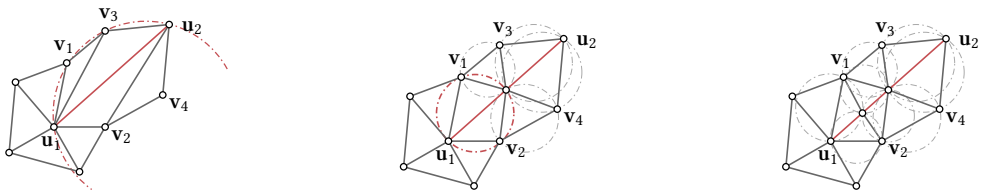


Figure 6.10: Computation of the conforming Delaunay triangulation: in (i), multiple vertices are enclosed by the circle defined by $\mathbf{v}_1$, $\mathbf{v}_3$, and $\mathbf{u}_1$, leading to a split of the red constrained edge. Red circles indicate an edge that is not locally Delaunay. After the second split (iii), the result is a Delaunay triangulation.

6.6 Triangulation quality

In this section, I introduce well-known quality measures, which I use to evaluate the performance of DistMesh and EikMesh. Furthermore, I discuss the effects of, and the relation between small and large angles.

Meshes can be categorized by many different properties such as their regularity, their underline base elements, how computationally expensive their generations is, and what kind of geometries they can represent. The established and flourishing branch of computational geometry brought many interesting results, including meshing algorithms that guarantee specific quality requirements such as the smallest and largest possible angles. Despite this theoretical breakthrough, the development of meshing algorithms that lack theoretical guarantees has not been stopped, since they are often superior in practice. EikMesh falls into this category. Therefore, a theoretical quality comparison is pointless. Instead, I compare the output under a fair and useful quality measure.

6.6.1 Why mesh quality matters

First of all, refining triangulations without dropping too much in quality is simple compared to the reverse operation. For example, splitting each triangle into 4 children by introducing a new vertex at the midpoint of each edge will preserve the mesh's quality, compare Fig. 6.11. Coarsening a fine mesh while keeping the mesh quality high is much more challenging. Therefore, the goal of most mesh generators is to compute high-quality meshes using as few mesh elements as possible. EikMesh is no exception.

I am interested in isotropic triangulations, for which the edge length does not depend on its orientation. Consequently, equilateral triangles are desirable. They are perfectly regular and produce the best results in most applications. In general, well-shaped elements are preferable because of the following two restrictions: (1) avoidance of large angles and (2) avoidance of small angles.

Avoidance of large angles: elements with large angles introduce large errors in numerical approximations of differential operators. More precisely, the error in the gradients computed using piece-wise linear interpolation becomes unbounded if the angle of a triangle approaches π [269, p. 23]. An example is illustrated in [275, Chapter 1]. These large errors are critical for solving the eikonal equation since numerical solvers rely on a numerical approximation of $\|\nabla\Phi\|$, see Section 9.2. Secondly, due to the causality condition of the numerical solvers introduced in Section 9.2.3, one must avoid non-acute angles to compute such an approximation. We can deal

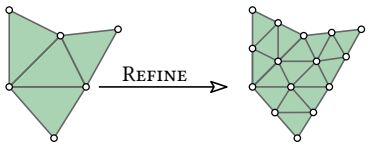


Figure 6.11: Simple mesh refinement: each edge is split at its midpoint. Each child triangle is congruent to its parent.

with those large angles but only by losing some accuracy in the approximation of the eikonal equation's solution.

Avoidance of small angles: elements with very small angles lead to poorly conditioned numerical integration schemes. Some methods require that the circumcenter lies within the element which is only true if and only if no angle is greater than $1/2\pi$ [28]. Even though small angles are less of an issue in my application, where we do not rely on integration schemes, there is an obvious connection between small and large angles: small angles lead to large angles, compare Eq. (6.13) and Eq. (6.15) below.

To simplify the remaining part of this section, I consider a non-degenerated triangle $\tau = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ with the area $|\tau|$, perimeter $p(\tau)$, edges of length $a = \|\mathbf{b}-\mathbf{c}\|$, $b = \|\mathbf{a}-\mathbf{c}\|$ and $c = \|\mathbf{a}-\mathbf{b}\|$, and denote the angle at vertex $\mathbf{a}$ (resp. $\mathbf{b}$, $\mathbf{c}$) as α (resp. β, γ) and the radius of the inscribed (resp. circumscribed) circle of τ as r_{in} (resp. r_{out}). Additionally, I use the following notation:

(i) $|\tau|_0 = \min\{a, b, c\}$,

(ii) $|\tau|_2 = \sqrt{a^2 + b^2 + c^2}$,

(iii) $|\tau|_\infty = \max\{a, b, c\}$,

(iv) $\theta_0 = \min\{\alpha, \beta, \gamma\}$,

(v) $\theta_\infty = \max\{\alpha, \beta, \gamma\}$.

6.6.2 Relation between small and large angles

Let me shortly analyze the relation between the smallest and largest angle of an element. The trivial fact $\pi = \alpha + \beta + \gamma$ yields a connection between the minimal θ_0 and maximal θ_∞ angel of a triangle. By definition

$$0 < \theta_0 \leq \frac{\pi}{3} \leq \theta_\infty \tag{6.11}$$

holds for a non-degenerated triangle. Using the pigeonhole principle $\theta_0 \leq \frac{\pi}{3}$ and

$$\pi - \theta_0 \leq 2\theta_\infty \ \wedge \ \theta_\infty \leq \pi - 2\theta_0 \tag{6.12}$$

follows. It gives us the bounds for the maximal angle that depends on the θ_0:

$$\frac{\pi - \theta_0}{2} \leq \theta_\infty \leq \pi - 2\theta_0 \text{ with } \theta_0 \in \left]0, \frac{\pi}{3}\right]. \tag{6.13}$$

We can establish similar bounds for the minimal angle by

$$0 < \theta_0 \leq \frac{\pi - \theta_\infty}{2} \text{ with } \theta_\infty \in \left[\frac{\pi}{3}, \pi\right[, \tag{6.14}$$

receptively. If θ_∞ is small, more precisly, if $\theta_\infty < \pi/2$ we get a tighter bound

$$\pi - 2\theta_\infty < \theta_0 \leq \frac{\pi - \theta_\infty}{2} \text{ with } \theta_\infty \in \left[\frac{\pi}{3}, \frac{\pi}{2}\right[. \tag{6.15}$$

Combining Eq. (6.15) and Eq. (6.13), it follows that a small largest angle implies a large smallest angle and vise verse.

6.6.3 Common quality measures

The relation between the incircle radius r_{in} and the circumscribed circle radius r_{out} of a triangle gives us the first useful quality measure (Definition 6.19). For a equiliteral triangle $2r_{\text{in}} = r_{\text{out}}$ holds. Furthermore, the relation between smallest and largest angle is given by the following transformation:

$$\begin{aligned}\rho_1(\tau) = \frac{2r_{\text{in}}}{r_{\text{out}}} &= \frac{2\sin(\alpha)\sin(\beta)\sin(\gamma)}{\sin(\alpha)+\sin(\beta)+\sin(\gamma)} \\ &= \frac{2\sin(\alpha)\sin(\beta)\sin(\alpha+\beta)}{\sin(\alpha)+\sin(\beta)+\sin(\alpha+\beta)} \\ &= \frac{2\sin(\theta_0)\sin(\theta_\infty)\sin(\theta_0+\theta_\infty)}{\sin(\theta_0)+\sin(\theta_\infty)+\sin(\theta_0+\theta_\infty)}.\end{aligned} \tag{6.16}$$

The second useful measure (Definition 6.20) is given by the longest edge to incircle radius ratio.

Definition 6.19 (radius-ratio [82])**.** Let r_{in} be the incircle radius and r_{out} be the circumscribed circle radius of τ. Then

$$\rho_1(\tau) = \frac{2r_{\text{in}}}{r_{\text{out}}} \tag{6.17}$$

is the *radius-ratio* of τ.

Definition 6.20 (longest edge to incircle raduis ratio [82])**.** Let $|\tau|_\infty$ be the edge length of the longest edge. Then

$$\rho_2(\tau) = 2\sqrt{3}\frac{r_{\text{in}}}{|\tau|_\infty} \tag{6.18}$$

is *longest edge to incircle raduis ratio* of τ.

The *radius-ratio* as well as the *longest edge to incircle raduis ratio* possess all four important properties of a measure ρ discussed in [82]:

(1) **detection**: it detects **all** degenerated elements,

(2) **non-dimensionality**: congruent triangles have the same quality,

(3) **boundedness**: the quality can not be arbitrarily large or small,

(4) **normalized**: $\forall\tau : \rho(\tau) \in [0;1]$.

Most importantly (1) ensures that

$$\rho_1(\tau) = \rho_2(\tau) = 0 \tag{6.19}$$

holds, if and only if τ is degenerated, i. e., if all three points are co-linear. Furthermore, it ensures that

$$\rho_1(\tau) = \rho_2(\tau) = 1 \tag{6.20}$$

holds, if and only if τ is a equilateral triangle. To compare different meshes I use the (overall) *mesh quality* (Definition 6.21), the *quality per element* ρ_1, ρ_2 and the *minimal quality* (Definition 6.22).

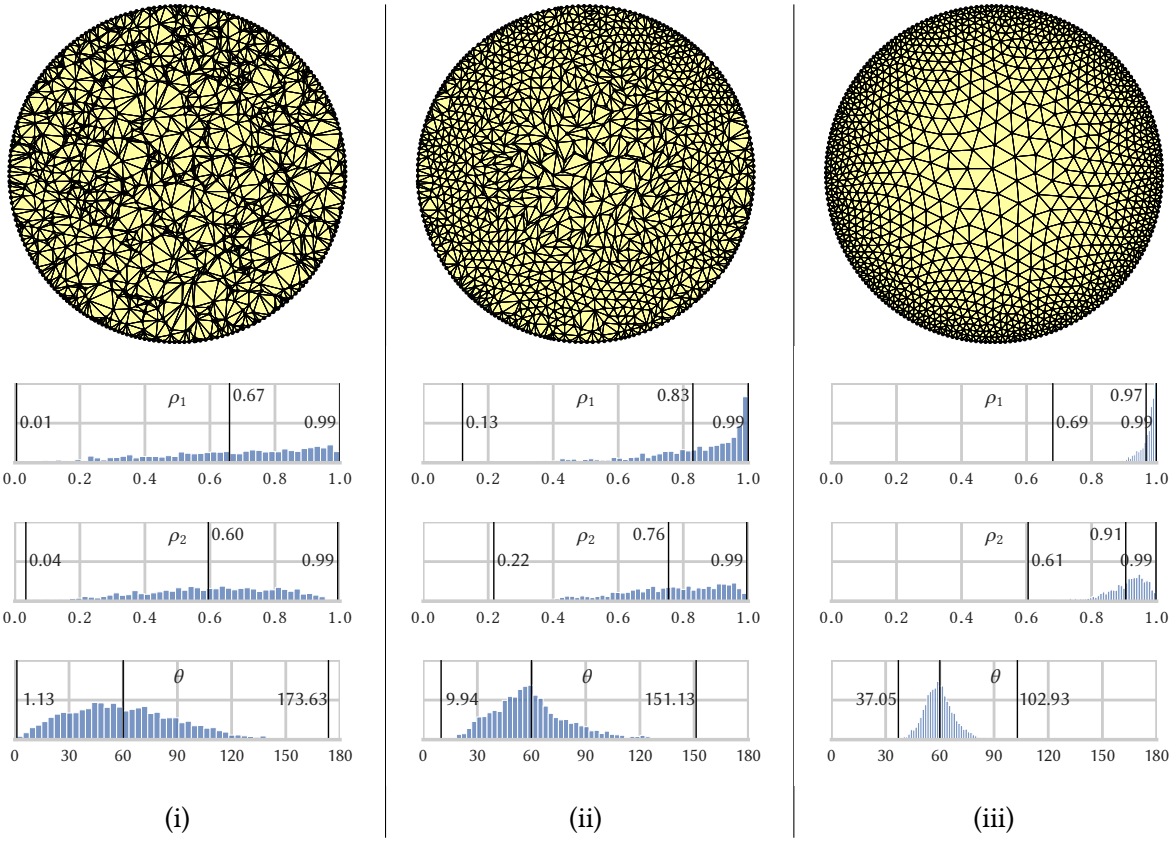


Figure 6.12: Different meshes and the quality and angles of their elements: the first row shows a histogram of ρ_1 and the second row of quality ρ_2, respectively. Histograms in the third row display triangle angles. Each histogram is normalized. The black vertical lines mark minimal, mean and maximal values.

Definition 6.21 (mesh quality)**.** Let $\mathcal{T}$ be a triangular mesh and $\rho \in \{\rho_1, \rho_2\}$ be the trianlge quality measure of our choice. Then

$$\rho(\mathcal{T}) = \frac{1}{m} \sum_{\tau \in \mathcal{T}} \rho(\tau) \tag{6.21}$$

is the quality of the mesh, where m is the number of triangles of $\mathcal{T}$.

Definition 6.22 (minimal triangle quality)**.** Let $\mathcal{T}$ be a triangular mesh and $\rho \in \{\rho_1, \rho_2\}$ be the triangle quality measure of our choice. Then

$$\rho_{\min}(\mathcal{T}) = \min_{\tau \in \mathcal{T}} \rho(\tau) \tag{6.22}$$

is the minimal triangle quality, i. e. the quality of the 'worst' element.

6.7 Point location algorithms

Given a triangular unstructured 2-d mesh $\mathcal{T}$ of n vertices and a point $\mathbf{q} \in |\mathcal{T}|$, a fundamental problem in computational geometry is to find the triangle or face τ such that $\mathbf{q} \in |\tau|$. How one solves the problem does not influence the final mesh generated, but the operation determines the generator's computational cost. Additionally, evaluating the navigation field $\Phi(\mathbf{q})$ or the distance function $d_\Omega(\mathbf{q})$ at an arbitrary point $\mathbf{q} \in \Omega$, where Φ and d_Ω are only defined for vertices of a mesh, requires interpolation. And to interpolate, we have to know neighboring vertices of the triangle τ that contains $\mathbf{q}$. Therefore, evaluating $\Phi(\mathbf{q})$ consists of a point location. It is also an essential part of incremental Delaunay-based meshing algorithms, such as [311, 35]. They require the location of τ containing $\mathbf{v}$ for each $\mathbf{v}$ of the final triangulation.

In general, mesh generation and the usage of unstructured meshes requires point location regularly. Since point location is not trivial for unstructured meshes, an efficient implementation is beneficial for both. Before moving on to a more detailed discussion of solving the point location problem, I introduce the following definition:

Definition 6.23 (k-ring)**.** Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the k-ring of a vertex $\mathbf{v} \in V$ is the set of vertices for which the shortest path to $\mathbf{v}$ is smaller or equal to k.

6.7.1 Walk strategies

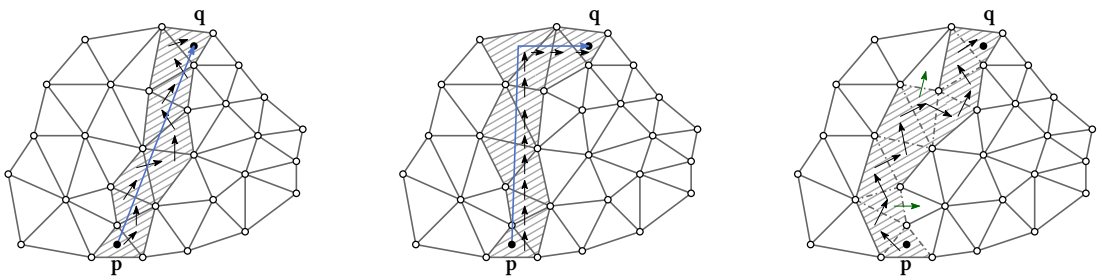
One can solve the point location by testing all triangles one-by-one. This naive approach requires $O(n)$ time, where n is the number of triangles in $\mathcal{T}$. More sophisticated so-called *walking strategies* achieve much better run times in practice. Devillers et al. [63] described and analyzed four of them:

(i) StraightWalk,

(ii) OrthogonalWalk,

(iii) VisibilityWalk,

(iv) StochasticWalk.



(i) StraightWalk (ii) OrthogonalWalk (iii) StochasticWalk

Figure 6.13: A straight (i), orthogonal (ii) and stochastic (iii) walk through a Delaunay triangulation of a random point set.

Algorithm 8: StraightWalk

Input: start triangle τ_0, querry point $\mathbf{q}$
Output: a triangle τ_i containing $\mathbf{q}$

```
1 p ← midpoint of τ_0, e_0 ← ∅, i ← 0;
2 while ¬IsContained(τ_i, q) do
3     foreach e = {v_0, v_1} edge in τ_i do
          /* treatment of the degenerated case is missing */
4         if e ≠ e_i ∧ Intersects(v_0, v_1, p, q) then
5             e_{i+1} ← e;
6             τ_{i+1} ← triangle τ with τ ∩ τ_i = e_{i+1};
7     i ← i + 1;
8 return τ_i;
```

Walking strategies operate directly on the mesh $\mathcal{T}$ without any additional data structure. They walk from a starting point $\mathbf{p}$ to the query point $\mathbf{q}$ by following a sequence $\tau_0, \ldots, \tau_k$ of neighboring faces. They only differ from the choice of the next visited face. If there is no predefined starting triangle τ_0, it is commonly chosen at random, and the starting point $\mathbf{p}$ is defined such that it lies within τ, e. g., the triangle's midpoint.

A StraightWalk is displayed in Fig. 6.13i. It traverses triangles by choosing the face that has an edge intersecting the line $(\mathbf{p}, \mathbf{q})$.

The VisibilityWalk chooses one of at most two edges, which separates the currently visited face from $\mathbf{q}$. More precisely, if we split $\mathbb{R}^2$ by the line defined by the two points of the edge into two parts Ω_1, Ω_2 with $\mathbf{q} \in \Omega_1$, then we continue the walk by its neighboring triangle τ if $|\tau| \subset \Omega_1$. By its definition, the walk does not require the handling of degenerated cases. However, for an arbitrary triangulation, the VisibilityWalk may cycle and therefore does not necessarily terminate. It is still popular since it terminates for any Delaunay triangulation. Furthermore, randomly choosing one of the two possible next edges guarantees termination [63]. This probabilistic extension is called StochasticWalk depicted in Fig. 6.13iii.

The OrthogonalWalk, illustrated in Fig. 6.13ii, visits triangles along an isothetic path from $\mathbf{p}$ to $\mathbf{q}$ by changing one coordinate at a time. The cost of evaluating an intersection predicate increases with the dimension. Decomposing the walk into pieces parallel to the coordinate axis simplifies this evaluation. For example, to test if an edge $((u_x, u_y), (v_x, v_y))$ intersects the ray $y = c$ we only have to test if

$$(u_y < c \ \wedge \ v_y > c) \ \vee \ (u_y > c \ \wedge \ v_y < c) \tag{6.23}$$

is satisfied. However, this strategy can lead to an increase in required tests [63].

Handling degenerated cases

When using the StraightWalk or the OrthogonalWalk, we may have to deal with degenerated cases, meaning that an edge might be part of the ray $(\mathbf{p}, \mathbf{q})$. As stated by Devillers et al. [63] and experienced by myself, handling degenerated cases leads to intricate code.

I solve the problem by the following strategy: Let us assume the ray goes precisely through some edge $\{\mathbf{v}, \mathbf{u}\}$, as seen in Fig. 6.14. Now let $\mathbf{u}_0, \ldots, \mathbf{u}_m$ be the counterclockwise arranged vertices of the 1-ring of $\mathbf{v}$. I test for line intersection between the ray and the line segments

$$(\mathbf{u}_0, \mathbf{u}_2), (\mathbf{u}_1, \mathbf{u}_3), \ldots (\mathbf{u}_{m-1}, \mathbf{u}_0), (\mathbf{u}_m, \mathbf{u}_1), \tag{6.24}$$

excluding the entering segment of the walk. Suppose $(\mathbf{u}_i, \mathbf{u}_{i+2})$ is the segment intersecting the ray. Then we know that the ray goes precisely through $\mathbf{u}_{i+1} = \mathbf{u}$, and I continue the walk with either $(\mathbf{u}_i, \mathbf{u}_{i+1})$ or $(\mathbf{u}_{i+1}, \mathbf{u}_{i+2})$.

In the worst case, this algorithm requires $O(d)$ steps where d is the degree of $\mathbf{v}$. However, the described situation rarely occurs in practice.

6.7.2 Accelerated point location

Point location algorithms accelerate *walking strategies* by supportive data structures or some additional method that computes a beneficial starting face τ_0. I implemented and tested four different point location algorithms:

(i) JumpAndWalk [64],

(ii) Delaunay-Tree [108],

(iii) Delaunay-Hierarchy [62], and

(iv) PlainWalk (no additional data structure or method).

For a random insertion order of n points, using the Delaunay-Tree or the Delaunay-Hierarchy leads to a time complexity of $O(\log(n))$ for each point location [108, 62]. The Delaunay-Tree is used by incremental methods to compute the Delaunay triangulation. It is a bookkeeping tree. Each node in the tree represents one of two operations required to construct the Delaunay triangulation: SplitFace or FlipEdge. The root is the virtual all-enclosing triangle. Its children are the three triangles generated by the first SplitFace operation. Locating a triangle is realized by a depth-first traversal of the tree. Therefore, the Delaunay-Tree does not depend on any walking strategy. However, removing or repositioning already inserted vertices is not supported by the Delaunay-Tree. It would require the adjustment of large parts of the history, possibly the whole tree, which is too computationally expensive.

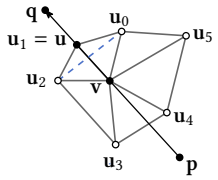


Figure 6.14: Dealing with degenerated cases: if the ray $(\mathbf{p}, \mathbf{q})$ goes through an edge $\{\mathbf{v}, \mathbf{u}_1\}$, the walk continues with either the edge $\{\mathbf{u}_2, \mathbf{u}_1\}$ or $\{\mathbf{u}_1, \mathbf{u}_0\}$.

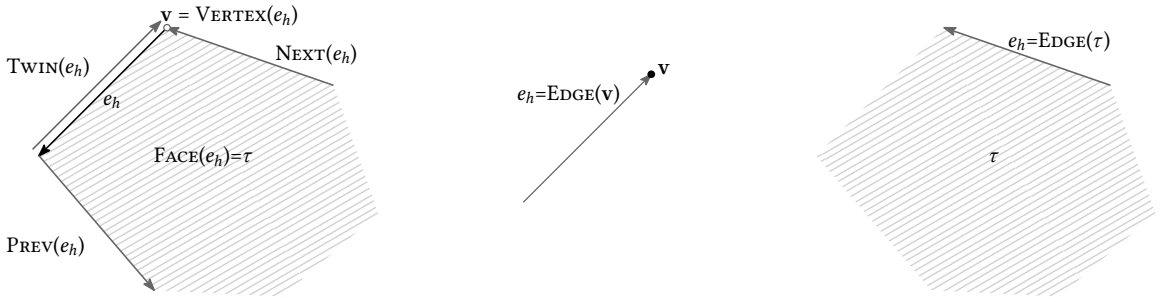
JumpAndWalk picks a random sample of $n^{1/3}$ vertices and starts the walk from the sample point, which is closest to the query point $\mathbf{q}$. It requires $O(n^{1/3})$ time [64]. Therefore, it performs theoretically worse than the Delaunay-Tree and the Delaunay-Hierarchy but is very effective in practice. Furthermore, JumpAndWalk does not require any additional data structure. Therefore, it offers the most flexibility, and its implementation is straightforward.

The Delaunay-Hierarchy supports removing or repositioning of already inserted vertices and performs well in theory and practice. However, its implementation is much more complicated, and in practice, I did not achieve a better performance compared to JumpAndWalk. Therefore, I choose JumpAndWalk to be my default point location algorithm.

Furthermore, consecutive point location queries are accelerated by a cache. I call this strategy CachedLocation. Let o be some object and $\mathbf{q}$ be the query point, then CachedLocation($\mathbf{q}$, o) executes JumpAndWalk if there is no result already available for the object o. Otherwise, it starts the *walk* from the last located triangle τ related to o. This cached point location strategy performs well if consecutive query points for the same object o are close together. Since agents move on a connected path across the underlying mesh, CachedLocation performs exceptionally well.

6.8 Mesh data structure

Several data structures can represent unstructured meshes, see [266, 188, 58]. In this thesis, I use the so-called *half-edge data structure*, also known as *doubly-connected edge list* (DCEL) [58, Chapter 2]. The components of the data structure are depicted in Fig. 6.15. A DCEL can manage not only triangulations but also planar straight-line graphs (PSLGs) in general. It contains a record for each *face*, *half-edge*, and *vertex* and incorporates structural and topological information such as the set of edges bounding a *face* and the adjacency relation of k-simplices. A *face* of a DCEL is a polygonal region whose boundary is formed by *half-edges* and *vertices*. In other words, a *face* is either a 2-simplex, that is, a triangle, the spatial domain $\mathbb{R}^2 \setminus \Omega$ (which I refer to as *border*), or a *hole*, that is, a simple polygon surrounded by 2-simplices. Each edge of the mesh or the PSLG is represented by two counterclockwise (CCW) arranged *half-edges* (one for each neighboring



(i) e_h (ii) $\mathbf{v}$ (iii) τ

Figure 6.15: The doubly-connected edge list data structure: given a half-edge e_h (i) one can access its Twin(e_h), predecessor Prev(e_h), successor Next(e_h), Vertex(e_h) and Face(e_h) in constant time. One can access some half-edge which end in the vertex $\mathbf{v}$ in constant time (ii), and given a face τ one can access some of its half-edges in constant time as well (iii).

Algorithm 9: DEGREE

Input: vertex $\mathbf{v}$
Output: degree d of vertex $\mathbf{v}$

```
e_{h,s} ← EDGE(v), e_{h,n} ← e_{h,s}, d ← 0;
do
    e_{h,n} ← TWIN(NEXT(e_{h,n}));
    d ← d + 1;
while e_{h,n} ≠ e_{h,s};
return d;
```

face). Both are *twins* of each other.

Accessing the NEXT(e), PREVIOUS(e) or TWIN(e) *half-edge* of a given *half-edge* e and accessing its VERTEX(e) or FACE(e) requires $O(1)$ time. Each *half-edge* ends in its vertex. One can access some *half-edge* of the DCEL that ends in some given vertex $\mathbf{v}$ in $O(1)$ time. Iterating over adjacent mesh elements is supported. One example request might be to compute the degree of a given vertex $\mathbf{v}$, which is solved by Algorithm 9. Since *faces* are always to the left of their *half-edges*, one can implement the algorithm for testing whether a point $\mathbf{x}$ is contained in a convex polygon, like a 2-simplex, by testing if it is to the left of all *half-edges* of the face, see Algorithm 10.

Besides the geometric and topological information, each record of the data structure can store additional information. For example, I store the three angles of a triangle at its three half-edges. Solving the eikonal equation involves the computation of these angles. Other stored values are the approximated solution of the equation Φ or the distance to the closest obstacle d_Ω.

I refer to half-edges and vertices at the domain boundary $\partial\Omega$ as *boundary half-edges* and *boundary vertices*, respectively. Often these elements have to be treated as special cases. For example, to accelerate the test ISATBOUNDARY($\mathbf{v}$), which returns true if $\mathbf{v}$ is a *boundary vertex*, it is good practice that EDGE($\mathbf{v}$) always returns the *boundary half-edge*. In that case ISATBOUNDARY($\mathbf{v}$) requires $O(1)$ time. Any topological change adjusts the half-edge of a *boundary vertex* accordingly. Otherwise, the request would lead to an iteration similar to Algorithm 9 and Algorithm 10.

EIKMESH offers two different implementations of the DCEL. The first one is pointer-based, the second one array-based. The pointer-based implementation is straightforward and is mostly used

Algorithm 10: CONTAINS

Input: a triangle τ, a querry point $\mathbf{x}$
Output: **true** if and only if $\mathbf{x} \in \tau$

```
e_{h,s} ← EDGE(τ), e_{h,n} ← e_{h,s};
do
    e_{h,n} ← TWIN(NEXT(e_{h,n}));
    if e_{h,n} is to the right of x then
        return false;
while e_{h,n} ≠ e_{h,s};
return true;
```

for testing and debugging. The array-based implementation gives more control over memory usage and its location. It is similar to a structure of arrays (SoA) where each array contains all mesh elements of one type. Array indices replace pointers such as the pointer from a half-edge to its face. In Section 8.5, I show how this data structure can organize mesh elements in a cache-friendly manner according to a space-filling curve.

6.9 Source code

The source code for all discussed algorithms and data structures is part of the open-source simulation framework Vadere [294]. More specifically, it is contained in the `VadereMeshing` subproject. `VadereMeshing` is an independent software library that can be used without the simulation software. Class names indicate the parts of algorithms a class implements. For example, `AFace.java`, `AHalfEdge.java`, `AVertex.java`, `AMesh.java` realize the array-based doubly-connected edge list (DCEL). Walking strategies can be found in `ITryConnectify.java`, and point location algorithms in `DelaunayTree.java`, `DelaunayHierarchy.java`, `JumAndWalk.java`, and `CachedPointLocation.java`. `GenDelaunayTriangulator.java` can be used to compute a Delaunay triangulation, and `GenConstrainedDelaunayTriangulator.java` to compute a constrained or conforming Delaunay triangulation. `GenRuppertsTriangulator.java` implements Ruppert's algorithm.

6.10 Summary

In this chapter, I presented and discussed unstructured two-dimensional mesh generation – a topic that is not restricted to pedestrian dynamics, but of great importance for it.

In Section 6.1, I identified the adherence to the spatial simulation domain, high-quality elements, and localized mesh resolution as important properties the mesh should fulfill. Consequently, the meshing algorithm must pursue these contradictory goals.

The chapter also contained an introduction to well-known algorithms for unstructured mesh generation. Most of these basics are required for the upcoming discourse and are the foundation of the developed mesh generator introduced in Chapter 8. In Section 6.2, I discussed different mesh types, their properties, advantages, and disadvantages. I decided to use unstructured triangular conforming meshes for the generation of navigation fields.

There are multiple unstructured mesh generators, but they all follow similar strategies described in Section 6.3. Because of the excellent properties that the Delaunay triangulation offers and the dynamic setting I am in, EikMesh is a Delaunay-based mesh-improver for unstructured conforming two-dimensional triangular meshes.

The rest of Chapter 6 introduced all ingredients and tools needed to develop a new meshing algorithm and to analyze the quality of its output. In Section 6.4, I introduced definitions, such as the homogeneous simplicial complex, its underlying space, and the triangulation of a planar straight-line graph.

Section 6.5 introduces the required numeric computations required for Delaunay-based mesh generators: the orientation and empty-circle certificate. I further elaborated on their application.

To evaluate the performance of EikMesh (and other methods), I picked two quality measures. Section 6.6 contains their description. Furthermore, I established the connection between the largest and smallest angle of a triangle. The connection shows that a triangulation's large smallest angles lead to small largest angles, one of the desirable properties listed in the first section of this chapter.

Because unstructured meshes entail the point location problem, I explored different well-known point location algorithms in Section 6.7. Since EikMesh inserts and removes vertices during its improvement phase (see Chapter 8), the approach requires a point location strategy that supports such a dynamic setting. Therefore, I decided to use and implement the so-called JumpAndWalk algorithm. Additionally, I developed CachePointLocation, a point location algorithm that performs best if multiple point locations for the same (moving) object are required.

In Section 6.8, I briefly described the doubly-connected edge list (DCEL). It is the data structure EikMesh is based on. The DCEL offers everything EikMesh requires, from access to adjacent mesh elements in constant time to efficient iteration over all boundary edges and vertices.

In the last section, I referred to the source code of the implemented algorithms of this chapter.

CHAPTER 7

The DistMesh algorithm

> *"Between stimulus and response, there is a space. In that space is our power to choose our response. In our response lies our growth and our freedom."*
>
> – Victor Emil Frankl

Most modern meshing algorithms are complex with respect to their code and handling. Some of them even require a specific language to describe the spatial domain and support the user by a graphical user interface to simplify their usage. Therefore, their code is difficult to integrate with other codes like Vadere, and their use entails learning its specifics. In contrast to this, Persson and Strang provide DistMesh, an accessible and flexible meshing algorithm for unstructured high-quality meshes [221]. DistMesh is freely available as MATLAB code. Aside from its accessibility, DistMesh is based on a simple physical analogy. It performs very well and offers the flexibility necessary to support adaptive meshing. Most importantly, it meets many of the requirements discussed in Section 6.1. Therefore, DistMesh serves as a starting point for the development of EikMesh.

Similar to the well-known Laplacian smoothing, DistMesh improves the mesh quality in every iteration by repositioning its vertices locally. It starts with some initial triangulation $\mathcal{T}_0$. For each iteration k a new triangulation $\mathcal{T}_k$ is constructed based on the previous one. The hope is that the quality improves with k such that

$$\forall k > 0 : \rho(\mathcal{T}_k) > \rho(\mathcal{T}_{k-1}) \tag{7.1}$$

holds. One significant difference to Laplacian smoothing is that vertices move more freely. They can leave the convex hull defined by their 1-ring neighborhood. This freedom can cause connectivity changes such that the topology does not necessarily remain invariant. However, it is one crucial reason why DistMesh outperforms the Laplacian smoothing with respect to mesh qualities [221].

Aside from being a mesh improver, DistMesh supports spatial adaptive meshing. Like many other meshing algorithms, it gives the user control over the edge length by an element size function

$$h : \mathbb{R}^2 \to \mathbb{R}^+. \tag{7.2}$$

In Section 8.6, I discuss how one can construct a reasonable element size function automatically.

7.1 Geometry descriptions

The name DistMesh emphasizes one important property: it is designed to work with an implicit description of the spatial domain and its boundary given by a signed distance function d_Ω. The spatial domain Ω is defined by $\Omega = \{\mathbf{x} \mid d_\Omega(\mathbf{x}) \leq 0\}$ and its boundary by $\partial\Omega = \{\mathbf{x} \mid d_\Omega(\mathbf{x}) = 0\}$. Furthermore, $|d_\Omega(\mathbf{x})|$ is the geodesic distance to $\partial\Omega$. For example, the distance function

$$d_{\text{circ}}(\mathbf{x}) = \|\mathbf{x}\| - 1 \tag{7.3}$$

defines a circle of radius one centered at $(0, 0)$. The ability to work with distance functions gives DistMesh flexibility, because we can transform an explicit geometry representation such as a planar straight-line graph into a signed distance function. For example, let

$$\{(-1/2, -1/2), (1/2, -1/2), (1/2, 1/2), (-1/2, 1/2)\}$$

be the set of points defining a square, then

$$d_{\text{rect}}(\mathbf{x}) = d_{\text{rect}}(x, y) = \max\{|x|, |y|\} - 1/2 \tag{7.4}$$

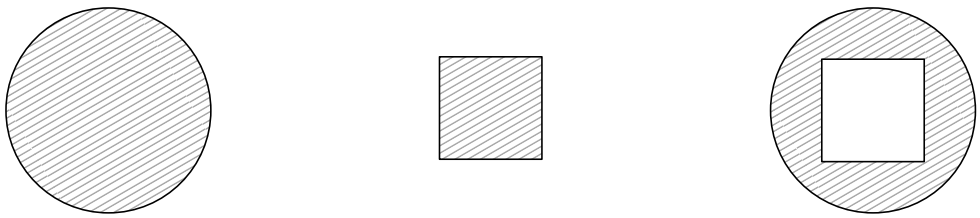
is the distance function defining the square. Furthermore, Persson and Strang described in [221] how to construct such a function by combining multiple functions each representing a geometrical object. For example,

$$d_{\text{sub}}(\mathbf{x}) = \max\{d_{\text{circ}}(\mathbf{x}), -d_{\text{rect}}(\mathbf{x})\} \tag{7.5}$$

defines the subtraction of $d_{\text{rect}}(\mathbf{x})$ from $d_{\text{circ}}(\mathbf{x})$. The result is illustrated in Fig. 7.1iii. Even though an implicit geometry representation is powerful, I show that it also imposes some limitations (see Sections 7.5 and 8.3).

7.2 The truss analogy

The best starting point for discussing DistMesh is a change in perspective. Instead of focusing on the process of mesh construction, we look at the result. Instead of asking: how do we get there? We ask: what conditions are fulfilled? To answer the second question, which leads to an



(i) d_{circ} (ii) d_{rect} (iii) d_{sub}

Figure 7.1: Definition of a spatial domain by combining multiple distance function. In this example d_{sub} (iii) defines a domain which equal is to the subtraction of the domain defined by d_{rect} (ii) from the one defined by d_{circ} (i).

answer to the first one, the authors in [221] use a physical analogy between a triangular mesh and a simplification of a simple truss structure. This analogy stems from engineering.

For engineers, a truss structure consists of so-called two-force members. For these objects, forces are applied to precisely two points. For this model, only compression and tension forces are considered leaving out the weight of truss elements and the transmitted moments. Since all shear and bending moments and other more complex stresses are approximately zero, they are neglected as well.

Engineers know that a well-designed truss structure can distribute expected external forces so that no member of the truss breaks under the exerted compression or tension. For example, a bridge's truss distributes the force caused by a heavy load to its strongest elements. Since a truss is a solid structure, each member is in a mechanical equilibrium – the net force at each node is zero.

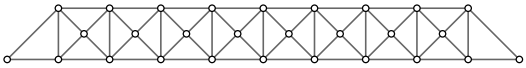


Figure 7.2: Example of a triangular truss structure of a bridge consisting of bars and nodes.

Following the analogy, a high-quality mesh represents a very stable truss structure. Persson and Strang imagined external forces of equal magnitude pulling orthogonal at the boundary of a mesh as depicted in Fig. 7.3ii. Meshes of high quality distribute the emerging pressure, compression, and tension evenly amongst all elements. On the one hand, a stable truss distributes forces evenly amongst bars. On the other hand, a high-quality mesh consists of evenly sized edges. The idea behind DistMesh is to connect both properties and to use bar forces to generate a mesh with well-sized edges iteratively.

7.3 Improving

Persson and Strang [221] achieve this connection by defining tension forces $F(\mathbf{v}_1, \mathbf{v}_2)$ and $F(\mathbf{v}_2, \mathbf{v}_1)$ for each edge $e = \{\mathbf{v}_1, \mathbf{v}_2\}$ of the mesh. Edges behave like two-force members. $F(\mathbf{v}_2, \mathbf{v}_1)$ is applied to $\mathbf{v}_1$ and $F(\mathbf{v}_1, \mathbf{v}_2)$ to $\mathbf{v}_2$, compare Fig. 7.3iii. Additionally,

$$F(\mathbf{v}_1, \mathbf{v}_2) = -F(\mathbf{v}_2, \mathbf{v}_1) \tag{7.6}$$

holds. The magnitude of these internal forces depends on the current and desired edge length, $\|e\|$ and $h(e)$, respectively. DistMesh simulates an external force that acts normal to the *boundary vertices*. This force stretches the whole mesh such that it expands towards the boundary. To keep vertices from leaving Ω, they are projected back if they cross $\partial\Omega$, compare Fig. 7.3iv. More precisely, those pushing forces $F_{\text{ext}}(\mathbf{v})$ counteract the pulling effect by acting in the opposite direction, that is, in the direction of

$$-\nabla d_\Omega(\mathbf{v}). \tag{7.7}$$

The magnitudes of F_{ext} are just large enough to keep nodes from moving outside [221]. Without F_{ext} the underlying space of the mesh $|\mathcal{T}|$ would grow indefinitely. Going back to the truss analogy, the net compression or tension of an edge $e = \{\mathbf{v}_2, \mathbf{v}_1\}$ is defined by the net forces of its

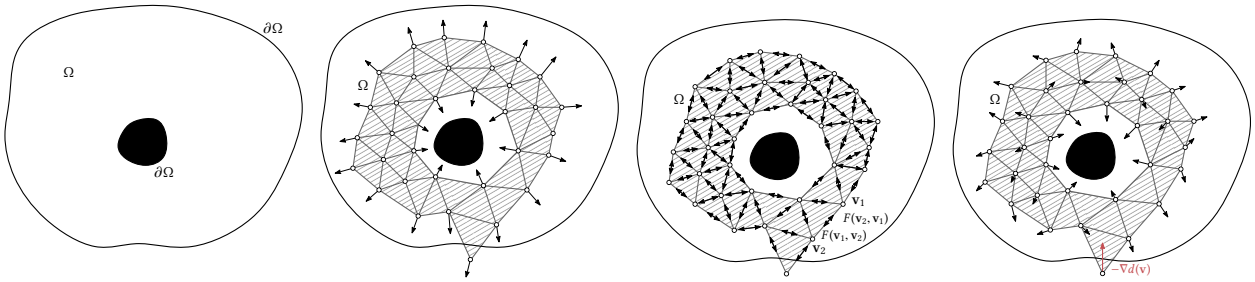


Figure 7.3: The translation of the physical analogy into the model of DistMesh for an example domain (i) and an initial mesh (ii) to (iv): external forces of the analogy (ii) are modeled by internal forces $F(\mathbf{v}_1, \mathbf{v}_2)$ (iii) and the back projection of vertices outside of Ω (iv). Internal forces (iii) applied to two-force members (edges). An improvement step of DistMesh moves vertices according to the net force at each vertex (iv).

end points $F(\mathbf{v}_1)$ and $F(\mathbf{v}_2)$. Therefore, even if $F(\mathbf{v}_1, \mathbf{v}_2)$ is a tension force, e might be either under tension or compression.

The DistMesh algorithm starts with a suboptimal initial mesh, for which the force equilibrium is violated. Over an artificial time t those forces displace mesh vertices until an equilibrium is reached. Algorithmically, the continuous time t, and with it, the continuous vertex displacement, is discretized. Consequently, DistMesh can be classified as an iterative mesh improver.

Let $\Omega \subset \mathbb{R}^2$ be the domain of interest. Furthermore, let $\mathcal{V}_k$ be the set of vertices of improvement step k and $\mathcal{E}_k$ the set of edges of its Delaunay triangulation $\mathcal{T}_k$. At any step k, forces $F(\mathcal{V}_k)$ act between the mesh points. To find an equilibrium

$$F(\mathcal{V}_k) = 0, \tag{7.8}$$

the following artificial time-dependence is introduced:

$$\frac{d\mathcal{V}}{dt} = F(\mathcal{V}), t \geq 0. \tag{7.9}$$

Any stationary solution of this ODE also satisfies Eq. (7.8). This stationary solution is found numerically using the forward Euler method

$$\mathcal{V}_{k+1} = \mathcal{V}_k + \Delta t \cdot F(\mathcal{V}_k). \tag{7.10}$$

In each Euler improvement step, points $\mathcal{V}_k$ move according to $F(\mathcal{V}_k)$. After the vertex displacement, topological changes are required to reestablish the Delaunay criterion. In fact, after large movements, the mesh may not even be a valid triangulation since points may have escaped the convex hull of their neighborhood. The empty-circle, and the orientation certificate may be violated (see Section 6.5.1). Therefore, it is essential to recompute the mesh topology. In [221], Persson and Strang suggest to recompute the Delaunay triangulation when a point moves further than a certain tolerance $\epsilon_{\text{tol}} \cdot h_0$, where h_0 is an approximation of the smallest edge length. They suggest to use $\epsilon_{\text{tol}} = 0.1$. Constructing the Delaunay triangulation for a large number of points is

computationally expensive. It requires $O(n \log(n))$ time (see Section 6.5.2) and does not exploit the fact that the topology of two consecutive meshes is similar. Since points move according to $F(\mathcal{V}_k)$, the key to a high-quality mesh lies in the definition of F, which is controlled by two user-defined functions.

The first is the already mentioned *signed distance function* d_Ω. It is the geodesic distance to the boundary of $\partial\Omega$:

$$d_\Omega(\mathbf{x}) = \text{sign}_\Omega(\mathbf{x}) \cdot \min_{\mathbf{y} \in \partial\Omega} \|\mathbf{x} - \mathbf{y}\|. \tag{7.11}$$

A negative sign indicates that the point is inside Ω, a positive sign that it is not. Of course, d_Ω depends on the geometry.

The second is a *desired edge length function*, also known as *element size function* h. In Section 8.6, I give a more detailed discussion. For now, it is enough to know that h controls the spatial mesh resolution. Let $e = (\mathbf{v}_1, \mathbf{v}_2)$ be an edge and $\mathbf{x}_e = (\mathbf{v}_1 + \mathbf{v}_2)/2$ its midpoint. For each iteration k, DistMesh transforms h into an actual desired edge length

$$h_k(e) = \psi(\mathcal{E}_k) \cdot \omega \cdot h(e), \tag{7.12}$$

which is scaled by

$$\psi(\mathcal{E}) = \left(\frac{\sum_{e \in \mathcal{E}} \|e\|^2}{\sum_{e \in \mathcal{E}} h(\mathbf{x}_e)^2} \right)^{1/2} \tag{7.13}$$

and ω. Similar to [164], I define

$$\lambda_k(e) = \frac{\|e\|}{h_k(e)} \tag{7.14}$$

to be the ratio of the edge length and the desired edge length of $e \in \mathcal{E}_k$. Clearly, we want

$$\forall e \in \mathcal{E}_k : \lambda_k(e) \approx 1. \tag{7.15}$$

The force $F(\mathbf{v}_1, \mathbf{v}_2)$ acting between $\mathbf{v}_1, \mathbf{v}_2$ is given by

$$F(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 - \mathbf{v}_2}{\|\mathbf{v}_1 - \mathbf{v}_2\|} \cdot \eta_k(e), \tag{7.16}$$

where $e = \{\mathbf{v}_1, \mathbf{v}_2\}$ and η_k is defined the following way

$$\eta_k(e) = \hat{\eta}(\lambda_k(e)) \cdot h_k(e). \tag{7.17}$$

$\eta_k(e)$ gives the magnitude of the force and $\hat{\eta}$ defines, based on $\lambda_k(e)$, the amount of tension or compression applied. Note that $\hat{\eta}$ is independent of the improvement step k. It makes sense to define $\hat{\eta}$ such that

$$\lambda_k(e) < 1 \Rightarrow \hat{\eta}(\lambda_k(e)) > 0 \ \wedge \ \lambda_k(e) \geq 1 \Rightarrow \hat{\eta}(\lambda_k(e)) \leq 0. \tag{7.18}$$

Persson and Stang suggested

$$\hat{\eta}(\lambda_k(e)) = \max\{1 - \lambda_k(e), 0\}. \tag{7.19}$$

They avoid compression for internal edge forces. Another option which includes compression is the Bosson-Heckbert smoothing function

$$\hat{\eta}(\lambda_k(e)) = (1 - \lambda_k(e)^4) \cdot \exp(-\lambda_k(e)^4) \tag{7.20}$$

suggested by [164]. Both functions are depicted in Fig. 7.4. I stick with the function proposed by Persson and Strang because its computation is cheaper, and I could not achieve better results by using the Bosson-Heckbert smoothing function. The net force acting on $\mathbf{v}_1$ is equal to the sum of forces of its incident edges

$$F(\mathbf{v}_1) = \sum_{\substack{e \in \mathcal{E} \\ e = \{\mathbf{v}_1, \mathbf{v}_2\}}} F(\mathbf{v}_1, \mathbf{v}_2). \tag{7.21}$$

Vertices crossing the boundary are projected back into Ω. They can move freely along the boundary $\partial\Omega$ but cannot distance themselves from the meshing domain Ω. If a vertex $\mathbf{v}$ ends up outside of Ω it is moved back to $\mathbf{v}^*$, defined by

$$\mathbf{v}^* = \mathbf{v} - d_\Omega(\mathbf{v}) \nabla d_\Omega(\mathbf{v}), \tag{7.22}$$

where the gradient $\nabla d_\Omega(\mathbf{v})$ is computed numerically by a finite difference scheme. Note that DistMesh assumes that $\nabla d_\Omega(\mathbf{v})$ is well-defined everywhere.

According to Strang and Persson, $F(\mathbf{v}_1, \mathbf{v}_2)$ should be larger than zero, when $\|\mathbf{v}_1 - \mathbf{v}_2\|$ is approximately equal to its desired length. This increased tension helps to spread out vertices across Ω and is achieved by choosing the scaling factor in Eq. (7.12) to be greater than 1, that is, $\omega > 1$. If ω is too small, vertices move too slowly or do not move at all. However, if it is too large, vertices tend to move back and forth indefinitely. Persson and Strang suggested to use $\omega = 1.2$, which I can confirm to work very well.

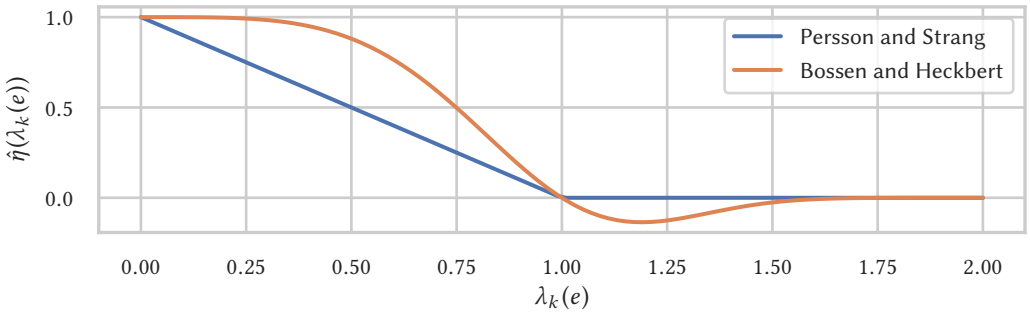


Figure 7.4: $\hat{\eta}$ chosen by Persson & Strang and Bossen & Heckbert: the first one is never negative. Thus it does not lead to (direct) compression. The The Bosson-Heckbert smoothing function leads to compression for $\lambda_k(e) > 1.0$.

7.4 Initialization

The improvement phase can only start after an initial triangular mesh $\mathcal{T}_0$ was constructed. DistMesh creates $\mathcal{T}_0$ by removing vertices $\mathbf{v}$ with a probability of

$$\frac{1}{h(\mathbf{v})^2} \tag{7.23}$$

from a uniform triangulation of side length h_0. $\mathcal{T}_0$ is the Delaunay triangulation of the remaining vertices, compare Fig. 7.7i.

7.5 Examples and discussion

I conclude the discussion of DistMesh by analyzing the result of two examples. The purpose of the first one is to strengthen the intuition of the forces-based smoothing process and to give an extensive analysis of DistMesh's performance. The second example showcases that the meshing algorithm might construct a mesh that has undesirable properties.

To evaluate performance, I use the quality measures introduced in Section 6.6. To give condensed information and to visualize the improvement progress properly, I use the kernel density estimation. For each intermediate result $\mathcal{T}_k$, I compute the normalized histogram for the two quality measures ρ_1, ρ_1 and the angles θ of each mesh element. Compare Figs. 7.7i to 7.7vi. To visually compare the whole series of histograms, I compute a series of kernel estimation distributions (KDEs) – for each intermediate result, one distribution is estimated. Therefore, I imagine ρ_1, ρ_2, and θ to be random distributions – which is not the case but gives a helpful data visualization. Each curve in Fig. 7.5 represents a KDE. The darkness of each curve encodes the improvement step k – darker colors indicate later steps.

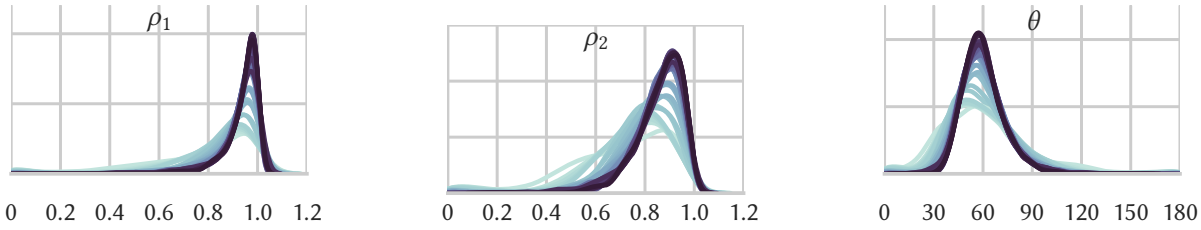
Example 1

I consider the circle with a rectangular hole defined by the distance function d_{sub} of Eq. (7.5) illustrated in Fig. 7.1. Furthermore, I use a desired edge length function

$$h_1(\mathbf{x}) = 0.05 + 0.2 \cdot |d_{\text{sub}}(\mathbf{x})|\,, \tag{7.24}$$

which depends on the distance function. Therefore, the element size h_1 increases with $-d_{\text{sub}}$, that is, with the distance to $\partial\Omega$. For this example, I use exactly 150 improvement steps. Because ∇d_{sub} is not uniquely defined everywhere, I also insert fix points at each corner of the rectangle defining the hole. An extensive discussion explaining the requirement for fix points, follows in Section 8.3. Fig. 7.7vi depicts the end and Figs. 7.7i to 7.7v intermediate results.

The element quality with respect to $\rho_1(\tau)$ and $\rho_2(\tau)$ with $\tau \in \mathcal{T}_0$ ranges from approximately zero to approximately one, compare Fig. 7.7i. Therefore, $\mathcal{T}_0$ consists of almost arbitrarily poor but also very rich elements. The improvement starts with $\rho_1(\mathcal{T}_0) = 0.78$, $\rho_2(\mathcal{T}_0) = 0.72$ and converges towards 0.93 and 0.86 respectively, compare Fig. 7.6i. The overall mesh quality jumps whenever



(i) (ii) (iii)

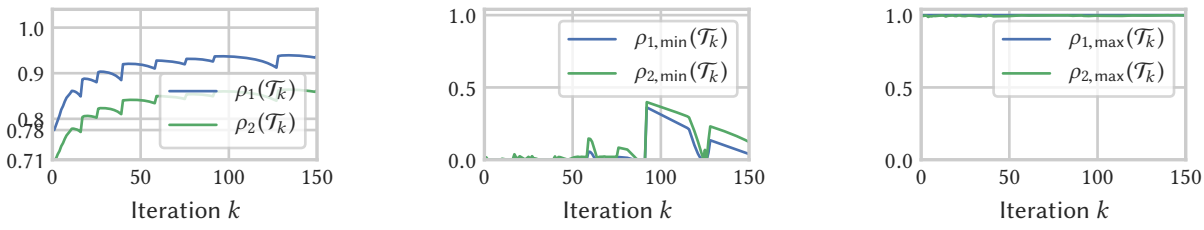
Figure 7.5: Plot of the series of estimation distributions ρ_1 (i), ρ_2 (ii) and θ (iii): each distribution is estimated using a specific mesh $\mathcal{T}_k$ with $k = 1, 2, 3, 4, 5, 10, 15, \ldots, 145, 150$. The larger k is, the darker the plotted curve becomes.

the Delaunay triangulation is recomputed and deteriorates between two consecutive computations of the Delaunay triangulation. Note that after the last improvement step has finished, the Delaunay triangulation is always recomputed.

The improvement starts with a minimal quality $\rho_{1,\min}(\mathcal{T}_0) = \rho_{2,\min}(\mathcal{T}_0) \approx 0.0$ which jumps up and down, compare Fig. 7.6ii. It never goes above 0.4. For the final mesh, the minimum is 0.04 and 0.13, respectively, compare Fig. 7.7vi. The quality of the 'best' element stays at approximately 1.0 during the whole improvement, compare Fig. 7.6iii.

The kernel density estimation of the histogram of qualities and angles of $\mathcal{T}_0$ reveals the influence of the Delaunay criterion. We can observe a peak around quality one and peaks at 30°, 60°, 90°, and 120°, compare Fig. 7.5. These peaks transform into a single growing peak at a quality equal to 1.0 and angle equal to 60°. As desired, the quality of each element tends to 1.0, and its angles tend to 60°.

One can observe that some of the initially best elements first drop in quality such that others, very poorly shaped triangles, can be improved. At first, the element quality gets more evenly distributed. Then, after each re-computation of the Delaunay triangulation, it is increased for all non-boundary elements. $\rho_1(\mathcal{T}_k)$ and $\rho_2(\mathcal{T}_k)$ increase with k, and their distributions move towards one.



(i) (ii) (iii)

Figure 7.6: Plots of the mean quality $\rho_1(\mathcal{T}_k)$, $\rho_2(\mathcal{T}_k)$ (i) minimal quality $\rho_{1,\min}(\mathcal{T}_k)$, $\rho_{2,\min}(\mathcal{T}_k)$ (ii) and maximum quality $\rho_{1,\max}(\mathcal{T}_k)$, $\rho_{2,\max}(\mathcal{T}_k)$ (iii): the mean quality ρ_1 approaches 0.93 while the minimum quality stays low and does never go above 0.4.

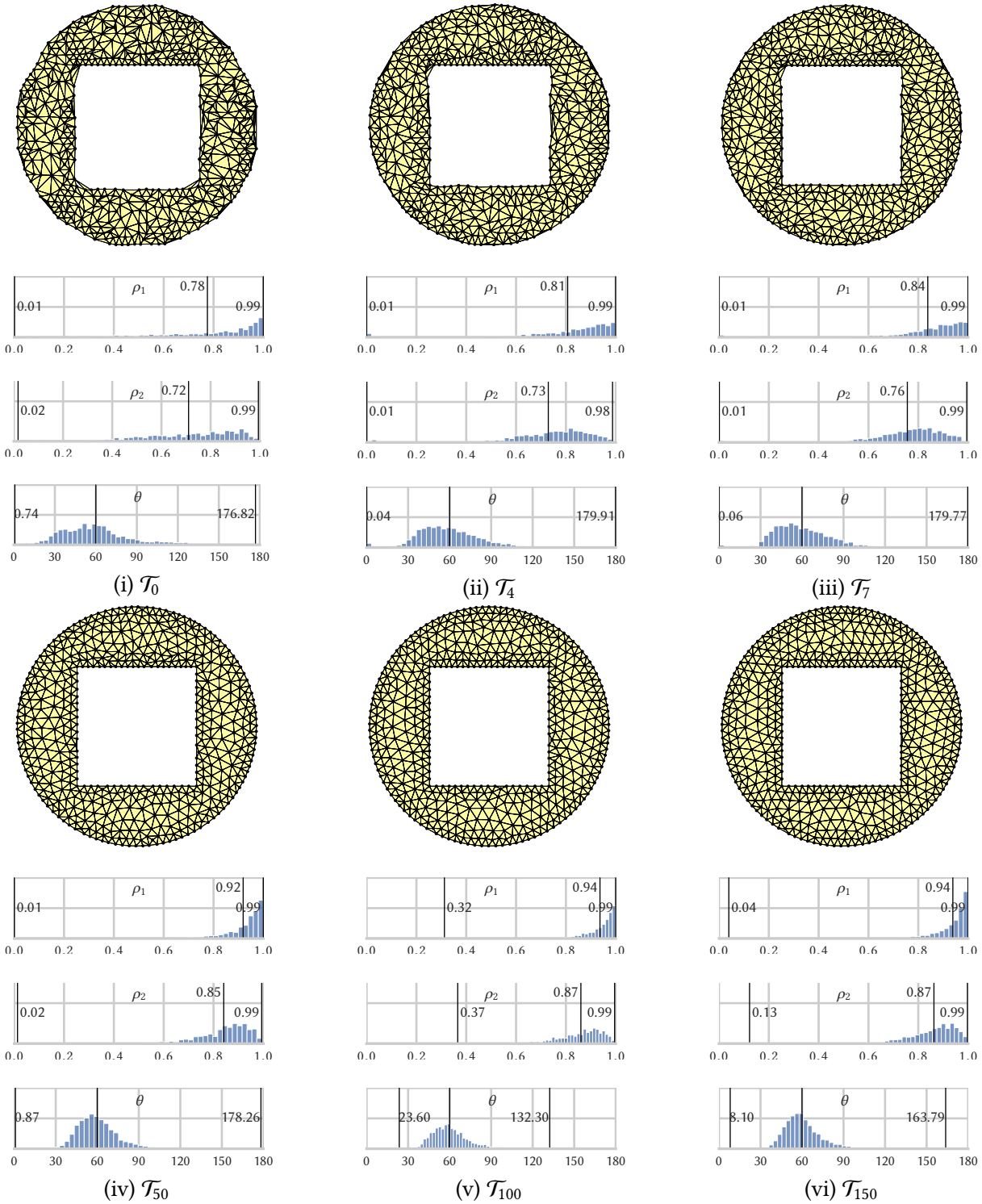
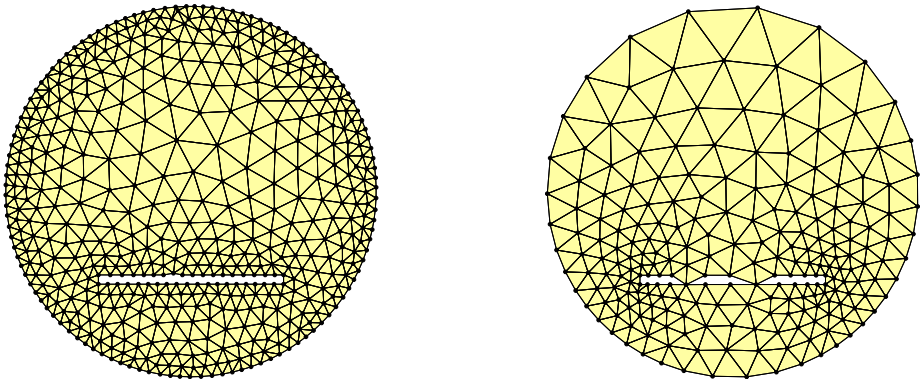


Figure 7.7: Intermediate results of the DistMesh algorithm: for iteration $k = 0, 4, 7, 50, 100, 150$ a histogram of qualities ρ_1, ρ_2 and the angles θ of each triangle of $\mathcal{T}_k$ is displayed. The algorithm starts with a Delaunay triangulation of randomly selected points (i). Points move around such that qualities are first harmonized (ii)-(iii) and then increase for all internal triangles (iv)-(vi). Minimum, maximum and mean values are highlighted by a vertical black line.



(i) (ii)

Figure 7.8: Two undesirable results constructed by DistMesh: by using h_1 (i), the constructed mesh seems to represent the geometry well but it does not fully adhere to the boundary at the top center of the hole. By using a slightly different but reasonable element size function h_2 (ii), the constructed mesh does no longer represent the geometry accurately.

However, the minimal quality plot reveals the existence of some poorly shaped outliers. These low-quality elements can be found near the boundary domain $\partial\Omega$, compare Figs. 7.7ii to 7.7vi. In Section 8.4, I explain that unwanted local force equilibriums are responsible for these low quality elements. Additionally, I show different strategies to fix the issue.

Example 2

For the second example, I replace the rectangle hole with another narrower rectangle of height 0.05. If one uses the edge length function from the previous example, everything seems to work. But if we look closely, the mesh does not entirely adhere to the boundary. Compare the narrow rectangle of Fig. 7.8i, especially at the top center.

The problem gets revealed if we replace h_1 with another element size function

$$h_2(\mathbf{x}) = 0.05 + 0.2 \cdot d_{\text{corner}}(\mathbf{x}), \tag{7.25}$$

that defines a high resolution imposed by the geometry (see Section 8.6). In Eq. (7.25), $d_{\text{corner}}(\mathbf{x})$ is the distance to the closest corner of the rectangle. h_2 should lead to a high-quality mesh because it defines a small element size where needed, that is, at positions close to the left and right vertical line of the rectangle. The problem is that triangles might intersect with the rectangle, but their midpoint is not contained in it. Therefore, these triangles will not be removed. Without an explicit representation of the rectangle, testing for intersection is difficult. One could evaluate some sample points but choosing the right one seems to be hard. And even if an explicit representation is available, testing for intersection is still much more expensive than evaluating d_Ω one single time. One could use another element size function to ensure that elements close to the boundary are small enough. But this could lead to an unacceptable amount of elements.

In Section 8.3.2, I present a simple solution by including explicit defined geometrical objects into the meshing process.

7.6 Source code

There are existing DistMesh implementations for different programming languages. The original algorithm is freely available at

http://persson.berkeley.edu/distmesh/.

An alternative Python implementation is available at

https://github.com/bfroehle/pydistmesh

and my own Java implementation is contained in `Distmesh.java` file of the `VadereMeshing` subproject that is part of open-source simulation framework Vadere [294].

7.7 Summary

In this chapter, I gave a detailed description of DistMesh, the meshing algorithm I adapted and extended. Therefore, I lay the ground for the upcoming chapter, in which I finally describe EikMesh.

At the beginning of this chapter, I explained why I looked at DistMesh to design a proper meshing algorithm. Apart from its accessibility, DistMesh is a useful mesh-improver, especially for the dynamic setting of my application. Secondly, DistMesh is known to generate high-quality meshes and outperforms many other mesh-improvers, such as the Laplacian smoothing.

In Section 7.1, I explained the difference between an implicit and explicit representation of the spatial domain Ω – a difference which is important because, as I showed, DistMesh relies on the gradient ∇d_Ω of an implicit representation. Later on, I explain how explicitly defined geometric objects can improve the mesh generation process (see Section 8.3). I also explained how one could combine distance functions of simple geometries to construct distance functions of more complex geometries.

To deepen my own and the reader's understanding, I shed light on the connection between the physical analogy of a truss structure and the DistMesh algorithm (see Section 7.2). This understanding led me to some algorithmic improvements that I introduce in the next chapter.

After describing the analogy, I looked into DistMesh in detail, including its improvement phase (Section 7.3) and its probabilistic initialization phase (Section 7.4).

By using two examples (see Section 7.5), I highlighted the strength of DistMesh, that is, the generation of high-quality meshes based on an implicit domain representation. But I also discussed its weaknesses: an arbitrarily small minimal quality, an unnecessarily restricted element size function h, and elements that might not adhere to the boundary. I showed that the mesh quality decreases between consecutive Delaunay triangulation computations. Furthermore, I established a method to visualize the change in the quality and angle distribution with respect to the improvement step k, which can be used to analyze other mesh improvement methods.

CHAPTER 8

The EikMesh algorithm

> *"The kind of freedom you will not hear much talked about in the great outside world of winning and achieving and displaying involves attention and awareness and discipline, and effort and being able truly to care about other people and to sacrifice for them [...]. That is real freedom. That is being educated."*
>
> – David Forster Wallace

In this chapter, I describe EikMesh in full detail. I introduce all adaptations of the DistMesh algorithm to enhance its usability in the context of microscopic pedestrian simulation. Its core, the forced-based vertex displacement, remains untouched while its initialization phase is improved and additional topological operators are added. By using a new initialization phase, the smoothing starts with a high-quality mesh. I show that in combination with a *non-recursive flipping algorithm*, inspired by FlipAll, EikMesh performs better than DistMesh for important examples. Since I get rid of the computation of Delaunay triangulations, I increase parallelism. The *non-recursive flipping algorithm* FlipEdges establishes the mesh topology after each point displacement. Special local topological changes improve the quality of boundary elements. I also show how a *constrained* version of EikMesh is advantageous for geometries represented by planar straight-line graphs (PSLGs). Afterwards, I discuss the local feature size function h, its importance, influence, and automatic computation. Finally, I show how each adaptation leads to parallelism, a property that can be exploited to accelerate EikMesh by executing the improvement phase on single instruction multiple data (SIMD) hardware architectures. In Section 8.8, I give results and comparisons between EikMesh and DistMesh.

The original MATLAB implementation [221] of DistMesh is only suitable for simple geometries. Reasonable element size functions h have to be provided by the user, and the execution time becomes intolerable for complicated geometries. The reason for this is the computation of multiple Delaunay triangulations and the frequent evaluation of d_Ω. Another problem, pointed out by Koko [164], is that boundary vertices may, in fact, not align with the boundary and that element qualities can drop suddenly at any point in the smoothing phase. See the plot in Fig. 7.6ii of Chapter 7.

EikMesh is an extension of DistMesh. It implements the same force-based vertex displacement strategy, takes special care of boundary elements, and avoids all computations of the De-

launay triangulation. It relies entirely on local mesh operations that benefit parallelism and fast memory-efficient data access. In summary, EikMesh improves DistMesh in three major areas:

(1) **robustness**: while meshes generated by DistMesh have an overall high quality the minimal element quality $\rho_{\min}$ is often small. EikMesh resolves this problem,

(2) **alignment**: EikMesh, other than DistMesh, guarantees that boundary vertices are located at the boundary of the spatial domain Ω,

(3) **computational cost**: compared to DistMesh, EikMesh requires reasonable computation costs even for large scale meshes. In addition, EikMesh scales well.

Additionally, EikMesh is implemented based on the doubly-connected edge list (DCEL) described in Section 6.8. Mesh elements are stored in a cache-friendly manner, and the DCEL allows the exploitation of the actual mesh topology. Therefore, the DCEL supports local operations required for the special treatment of boundary elements. A new initialization phase, see Section 8.5, distributes mesh points deterministically according to h. Except for boundary elements, the initial constructed mesh $\mathcal{T}_0$ consists of well-shaped triangles. The last two ingredients I introduce is a strategy to automatically compute an element size function h and an approximation of distance function d_Ω for any given planar straight-line graph.

8.1 Local operations

Before discussing EikMesh in more detail, I give an overview of all *local mesh operations*. These operations implement all topological changes required by the algorithm. *Local* means that each operation only changes a *small number* of *neighboring* mesh elements. Because EikMesh computes the topology of $\mathcal{T}_{k+1}$ based on the topology of the previously generated mesh $\mathcal{T}_k$, all intermediate results have to be homogeneous simplicial complexes. Therefore, each local operation has to be *valid*.

Definition 8.1 (valid mesh operation)**.** Let $\mathcal{T}_k$ be a homogeneous simplicial complex. Then a (local) mesh operation is *valid* if and only if the change it causes leads to a homogeneous simplicial complex.

The most essential local operation, called FlipEdge(e), changes the connectivity of neighboring triangles of an edge e, compare Fig. 8.1iii. A flip is *legal* if and only if an edge is *locally Delaunay* after its execution. Inserting new points into a given triangulation is done by SplitFace($\tau, \mathbf{x}$). It inserts $\mathbf{x}$ into $\mathcal{T}$ by splitting τ into three new triangles τ_1, τ_2, τ_3 such that

$$|\tau| = |\tau_1| \cup |\tau_2| \cup |\tau_3|. \tag{8.1}$$

The operation assumes that $\mathbf{x}$ IsContained($\tau, \mathbf{x}$) in τ, compare Fig. 8.1i. The last obvious basic operation, Move($\mathbf{v}, \Delta\mathbf{x}$) is almost completely adopted from DistMesh [221], compare Fig. 8.1ii. An additional validity test ensures that vertex displacements are *valid*. The following additional local operations realize the special treatment of *boundary elements*, see Sections 8.2.2 and 8.4:

(i) RemoveFace(τ): removes a boundary triangle,

(ii) CreateFace($\mathbf{v}$): assumes that the angle $\alpha_\mathbf{v}$ at $\mathbf{v}$ is acute and creates a new boundary triangle by connecting the two boundary neighbors of $\mathbf{v}$ by an edge, compare Fig. 8.2ii,

(iii) CollapseVertex($\mathbf{v}$): assumes that $\mathbf{v}$ is a three degree boundary vertex, It removes $\mathbf{v}$ and merges the two neighboring triangles,

(iv) SplitEdge(e): splits an edge e into two new edges by inserting a vertex at its midpoint $\mathbf{x}_e$,

(v) CreateHole(τ): converts τ into a hole, i. e., a face that is not part of $|\mathcal{T}|$,

(vi) CollapseEdge(e): removes an edge e by merging its endpoints into its midpoint $\mathbf{x}_e$. Its endpoints are removed. All former neighbors of those endpoints are connected to $\mathbf{x}_e$,

(vii) CollapseFace(τ): combines RemoveFace and Move to remove a triangle $\tau \in \mathcal{T}$ without changing the underlying space $|\mathcal{T}|$. First RemoveFace(τ) is executed. Then Move($\mathbf{v}, \mathbf{x}_e - \mathbf{v}$) moves the vertex $\mathbf{v}$ opposite from the former boundary edge e to the midpoint $\mathbf{x}_e$ of e.

All these operations are depicted in Fig. 8.2.

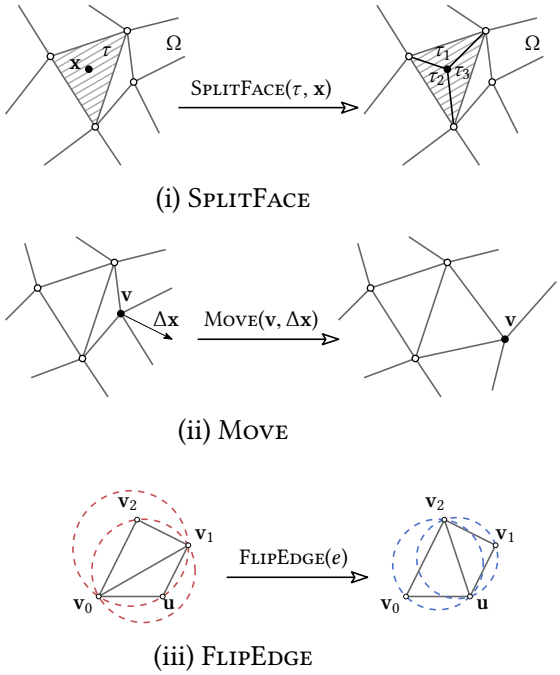
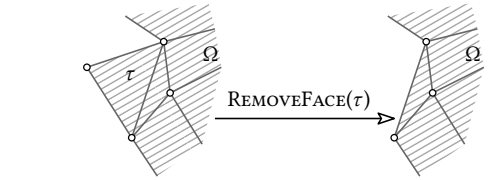
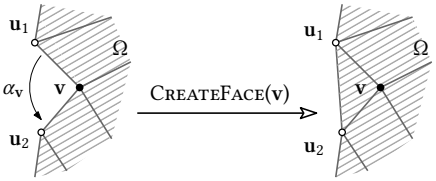


Figure 8.1: Basic local mesh operations.

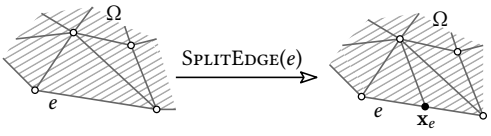


(i) RemoveFace

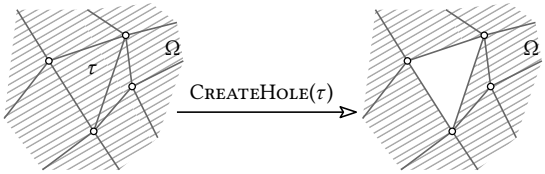


(ii) CreateFace

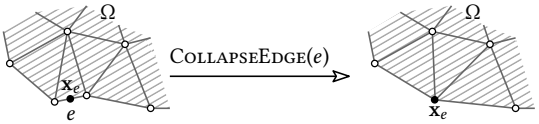
(iii) CollapseVertex



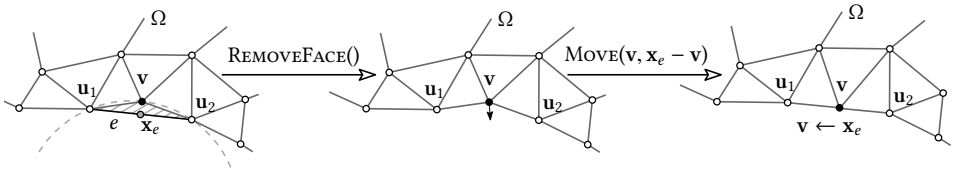
(iv) SplitEdge



(v) CreateHole



(vi) CollapseEdge



(vii) CollapseFace

Figure 8.2: Special local mesh operations.

8.2 Non-recursive flips

In this section, I describe the *non-recursive flipping algorithm*, which strives for a Delaunay triangulation without computing it from scratch. The method demands a valid mesh at all times. I explain how the method avoids any illegal point displacements.

Algorithm 11: FlipEdges

Input: triangulation $\mathcal{T}$
Output: triangulation $\mathcal{T}$

```
for e edge in T_k do
    if e is not Delaunay then
        T ← FlipEdge(e);
return T;
```

While analyzing DistMesh, one observes the following: during its improvement phase, the topology of consecutively generated meshes $\mathcal{T}_k, \mathcal{T}_{k+1}$ is very similar, especially for large k. The plot displayed in Fig. 8.3 underpins this observation. Additionally, the mesh quality $\rho(\mathcal{T}_k)$ contains jumps, and even for simple geometries, the minimal quality $\rho_{\min}(\mathcal{T}_k)$ can be arbitrarily low for all k. This phenomenon is described in Section 7.5 and presented by Koko [164].

Instead of computing the Delaunay triangulation from scratch, I use the edge flip method adapted from Lawson [181]: for each iteration, an edge in $\mathcal{E}_k$ is flipped if it is not *locally Delaunay*, i. e., IsInCircle is true. Flipping until all edges are locally Delaunay requires $O(|\mathcal{E}_k|^2) = O(n^2)$ time. However, if only a few topological changes are necessary, which is the case if $\rho(\mathcal{T})$ is high, very little work has to be done. When analyzing multiple experiments, it turned out that flipping each edge at most once for every iteration step (Algorithm 11), was enough to establish a Delaunay triangulation for the final mesh. As a consequence, the theoretical time complexity for each iteration step of the improvement phase is reduced from $O(n \log(n))$ to $O(n)$ compared to DistMesh. There is no guarantee that the final mesh is, in fact, a Delaunay triangulation. How-

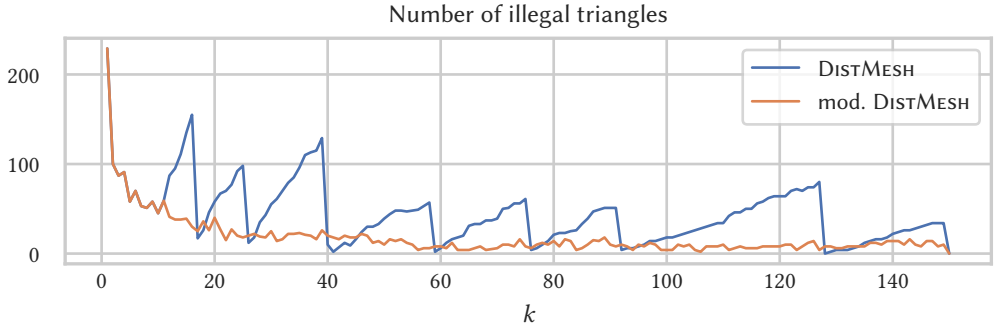


Figure 8.3: Number of non-Delaunay triangles after each point displacement of DistMesh (blue) and a modified version (orange) for which the Delaunay triangulation is computed for each improvement step. I use the first example of Section 7.5

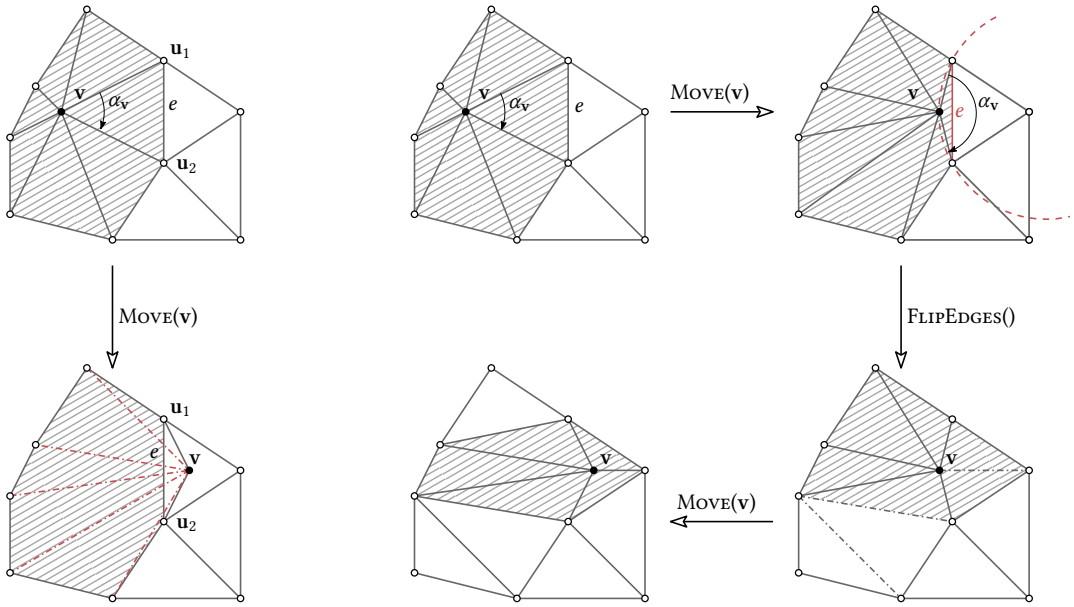


Figure 8.4: An illegal (i) and legal (ii) vertex displacement: the resulting mesh of an illegal vertex displacement is not a (valid) simplicial complex. If a vertex gets close to an edge e, it will be flipped eventually.

ever, since the algorithm requires multiple iterations, non-Delaunay edges are flipped eventually. I observed that the final result is a Delaunay or constrained Delaunay triangulation for all test cases. As a side remark, the number of required flips depends on the initial triangulation $\mathcal{T}_0$ and the element size function h, i. e., the distance vertices move to get a "good" position.

Non-recursive flips FlipEdges introduces an important restriction, because we no longer reconstruct the complete mesh topology. Therefore, after each mesh operation, the mesh has to be a (valid) triangulation. If a vertex moves outside of the convex hull of its 1-ring neighborhood, FlipEdges does not necessarily suffice to reestablish validity. Since EikMesh flips edges after they are displaced by calling MoveVertices (Line 5 in Algorithm 12), the resulting mesh has to be valid. However, this is not necessarily guaranteed, compare Fig. 8.4. FlipEdge, IsInCircle, IsInside, and many more operations lead to unexpected results if they operate on an invalid mesh. Since IsInside, IsLeft, and IsRight might return false values, the point location might also fail. Therefore, almost every operation involving the mesh ends unexpectedly if the orientation certificate is invalid. DistMesh does not run into this problem because the topology computation does not rely on the previously constructed mesh. As a consequence, EikMesh has to avoid any illegal vertex displacement. Two possible illegal displacements may occur: an edge crossing vertex and a boundary collision. Below I describe how I deal with them.

8.2.1 The edge crossing vertex

The first kind of illegal displacement, which might occur, is a vertex that crosses the convex hull of its 1-ring, as depicted in Fig. 8.4i. In the following, I argue that this can not happen if Δt is

sufficiently small.

Let us assume that Δt is sufficiently small and let $e = \{\mathbf{u}_1, \mathbf{u}_2\}$ be the edge it crosses. Furthermore, let $\alpha_\mathbf{v}$ be the angle at $\mathbf{v}$ of the triangle $\mathbf{u}_1\mathbf{v}\mathbf{u}_2$ and $d(\mathbf{v}, e)$ be the distance between $\mathbf{v}$ and e. At some point, $\mathbf{v}$ has to be very close to e. For this situation, let us look at the circumcenter radius of the triangle $\mathbf{u}_1\mathbf{v}\mathbf{u}_2$. There are two cases depicted in Fig. 8.5.

Large circumcircle radius

If $\mathbf{v}$ does not move towards a vertex of e, we get

$$\alpha_\mathbf{v} \to \pi \text{ for } d(\mathbf{v}, e) \to 0. \tag{8.2}$$

It implies that the circumradius of the triangle $\mathbf{u}_1\mathbf{v}\mathbf{u}_2$ becomes infinitely large. If e is not at the boundary, the circumradius will eventually contain some other vertex, and e will be flipped by FlipEdges before it crosses e, compare Fig. 8.4ii. I show how one deals with the boundary case later on.

Small circumcircle radius

Let us look at the other case, for which $\mathbf{v}$ moves towards a vertex $\mathbf{u}_1$ of e. The force

$$F(\mathbf{u}_1, \mathbf{v}) = \frac{\mathbf{u}_1 - \mathbf{v}}{\|\mathbf{u}_1 - \mathbf{v}\|} \cdot f_k(\mathbf{u}_1, \mathbf{v}) \tag{8.3}$$

pushes $\mathbf{v}$ away from $\mathbf{u}_1$. Since

$$\|\mathbf{v} - \mathbf{u}_1\| \to 0, \tag{8.4}$$

the magnitude of $F(\mathbf{u}_1, \mathbf{v})$ approaches its maximum

$$\hat{\eta}_k(0) \cdot h_k(\mathbf{v}) = 1 \cdot h_k\mathbf{v}) = h_k(\mathbf{v}) \tag{8.5}$$

defined by Eq. (7.17), while all other forces acting on $\mathbf{v}$ become weaker. For a reasonable element size function h, $F(\mathbf{u}_1, \mathbf{v})$ should become strong enough to prevent $\mathbf{v}$ from collapsing into $\mathbf{u}_1$. This is not a formal proof, but it explains why such a collapse never occurs for all my examples.

In summary, vertices distant from the boundary, i. e., their one 1-ring neighborhood consists of non-boundary vertices, do not cross any edge if Δt is sufficiently small. Because for tiny Δt,

(i) (ii)

Figure 8.5: The two cases of an edge crossing vertex: the vertex does not move towards $\mathbf{u}_1$ or $\mathbf{u}_2$ (i). In that case, a large circumradius eventually leads to an edge flip. In the second case, the vertex moves towards $\mathbf{u}_1$ (ii). This leads to a strong repulsive force between $\mathbf{u}_1$ and $\mathbf{v}$, preventing $\mathbf{v}$ from collapsing into $\mathbf{u}_1$.

Algorithm 12: EikMeshSmoothing

Input: initial triangulation $\mathcal{T}_0$, element size function h, distance function d_Ω
Output: a high-quality triangulation $\mathcal{T}_k$

$k \leftarrow 0$;
do
  $\psi_k \leftarrow$ compute $\psi(\mathcal{E}_k)$ **in parallel**;
  $F(\mathcal{V}_k) \leftarrow$ ComputeVertexForces($\mathcal{T}_{k+1}, h, d_\Omega, \psi_k$) **in parallel**;
  $\mathcal{T}_{k+1} \leftarrow$ MoveVertices($\mathcal{T}_{k+1}, F(\mathcal{V}_k)$) **in parallel**;
  $\mathcal{T}_{k+1} \leftarrow$ UpdateBoundaryElements($\mathcal{T}_{k+1}, h, d_\Omega, F(\mathcal{V}_k)$) **in parallel**;
  $\mathcal{T}_{k+1} \leftarrow$ FlipEdges($\mathcal{T}_k$) **in parallel**;
  $k \leftarrow k + 1$;
while $F(\mathcal{V}_k) \approx \mathbf{0}$;
return $\mathcal{T}_k$;

the convergence rate decreases, Δt should not be arbitrarily small, and vertices might be pushed outside their surrounding convex hull. However, since

$$\lim_{k\to\infty} F(\mathcal{V}_k) \to \mathbf{0} \tag{8.6}$$

and the mesh quality $\rho(\mathcal{T}_k)$ increases with k, such illegal displacements only appear in practice for early improvement steps, i. e., for small k. Remember, if the quality of each simplex τ neighboring a vertex is high, the vertex is distant from any opposite edge and, therefore, close to the center of the convex hull of all neighboring vertices, compare Fig. 8.7.

To guarantee legal vertex movements, EikMesh halves the magnitude of $F(\mathbf{v})$ until the displacement of $\mathbf{v}$ becomes legal.

8.2.2 Boundary collision

The second kind of illegal displacement can occur at the mesh boundary. Since flips can never be executed for a boundary edge, the mesh boundary might collapse into itself. Such a situation is illustrated in Fig. 8.6. In that case, the mesh $\mathcal{T}$ does not represent a homogeneous simplicial complex (Definition 6.5) any longer because there are overlapping triangles.

Detecting these overlaps is computationally expensive. It would drain a lot of resources such that the run time would become unacceptable. Furthermore, even if we found these illegal simplexes, fixing the mesh would be complicated. We could remove and re-insert all vertices close to the overlapping region or re-compute the whole Delaunay triangulation $\mathcal{DT}(\mathcal{V}_k)$. I want to avoid this computation, therefore, EikMesh makes sure that the boundary never collapses into itself.

For this reason, it implements the following smoothing technique: let us assume $\mathbf{v}$ is a boundary vertex and $\mathbf{u}_1, \mathbf{u}_2$ are its adjacent boundary vertices. Then a new edge $\{\mathbf{u}_1, \mathbf{u}_2\}$ is introduced by CreateFace if

(i) the angle $\sphericalangle \mathbf{u}_1\mathbf{v}\mathbf{u}_2 = \alpha_\mathbf{v}$ is acute, that is, if $\alpha_\mathbf{v} < \frac{1}{2}\pi$, and

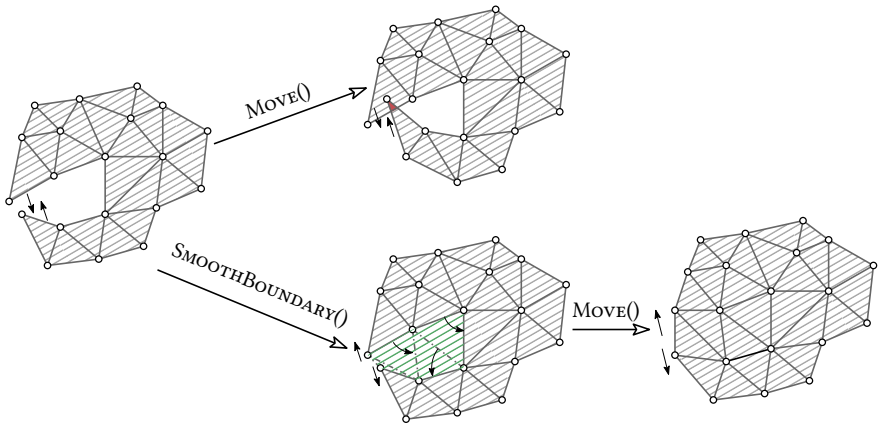


Figure 8.6: The smoothing of unwanted acute boundary angles: by eliminating these angles, EikMesh prevents the boundary from collapsing into itself.

(ii) the midpoint $\mathbf{x}_\tau$ of the triangle $\mathbf{u}_1\mathbf{v}\mathbf{u}_2$ is inside the domain Ω ($d_\Omega(\mathbf{x}_\tau) < 0$).

By this additional step within each improvement iteration, EikMesh prevents boundary collisions because it eliminates unwanted acute angles at the boundary by introducing new edges and triangles, as depicted in Fig. 8.6.

8.3 Boundary adherence

One of the most significant challenges in mesh generation is the treatment of the domain boundary $\partial\Omega$. In 2-d, generating a regular triangulation containing only equilateral triangles is trivial if one ignores the boundary, compare Fig. 8.7. The problem gets delicate because of the restrictions the boundary imposes. I already discussed how different approaches deal with the challenge. For example, advancing-front methods start the mesh generation at the boundary to profit from 'the freedom of choice' to insert new Steiner vertices. Delaunay-based approaches, on the other hand, start with a constrained Delaunay triangulation, which perfectly aligns with the boundary if a planar straight-line graph defines it. Both strategies use an explicit representation of the geometry.

DistMesh, on the other hand, is based on an implicit representation of Ω, realized by a signed distance function. It does not make use of explicit geometric information. Instead, it achieves alignment by a combination of back projection, using ∇d_Ω, and the deletion of triangles, which are outside of Ω. In general, the underlying space of an initial mesh $\mathcal{T}_0$ is unequal to the domain, i. e., $|\mathcal{T}_0| \neq \Omega$. The hope is that it changes with respect to d_Ω until $|\mathcal{T}_k| = \Omega$ holds. For many

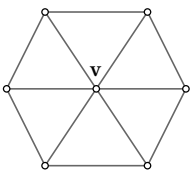


Figure 8.7: A vertex $\mathbf{v}$ surrounded by high-quality triangles.

examples, this treatment of the boundary results in high-quality meshes which adhere to the boundary. However, this is not true for all tested geometries [164]. Sometimes it is also handy to ensure that specific line-segments (constraints that are not part of the boundary) are present in the final result.

In this section, I discuss why the gradient dependent projection can lead to a mesh, that does not adhere to $\partial\Omega$ and how we can solve the issue. Besides, I show how EikMesh uses the additional information given by an explicit representation of the geometry to improve the mesh construction.

8.3.1 Fix points

Let us assume a domain Ω defined by the following distance function:

$$d_\Omega(\mathbf{x}) = d(x_1, x_2) = |x_1| - |x_2|. \tag{8.7}$$

$d_\Omega(\mathbf{0}) = 0$, therefore, $\mathbf{0}$ is part of the domain boundary $\partial\Omega$. However, the gradient ∇d_Ω at $\mathbf{0}$ is not uniquely defined, compare Fig. 8.8. In fact, there are infinitely many subgradients. The two limiting ones are

$$\nabla d^+_\Omega(\mathbf{0}) = \begin{pmatrix} 1/\sqrt{2} \\ -1\sqrt{2} \end{pmatrix}, \text{ and } \nabla d^-_\Omega(\mathbf{0}) = \begin{pmatrix} -1/\sqrt{2} \\ -1\sqrt{2} \end{pmatrix}. \tag{8.8}$$

The mesh $\mathcal{T}$ only aligns with the boundary if $\mathbf{0} \in \mathcal{T}$, but projecting points like $\mathbf{v}$ towards $\partial\Omega$ using either $\nabla d^+_\Omega(\mathbf{0})$ or $\nabla d^-_\Omega(\mathbf{0})$, will most certainly not suffice. It leads to $d_\Omega(\mathbf{v}) = 0$, but there is no guaranteed force pressuring $\mathbf{v}$ towards $\mathbf{0}$, and even if there is one, $\mathbf{v}$ might converge very slowly towards $\mathbf{0}$.

Instead of using ∇d_Ω, one should use $(\mathbf{0} - \mathbf{v})$ for the projection. Consequently, EikMesh inserts a fix point $\mathbf{v}_f = \mathbf{0}$. Fix points are part of the mesh and will never be displaced or removed. Inserting them requires additional explicit knowledge about the given geometry. However, to guarantee perfect alignment, some explicit geometrical information is needed. To guarantee perfect adherence, fix points have to be inserted at all positions $\mathbf{x}$, for which $d_\Omega(\mathbf{x}) = 0$ and $\nabla d_\Omega(\mathbf{x})$ is not uniquely defined.

8.3.2 Geometric constraints

So far the input for EikMesh is identical to DistMesh, that is,

Figure 8.8: A problematic boundary which is difficult to adhere to since $\nabla d_\Omega(\mathbf{0})$ is not uniquely defined (i). A fix point inserted at $\mathbf{0}$ resolves the issue (ii).

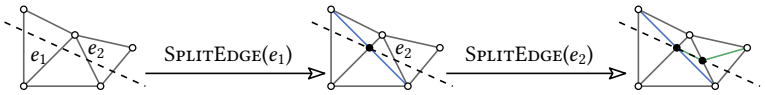


Figure 8.9: Edges of the initial mesh are split if they intersect with a constrained line-segment.

(1) a distance function d_Ω,

(2) an element size / edge length function h and

(3) an optional set of fix points.

In this section, I add another component to the input: a segment-bounded planar straight-line graph $\mathcal{P}$. Furthermore, I describe how EikMesh uses $\mathcal{P}$ to enhance the meshing process.

First, let us illustrate the difference between both geometrical representations given by d_Ω and $\mathcal{P}$, respectively. For example, let Ω be a square

$$S_{\text{in}} = \{(-1/2, -1/2), (1/2, -1/2), (1/2, 1/2), (-1/2, 1/2)\} \tag{8.9}$$

subtracted by another square

$$S_{\text{out}} = \{(-1, -1), (1, -1), (1, 1), (-1, 1)\}. \tag{8.10}$$

Its explicit representation is given by S_{out} and S_{in}, while its implicit representation is given by d_Ω defined in the following way:

$$\begin{aligned} d_{\text{out}}(\mathbf{x}) = d_{\text{out}}(x_1, x_2) &= \max\{|x_1|, |x_2|\} - 1, \\ d_{\text{in}}(\mathbf{x}) = d_{\text{in}}(x_1, x_2) &= \max\{|x_1|, |x_2|\} - 1/2, \\ d_\Omega(\mathbf{x}) &= \max\{d_{\text{out}}(\mathbf{x}), -d_{\text{in}}(\mathbf{x})\}. \end{aligned} \tag{8.11}$$

Each of the 8 line-segments of $\mathcal{P}$, such as $l = \{(-1/2, -1/2), (1/2, -1/2)\}$, are treated like *constraints* of a conforming Delaunay triangulation.

Let $\mathcal{L}$ be the set of all constrained line-segments. Similar to DistMesh, EikMesh starts by constructing an initial triangulation. Afterwards, it inserts each given constrained line-segment $l \in \mathcal{L}$ by the following strategy: first, both end points $\mathbf{p}_1, \mathbf{p}_2$ of l are inserted and marked as fix points. Then all triangles that intersect l are gathered by using a *straight triangle walk* from $\mathbf{p}_1$ to $\mathbf{p}_2$, see Section 6.7. EikMesh splits each edge of the gathered triangles intersecting l at the respective intersection point. After this process, l is represented by *constrained* edges – these edges will never be flipped.

This line-segment insertion is very similar to constructing a conforming Delaunay triangulation discussed in Section 6.5.3, but EikMesh omits all edge flips because non-Delaunay edges are flipped eventually during the improvement phase. Each vertex that is part of a constraint lies on a line-segment l and is marked as *slide point* of l.

Definition 8.2 (slide point of a line-segment)**.** Let l be a line-segment inserted into a mesh $\mathcal{T}$. Then a vertex $\mathbf{v}_l \in \mathcal{V}$ is a *slide point* of l if and only if $\mathbf{v}_l \in |l|$.

A *slide point* $\mathbf{v}_l$ is also a *fix point* if it is an endpoint of l. Essentially, $\mathbf{v}_l$ is only allowed to move on l. This is realized by projecting its net force $F(\mathbf{v}_l)$ onto l, compare Fig. 8.11.

Whenever EikMesh splits a constrained edge of $\mathcal{T}$ executing SplitEdge(e), the newly introduced vertex becomes a *slide point*. Different from *fix points*, EikMesh might remove *slide points* during the improvement phase.

To provide a reasonable performance, I maintain a mapping $\ell : \mathcal{V} \to \mathcal{L}$ to access the specific line-segment $l = \ell(\mathbf{v}_l)$ for any *slide point* $\mathbf{v}_l$ in $O(1)$ time.

If the geometry is completely defined by $\mathcal{P}$, *slide points* replace the projection method, which relies on ∇d_Ω. Additionally, EikMesh can deal with a geometry that is fully defined by d_Ω and only partly defined by $\mathcal{P}$. For example, let us change d_{in} to

$$d_{\text{in}}(x_1, x_2) = d_{\text{in}}(\mathbf{x}) = \|x\| - 1. \tag{8.12}$$

Then EikMesh can deal with an input PSLG that only contains S_{out} because ∇d_{in} is uniquely defined everywhere.

8.4 Boundary elements

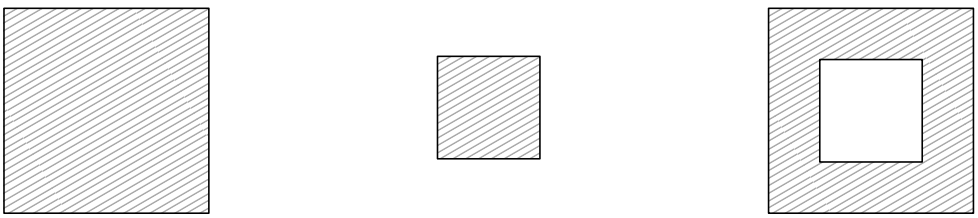
Boundary restrictions do not only make the mesh alignment difficult, but many meshing algorithms produce low-quality elements near $\partial\Omega$. As demonstrated in Section 7.5, DistMesh is no exception. In this section, we discover the root cause of these low-quality triangles and how special local operations can improve their quality. Inspired by the truss analogy, I develop different solutions leading to an overall increased minimal element quality ρ_{min}.

The forced-based smoothing introduced by Persson and Strang runs into problems if, for some triangle τ defined by vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$, there is a force-equilibrium, that is,

$$F(\mathbf{v}_1) \leq \epsilon \ \wedge \ F(\mathbf{v}_2) \leq \epsilon \ \wedge \ F(\mathbf{v}_3) \leq \epsilon \tag{8.13}$$

for some small threshold $\epsilon > 0$, but τ is poorly shaped. In other words, the equilibrium is unwanted. For internal triangles such a force-equilibrium might occur but will eventually disappear due to the displacement of nearby vertices. For boundary elements, this is not necessarily the case, but why? Two properties are causing the appearance of unwanted local force-equilibriums:

(i) boundary edges can never be flipped, and



(i) d_{out} (ii) d_{in} (iii) d_{sub}

Figure 8.10: Definition of a spatial domain combining multiple distance functions: the domain d_{out} (i) is subtracted by d_{in} (ii) to construct d_{sub} (iii).

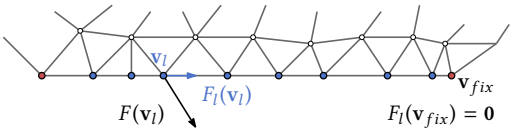


Figure 8.11: A slide point $\mathbf{v}_l$ (blue) can only move on a line-segment l, i. e., between two fix points (red). This is realized by projecting $F(\mathbf{v}_l)$ onto l.

(ii) the movement of boundary vertices is restricted.

The first property causes long and the second one short boundary edges.

8.4.1 Long boundary edges

Let us look at the first case, i. e., long boundary edges. In Section 8.2.1, I showed how edge flips prevent vertices from crossing edges. However, for boundary edges, this is not possible. Therefore, a vertex might move arbitrarily close to a boundary edge and might cross it. Consequently, the quality of boundary triangles can drop arbitrarily low at any point in the smoothing phase. This situation is depicted in Fig. 8.12. The root cause of the problem is that there is no force pushing $\mathbf{v}$ away from the boundary. The boundary edge becomes the longest edge of the low-quality element τ. If we look closely, the force-based smoothing might move $\mathbf{v}$ very slowly towards e – it might never reach e and thus might never leave Ω. Consequently, neither $\mathbf{v}$ nor the poorly shaped triangle $\mathbf{u}_2\mathbf{v}\mathbf{u}_1$ will ever be removed by the approach suggested by Persson and Strang. Because each single edge force acts only repulsively, e is not necessarily compressed, and since e can not flip, there is no force pushing $\mathbf{v}$ away from e. Let us look at three strategies, which all significantly increase ρ_{min} compared to DistMesh.

Vertex projection

One solution to the problem, which I suggested in [336, 338], is to remove low-quality triangles by removing e and projecting $\mathbf{v}$ onto $\partial\Omega$. Projection can be realized by a combination of RemoveFace and Move, see Fig. 8.2vii. Instead of projecting only outside vertices back in, EikMesh projects all boundary vertices. It removes a boundary face τ only if its (single) boundary edge is its longest edge and $\rho(\tau)$ is below some quality minimum. Going back to the truss analogy, one can imagine a bar e breaks under the applied tension.

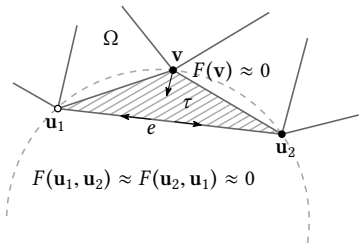


Figure 8.12: A low quality boundary triangle $\mathbf{u}_2\mathbf{v}\mathbf{u}_1$.

Figure 8.13: The concept of virtual edges: the virtual edge e_{vir} helps to push $\mathbf{v}$ away from $\partial\Omega$ (i). The triangle τ is perfectly shaped for $\|e_{\text{vir}}\| = \sqrt{3}\|e\|$ (ii).

Edge split

Another technique I propose is to split e at its midpoint $\mathbf{x}_e$ by executing SplitEdge(e), compare Fig. 8.2iv. Again we can imagine e to almost break under the tension applied, but instead of letting it break, we strengthen the structure by introducing new bars and a new node. EikMesh decides by the same quality minimum if an edge should be split.

Virtual edges

The last technique I propose is to introduce *virtual edges*, that is, edges that are not part of the mesh $\mathcal{T}$ but part of the force computation. The idea is to compensate for the 'missing' triangle neighboring e. Let us imagine a perfectly shaped copy of τ mirrored at e. If $\mathbf{v}$ moves towards e, it will be flipped. Let us call the flipped edge e_{vir} and its second end point $\mathbf{v}_{\text{vir}}$. Furthermore, let $\mathbf{x}_e$ be the midpoint of e, then

$$e_{\text{vir}} = \{\mathbf{v}, \mathbf{v} + 2(\mathbf{x}_e - \mathbf{v})\} = \{\mathbf{v}, 2\mathbf{x}_e - \mathbf{v}\} \tag{8.14}$$

is the virtual edge of e. τ would be perfectly shaped if it would be an equilateral triangle of side length $\|e\|$. Since the side length of e might change according to h, I choose

$$\lambda_k(e_{\text{vir}}) = \frac{\|e_{\text{vir}}\|}{\sqrt{3}h(e)} = \frac{\|e_{\text{vir}}\|}{h(e_{\text{vir}})}. \tag{8.15}$$

Therefore, the desired edge length of the *virtual edge* e_{vir} is $\sqrt{3}h(e)$. Consequently,

$$F_{\text{int}}(\mathbf{v}, \mathbf{v}_{\text{vir}}) = \frac{\mathbf{v} - \mathbf{x}_e}{\|\mathbf{v} - \mathbf{x}_e\|} \cdot \hat{\eta}(\lambda_k(e_{\text{vir}})) \cdot \sqrt{3}h(e) = \frac{\mathbf{v} - \mathbf{x}_e}{\|\mathbf{v} - \mathbf{x}_e\|} \cdot \hat{\eta}(\lambda_k(e_{\text{vir}})) \cdot h(e_{\text{vir}}) \tag{8.16}$$

is the *virtual force* that acts on $\mathbf{v}$. The technique does neither introduce new mesh elements nor removes any. The advantage of using *virtual edges* over the other two techniques is a more stable number of mesh points and an even smoother vertex movement. Furthermore, no additional parameter, such as a minimum quality, has to be introduced. It naturally extends the forced-based approach. Therefore, it is the default method used by EikMesh.

8.4.2 Short boundary edges

Besides long edges, too short edges are the second reason for poorly shaped boundary triangles. The movement restriction at the boundary causes these short edges. Again, the root cause is a local force equilibrium that hinders the mesh improvement process. This time the pressure applied is high.

Let us imagine such a short boundary edge $e = \{\mathbf{v}, \mathbf{u}_1\}$. By its definition, $F(\mathbf{v}, \mathbf{u}_1)$ is rather large thus e should become longer during the improvement phase. The only reason why this might not happen is a force equilibrium at $\mathbf{v}$ and $\mathbf{u}_1$, that is,

$$F(\mathbf{v}) \leq \epsilon \ \wedge \ F(\mathbf{u}_1) \leq \epsilon$$

for some threshold $\epsilon > 0$. Consequently, there has to be a strong force acting in the opposite direction of $F(\mathbf{v}, \mathbf{u}_1)$ and $F(\mathbf{u}_1, \mathbf{v})$ at $\mathbf{v}$ and $\mathbf{u}_1$, respectively. I observed that this situation only occurs at three-degree vertices, as depicted in Fig. 8.14. For any other situation, a force equilibrium can either not be established or eventually vanishes after some improvement steps.

My solution to overcome this problem stems from the truss analogy: in combination, a small net force $F(\mathbf{v})$ and a large absolute force

$$F_{\text{abs}}(\mathbf{v}) = \sum_{\substack{e \in \mathcal{E} \\ e = \{\mathbf{v}, \mathbf{u}\}}} \|F(\mathbf{v}, \mathbf{u})\| \tag{8.17}$$

indicate high unavoidable pressure at $\mathbf{v}$. Consequently, the node breaks and CollapseVertex($\mathbf{v}$) is executed. For estimating the breaking point, I approximate the maximal possible force by

$$F_{\text{max}}(\mathbf{v}) = 3 \cdot \eta_k(\mathbf{v}, \mathbf{v}) = 3 \cdot \hat{\eta}(0) \cdot h_k(\mathbf{v}) = 3 \cdot 1 \cdot h_k(\mathbf{v}) = h_k(\mathbf{v}). \tag{8.18}$$

Note that $h_k(\mathbf{v})$ is the magnitude of an internal edge force at $\mathbf{v}$, if the length of the edge is zero. Since $\mathbf{v}$ is a three-degree vertex, everything is multiplied by 3. The condition for executing CollapseVertex($\mathbf{v}$) is given by

$$\|F(\mathbf{v})\| < \lambda_1 \cdot F_{\text{max}}(\mathbf{v}) \tag{8.19}$$

and

$$F_{\text{abs}(\mathbf{v})} > \lambda_2 \cdot F_{\text{max}}(\mathbf{v}) \tag{8.20}$$

with $0 \leq \lambda_1, \lambda_2 \leq 1$. I propose to use $\lambda_1 = 0.1$ and $\lambda_2 = 0.4$, compare Fig. 8.14.

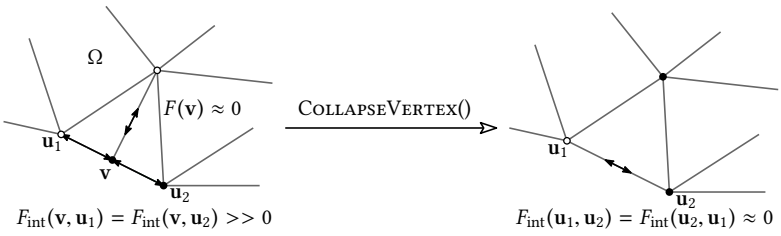


Figure 8.14: Collapse of vertices: it improves the quality of boundary triangles by eliminating short edges.

8.5 Initialization

So far, I have assumed that some initial mesh $\mathcal{T}_0$ is given, and I have omitted its construction. One could use the initialization phase of DistMesh. It is easy to implement and leads to high-quality meshes for geometries, for which the ∇d_Ω is uniquely defined everywhere. However, since DistMesh rejects points randomly, the overall quality of the initial mesh is low. Poorly shaped elements might be present anywhere, and the quality of the "worst" triangle $\rho_{\min}$ can be arbitrarily low. Therefore, much work is required. Some points move a considerable distance which triggers several topological changes. Furthermore, the method may generate many unused vertices that will be inserted to be rejected right afterward – a tedious and wasteful process, especially when using a more sophisticated mesh data structure like the doubly connected edge list (DCEL). Since the rejection is probabilistic, one has less control over the spatial resolution of $\mathcal{T}_0$. Moreover, the process requires to compute the Delaunay triangulation, which I want to avoid.

Therefore, I developed a new initialization strategy. The goal was to replace the probabilistic method of Persson and Strang [221] with one that

(i) is deterministic,

(ii) generates a mesh consisting of mostly high-quality elements,

(iii) does not rely on the computation of the Delaunay triangulation,

(iv) and is stored in a cache-friendly manner.

In the following sections, I provide an in-depth specification of the implemented initialization phase.

8.5.1 Refinement strategy

EikMesh constructs an initial high-quality mesh by applying the hierarchical mesh refinement strategy described in [21, 336]. The starting point is a square with side length h_Ω containing the whole domain Ω. This square is split into two triangles τ_1, τ_2 by connecting two of its non-adjacent vertices. The longest, which is also the only internal edge, is marked. Triangles are split by inserting a new vertex at the midpoint of their longest edge using SplitEdge(e). EikMesh splits an edge e only if it is the longest edge of both neighboring triangles. It inserts the vertex at the midpoint of the edge and connects it to the two opposite vertices. Figure 8.15 depicts the situation.

Splitting internal triangles creates four child triangles while splitting a boundary element returns only two new children. By construction, marked edges are the longest edge of their respective triangle. This procedure is repeated recursively until marked edges are sufficiently short,

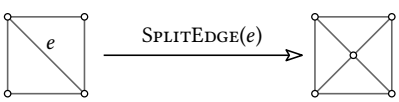


Figure 8.15: SplitEdge(e) of the longest edge of two triangles.

that is, they satisfy

$$\forall \mathbf{x} \in |e| : \|e\| \leq h(\mathbf{x}). \tag{8.21}$$

Despite $\{\mathbf{x} \in |e|\}$ being an infinite set of points, we can use the following minimum

$$h_{\min}(e) = \min_{\mathbf{x} \in e} h(\mathbf{x}) \tag{8.22}$$

to evaluate Eq. (8.21) for finitely many sample points. $h_{\min}$ has to be provided. In Section 8.6, I explain how one can compute it.

Following this refinement strategy, the constructed isosceles triangles are congruent to each other. Let us assume h is constant, and $m = 2^k \leq 2n$ is the number of triangles of $\mathcal{T}_0$, then

$$2^k \cdot \sum_{i=0}^{k} \frac{1}{2^i} = 2^{k+1} - 1 = 2m - 1 \tag{8.23}$$

is the total number of constructed triangles. The construction requires $O(2m) = O(n)$ time and $O(k) = O(\log(n))$ parallel time. The length of the longest edge of each triangle is $2^{-k+(3/2)} h_\Omega$. Its short edges have a length equal to $2^{-k+(1/2)} h_\Omega$. The quality of each constructed triangle is $\rho_1(\tau) = 2\sqrt{2} - 2 \approx 0.83$. Note that the quality estimation is accurate for arbitrary element size functions h. If no constrained line-segments are inserted, the initial quality of the mesh is $\rho_1(\mathcal{T}_0) \approx 0.83$ as well. Otherwise, triangles neighboring those line-segments can be poorly shaped. Consequently, most movement and, therefore, most topological changes occur at regions close to geometrical constraints.

8.5.2 Construction of the Sierpinski list

During the recursive creation, I maintain a (linear) list containing all triangles in an order based on a space-filling curve called the Sierpinski curve. From now on, I call this list *Sierpinski list*. The planar curve goes through the midpoint of each triangle one by one. Thereby, it sorts two-dimensional objects in a one-dimensional fashion. As Behrens and Bader states:

> *"It can be shown that neighborhood relations in the mesh are preserved to a large extent on the curve. In other words, if elements are neighbors in the mesh, they are also most likely to be close to each other in the consecutive sequence of the SFC."*
> – Behrens and Bader [21]

Consequently, spatially close triangles are likely to be close in the maintained *Sierpinski list*.

It is beneficial for modern CPUs if consecutive memory accesses are close together in the address space of the memory. Since all operations of EikMesh are local, their execution can be improved on cache-sensitive hardware, like CPUs, by placing neighboring mesh elements close in memory. Like pearls on a string, I order these mesh elements along the Sierpinski curve in the main memory. Since the ordering is mapped into the hardware memory, one benefits from the cache-effect: in short, if a mesh element is accessed, a whole cache-line is loaded from the main memory into the much more efficient cache. Therefore, accessing another mesh element that is part of the cache-line does not require this memory transfer; hence, it is much faster. If the

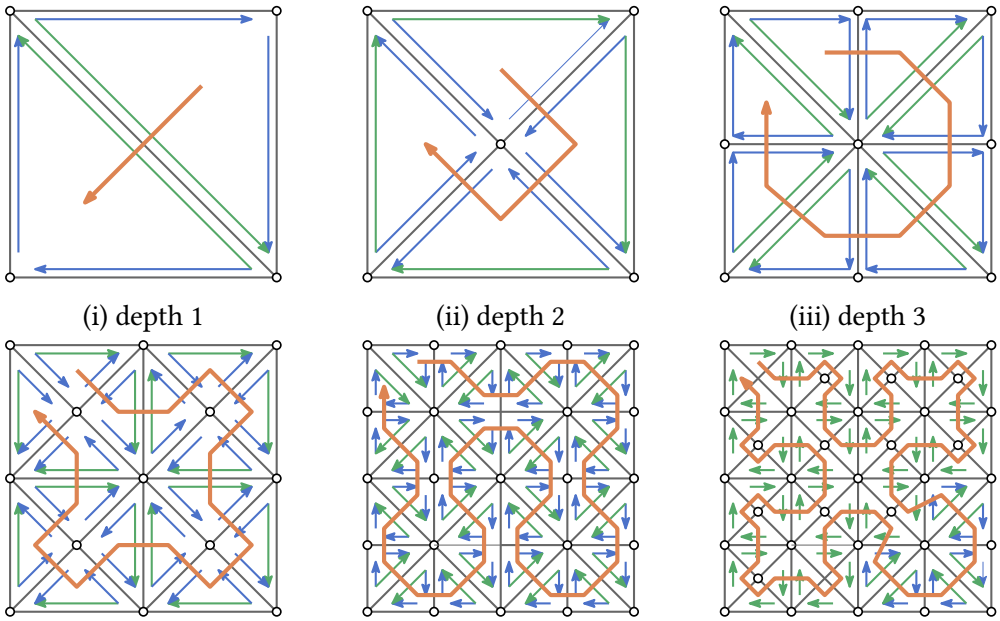


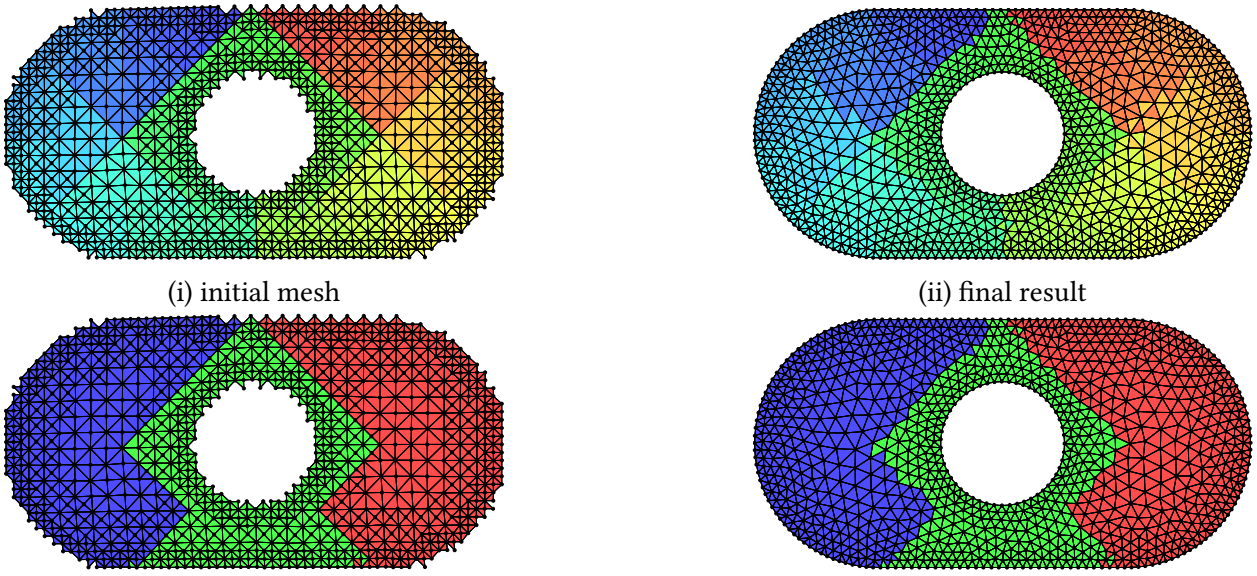
Figure 8.16: Recursive construction of the *Sierpinski list* indicated by the green arrows and the orange path. The blue arrows indicate elements of the *Sierpinski list* of the next depth. For each consecutive recursion depth, the orientation of the *Sierpinski list* changes.

CPU has to fetch a cache line, it might run out of work and stall while waiting for the slow main memory. Stalls due to cache misses delay potential computation since modern CPUs can execute hundreds of instructions in the time taken to fetch a single cache line from the main memory. The performance of algorithms, which successively access spatially close mesh elements benefits this memory arrangement. Many algorithms, such as numerical eikonal solvers, which I discuss in Chapter 9, fall into this category. In the following, I describe the construction of the *Sierpinski list* based on the doubly-connected edge list (DCEL).

I do not maintain a list of triangles but a list of longest (half-)edges of the respective triangles. After splitting the square containing Ω, the *Sierpinski list* contains two half-edges. In Fig. 8.16 list elements are represented by purple arrows. Note that the arrows' orientation in Fig. 8.16 indicate the order of the elements in the *Sierpinski list*, which is different from the orientation of the actual half-edge. To differentiate, I call these arrows *Sierpinski edges*.

After splitting a half-edge e_h, it is replaced by $e_{h,\text{prev}} = \textsc{Previous}(e_h)$ and $e_{h,\text{next}} = \textsc{Next}(e_h)$ in the *Sierpinski list*. Figure 8.16 shows these half-edges as blue arrows. The orientation of half-edges is counterclockwise, but in the case of a *Sierpinski edge*, it changes back and forth. In Fig. 8.16i it is counterclockwise, in Fig. 8.16ii clockwise, changing back to counterclockwise in Fig. 8.16iii, and so on. This orientation gives the order in which the split half-edge is replaced, that is, either by $e_{h,\text{prev}}, e_{h,\text{next}}$ or $e_{h,\text{next}}, e_{h,\text{prev}}$. Therefore, I also maintain a mapping that gives the orientation of the *Sierpinski edges* of the respective half-edge.

During the splitting process, I ensure that the triangulation remains conforming – a split of a half-edge always results in the split of its twin. The only topological requirement is that both



(i) initial mesh (ii) final result

(iii) initial mesh (iv) final result

Figure 8.17: The ordering of triangles according to the *Sierpinski list*: colors code the index of the triangle in the list. For the initial mesh $\mathcal{T}_0$ the *Sierpinski list* increases the chance that triangles, which are spatially close, are also close in memory (i). This property remains for the final mesh (ii). A partition (iii) - (iv) can be constructed by splitting the *Sierpinski list*.

triangles participating in the split have the same size. Therefore, these splits are local operations that can be processed in parallel. Furthermore, splitting multiple different sized triangles in parallel is possible. In Fig. 8.16vi the resulting mesh consists of triangles of different sizes.

After the refinement has finished, EikMesh inserts constrained line-segments as described in Section 8.3. This requires extra edge splits and the four newly generated child triangles replace their parents in the *Sierpinski list*.

After this process has finished, all mesh elements are sorted according to the constructed list of triangles. Fig. 8.17i displays an initial mesh $\mathcal{T}_0$ where triangles are colored according to their order in the *Sierpinski list*. Because EikMesh displaces vertices and changes the topology, triangles that are neighbors in the main memory might no longer share a common edge. But since most triangles of the initial mesh are well-shaped, only a few topological changes are required. Therefore, most of the neighboring relationships defined by the *Sierpinski list* are either invariant or two consecutive triangles contained in the list are still spatially close together, compare Fig. 8.17ii.

If we store all information necessary to compute the next position of an agent on the mesh elements, a distributed implementation is possible for many models. Such decomposition enables distributed memory parallelization for various problems, including pedestrian simulations. For future work, workload distributing is a source for acceleration since we can use more processors and, therefore, potentially achieve run times for large-scale simulations. One such decomposition results from splitting the *Sierpinski list* into p equally sized sub-lists, where p is the number of processors available, compare Fig. 8.17iv.

8.6 Element size function

Like many other meshing algorithms, EikMesh expects an element size function h, which controls the localized mesh resolution. More precisely, $h(\mathbf{x})$ is the (desired) edge length for elements at $\mathbf{x} \in \Omega$. This function incorporates sizing constraints imposed by two sources: the user (or application) and the geometry of the meshed domain Ω. The application might require a certain mesh resolution at important areas. At the same time, a narrow geometry imposes small elements to be accurately represented. For now, I assume the user-specific requirements are specified by an element size function h_u. Later, in Section 9.4, I come back to this topic. h_Ω is the element size function imposing geometric constraints. Overall, an element size function h must satisfy

$$h(\mathbf{x}) \leq h_\Omega(\mathbf{x}) \ \wedge \ h(\mathbf{x}) \leq h_u(\mathbf{x}) \tag{8.24}$$

for all $\mathbf{x} \in \Omega$. To construct h we require

(1) a user/application defined element size function h_u,

(2) an element size function which captures all geometric constrains h_Ω,

(3) and a construction method.

In this section, I describe the construction of h_Ω and h. I assume h_u and the domain Ω is given.

8.6.1 The local feature size

The curvature of Ω and the so-called local feature size lfs imposes geometric size constraints. The local feature size defined by Ruppert captures local thickness and separation of geometric objects. It is defined for planar straight-line graphs and thus does not consider curved objects.

Definition 8.3 (local feature size of a planar straight-line graph [241])**.** The *local feature size* $\text{lfs}_\mathcal{P} : \mathbb{R}^2 \to \mathbb{R}$ of a planar straight-line graph $\mathcal{P}$ is the radius $\text{lfs}_\mathcal{P}(\mathbf{x})$ of the smallest circle centered at $\mathbf{x}$ which intersects any two disjoint mesh vertices or line-segments.

Another general definition that captures local thickness, separation, and curvature was later introduced by Amenta and Bern [14]. They defined the local feature size of a manifold at $\mathbf{x}$ to be the distance from $\mathbf{x}$ to the manifold's medial axis. To extrapolate this function to the interior Alliez et al. [13] incorporate the distance function d_Ω into its definition.

Definition 8.4 (local feature size of a manifold)**.** Let Ω be some manifold and S_Ω its medial axis. Then its *local feature size* $\text{lfs}_\Omega : \mathbb{R}^2 \to \mathbb{R}$ for any point $\mathbf{x} \in \mathbb{R}^2$ is defined by

$$\text{lfs}_\Omega(\mathbf{x}) = |d_\Omega(\mathbf{x})| + d(\mathbf{x}, S_\Omega), \tag{8.25}$$

where $d(\mathbf{x}, S_\Omega)$ is the distance between $\mathbf{x}$ and the medial axis S_Ω.

In the following, I describe algorithms in order to compute the local feature size for a manifold and a planar straight-line graph. Since for urban environments or buildings, curved objects are either not present or approximated by multiple line-segments, I focus on $\text{lfs}_\mathcal{P}$.

Local feature size for arbitrary objects

Interestingly, one way to compute the local feature size for arbitrary objects is to solving the eikonal equation twice. The idea is that the medial axis of Ω is given by the singularity points of geodesic distance from $\partial\Omega$ to any point in Ω. Rumpf and Telea [240] described a procedure to find these singularity points, and this concept was also used by Koko [164]. In the first step, the authors of [240] solve the following equation

$$\begin{aligned} \|\Phi_{\partial\Omega}(\mathbf{x})\| &= 1, \\ \forall \mathbf{x} \in \partial\Omega : \Phi_{\partial\Omega}(\mathbf{x}) &= 0. \end{aligned} \tag{8.26}$$

$\Phi_{\partial\Omega}(\mathbf{x})$ gives the geodesic distance between $\mathbf{x}$ and the closest of the boundary $\partial\Omega$ and therefore, replaces $|d_\Omega(\mathbf{x})|$ in Eq. (8.25) of Definition 8.4. In the second step one can construct an approximation of the medial axis S_Ω by

$$S_\Omega = \{\mathbf{x} \in \Omega \mid \forall \mathbf{y} \in \mathcal{B}_\epsilon(\mathbf{x}) : \Phi_{\partial\Omega}(\mathbf{x}) \geq \Phi_{\partial\Omega}(\mathbf{y})\}, \tag{8.27}$$

where $\mathcal{B}_\epsilon(\mathbf{x})$ is a ball around $\mathbf{x}$ of some small radius ϵ. The solution of the second eikonal equation

$$\begin{aligned} \|\Phi_{S_\Omega}(\mathbf{x})\| &= 1, \\ \forall \mathbf{x} \in S_\Omega : \Phi_{S_\Omega}(\mathbf{x}) &= 0 \end{aligned} \tag{8.28}$$

computes the geodesic distance from any point $\mathbf{x}$ to the medial axis S_Ω. Therefore, one can replace $d(\mathbf{x}, S_\Omega)$ by $\Phi_{S_\Omega}(\mathbf{x})$ in Eq. (8.25). For a more detailed description of finding a good approximations of S_Ω after solving Eq. (8.26), I refer to [240].

Local feature size for planar straight-line graphs

I compute the local feature size of a domain defined by a planar straight-line graph. To do so, I use Ruppert's algorithm, see Section 6.5.4. As observed by Pav and Walkington [218], Ruppert developed not only a Delaunay-based meshing algorithm but a method to compute an approximation of the local feature size of a planar straight-line graph. Theorem 8.1 gives the connection.

Theorem 8.1 (local feature size of a planar straight-line graph [218]). *Given any planar straight-line graph $\mathcal{P}$, then Ruppert($\mathcal{P}$) terminates and upon termination for any vertex $\mathbf{v}$ of the resulting triangulation*

$$\frac{1}{2}\operatorname{lfs}_\mathcal{P}(\mathbf{v}) \leq \min_{\substack{e \in \mathcal{E} \\ e = \{\mathbf{v},\mathbf{u}\}}} \|\mathbf{v} - \mathbf{u}\| \leq \sqrt{2}\operatorname{lfs}_\mathcal{P}(\mathbf{v}) \tag{8.29}$$

holds.

It states that the edge length of the shortest edge incident to the vertex $\mathbf{v}$ is a good estimation for the local feature size at $\mathbf{v}$. Therefore, in order to compute a good approximation of $\operatorname{lfs}_\mathcal{P}$ for a planar straight-line graph $\mathcal{P}$, I construct a background mesh using Ruppert($\mathcal{P}$). Then

$$\min_{\substack{e \in \mathcal{E} \\ e = \{\mathbf{v},\mathbf{u}\}}} \|\mathbf{v} - \mathbf{u}\| \cdot \frac{1}{\sqrt{2}} \tag{8.30}$$

(i) (ii)

Figure 8.18: Quality of a isosceles triangle τ depending on the length of its legs c (i) and $\nabla h = \delta$ (ii): the length of the base is fixed at 1 while the length of c varies. ρ_1 drops below 0.9 for $c < 0.75$ or $c > 1.5$. For $c = b/2 = 1/2$, τ is degenerated (i). If $\delta \geq 2$, the construction of an isosceles triangle becomes impossible. The quality of τ based on δ is depicted in (ii).

is a good approximation for $\text{lfs}_{\mathcal{P}}(\mathbf{v})$, where $\mathbf{v}$ is an arbitrary vertex of the background mesh. I use the same values for $h_\Omega(\mathbf{v})$ because it is also a good approximation for h_Ω. In order to compute values for arbitrary points $\mathbf{x}$, I use the barycentric interpolation for the triangle τ containing $\mathbf{x}$. Note that finding τ requires a point location (see Section 6.7), but the location is fast, because background meshes are coarse.

8.6.2 δ-Lipschitz element size function

Good element size functions balance coarseness and quality of the mesh and, therefore, two divergent requirements. If a certain quality has to be reached, it is essential to limit the gradient of h but not too much to keep the element number low. Let me illustrate this by a simple model, which I use to estimate this limit δ:

let τ be an isosceles triangle, $b = 1$ be the length of its base, and c be the length of its legs. If h is constant, c should approximately be equal to b resulting in a well-shaped triangle τ. If h is not constant and c increases or decreases, the quality of τ drops. Figure 8.18i shows the quality of τ depending on the length of c. For this model, achieving a quality $\rho_1(\tau) \geq 0.9$ implies $0.75 \leq c \leq 1.5$. In the following, I connect c to an actual edge length function h.

Let $\mathbf{x}_b, \mathbf{x}_c$ be the midpoint of the base and the left leg of τ. First we fix $h(\mathbf{x}_b) = 1$. Furthermore, let us assume h is linear (and δ-Lipschitz), that is,

$$\forall \mathbf{x} : \nabla h(\mathbf{x}) = \delta \text{ for some } \delta > 0. \tag{8.31}$$

These requirements imply

$$h(\mathbf{x}) = 1 + \|\mathbf{x} - \mathbf{x}_b\| \cdot \delta. \tag{8.32}$$

Because the triangle τ is isosceles, $\|\mathbf{x}_c - \mathbf{x}_b\|$ is equal to $c/2$ and we get

$$h(\mathbf{x}_c) = 1 + c/2 \cdot \delta. \tag{8.33}$$

Assuming $h(\mathbf{x}_c) = c$ and solving for c gives

$$c = \frac{2}{2-\delta}. \tag{8.34}$$

Substituting $2/(2-\delta)$ for c in the definition of ρ_1 gives

$$\rho_1(\delta) = -\frac{1}{4}(\delta - 2)(\delta + 2) \tag{8.35}$$

and solving for δ leads to

$$\delta_1(\rho) = 2\sqrt{1-\rho}. \tag{8.36}$$

Note that $\delta \geq 2$ makes the construction of the isosceles triangle impossible because $h(\mathbf{x}_c)$ grows faster than c – in other words, we ask for $h(\mathbf{x}_c) > c$ which is a contradiction. The described model is optimistic, because it disregards all geometric constraints. Consequently, if one chooses $\delta = \delta_1(0.95) \approx 0.45$ one expects the quality ρ_1 of the resulting mesh to be at most 0.95. I propose that h should be δ-Lipschitz for

$$\delta_1(0.98) \leq \delta \leq \delta_1(0.90) \Rightarrow 0.28 \leq \delta \leq 0.63. \tag{8.37}$$

Remember, h combines both geometric constraints (h_Ω) and constraints defined by the user/application (h_u).

Aside from h to be linear and δ-Lipschitz, I need h to be defined not only for the meshed domain Ω but for the whole Euclidean space. During the meshing process, points move outside of Ω. This requirement ensures that we can deal with these outside points. The following definition emerges.

Definition 8.5 (element size function)**.** A function $h : \mathbb{R}^2 \to \mathbb{R}^+$ is an element size function if h is stepwise linear and δ-Lipschitz, that is, for all $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^2$

$$h(\mathbf{x}_j) \leq h(\mathbf{x}_i) + \delta \cdot \|\mathbf{x}_i - \mathbf{x}_j\| \tag{8.38}$$

holds, where δ is some constant.

I already used and extended this definition to edges of a mesh $\mathcal{E}$ such that $h(e) = h(\mathbf{x}_e)$ with $e = \{\mathbf{v}_1, \mathbf{v}_2\} \in \mathcal{E}$, $\mathbf{x}_e = (\mathbf{v}_1 + \mathbf{v}_2)/2$. As a side note, $\text{lfs}_\mathcal{P}$ as well as lfs_Ω are 1-Lipschitz.

To limit the gradient of h, I propose a construction of an 'almost' δ-Lipschitz element size function h, that is,

$$h(\mathbf{v}_j) \leq h(\mathbf{v}_i) + \delta \cdot d_\mathcal{T}(\mathbf{v}_i, \mathbf{v}_j), \tag{8.39}$$

where $d_\mathcal{T}(\mathbf{v}_i, \mathbf{v}_j)$ is the length of the shortest path between $\mathbf{v}_i$ and $\mathbf{v}_j$ on the mesh $\mathcal{T}$. Since

$$d_\mathcal{T}(\mathbf{v}_i, \mathbf{v}_j) \geq \|\mathbf{v}_i - \mathbf{v}_j\|, \tag{8.40}$$

it makes sense to choose δ to be slightly smaller than the suggested values above. However, if $\mathbf{v}_i$ is adjacent to $\mathbf{v}_j$, Eq. (8.40) matches Eq. (8.38) because the nearest-neighbor graph is a subgraph of the Delaunay triangulation. To ensure h to be approximately δ-Lipschitz, I propose Algorithm 13 for its construction. It uses a background mesh $\mathcal{T}$ constructed by Ruppert($\mathcal{P}$) and an element size function h_u defined by the user. The heap $\mathcal{H}$ contains tuples $(h_\mathbf{v}, \mathbf{v})$ where $h_\mathbf{v}$ represents $h(\mathbf{v})$ and $\mathbf{v}$ is a vertex of the background mesh. $\mathcal{H}$ is sorted by $h_\mathbf{v}$ and $\mathcal{H}$.min() (Line 7) returns the smallest element in $\mathcal{H}$.

Algorithm 13: ElementSizeConstruction

Input: background mesh $\mathcal{T}$, user-defined element size function h_u, a constant δ
Output: δ-Liptschitz element size function h

foreach $\mathbf{v} \in \mathcal{T}$ **do**
  $h_\mathbf{v} \leftarrow h_u(\mathbf{v})$;
  foreach $\mathbf{w}$ adjacent to $\mathbf{v}$ **do**
    $h_\mathbf{v} \leftarrow \min\{h_\mathbf{v}, \|\mathbf{v} - \mathbf{w}\|/\sqrt{2}\}$;
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{(h_\mathbf{v}, \mathbf{v})\}$;
while $\mathcal{H} \neq \emptyset$ **do**
  $(h_\mathbf{v}, \mathbf{v}) \leftarrow \mathcal{H}.\min()$;
  $\mathcal{H} \leftarrow \mathcal{H} \setminus \{(h_\mathbf{v}, \mathbf{v})\}$;
  foreach $\mathbf{w}$ adjacent to $\mathbf{v}$ **do**
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \{(h_\mathbf{w}, \mathbf{w})\}$;
    $h_\mathbf{w} \leftarrow \min\{h_\mathbf{w}, h_\mathbf{v} + \delta \cdot \|\mathbf{v} - \mathbf{w}\|\}$;
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(h_\mathbf{w}, \mathbf{w})\}$;
$\forall \mathbf{v} \in \mathcal{T} : h(\mathbf{v}) \leftarrow h_\mathbf{v}$;
return h;

Lemma 8.1 (element size construction termination)**.** *ElementSizeConstruction terminates after n steps and requires $O(n \log(n))$ time, where n is the number of vertices of $\mathcal{T}$.*

Proof. Since after each iteration $\mathcal{H}$ contains one less element, ElementSizeConstruction terminates after n steps, where n is the number of vertices of $\mathcal{T}$. Updating an element of $\mathcal{H}$ requires $O(\log(n))$ time. Therefore, the overall construction requires $O(n \cdot \log(n))$ time. ∎

Lemma 8.2 (element size construction)**.** *ElementSizeConstruction ensures that for*

$$\forall \mathbf{v}, \mathbf{w} : h_\mathbf{w} \leq h_\mathbf{v} + \delta \cdot d_\mathcal{T}(\mathbf{w}, \mathbf{v}) \tag{8.41}$$

holds.

Proof. I prove the following invariant: let $\mathbf{v}, \mathbf{w}$ be any two vertices that have already been removed from the heap, then

$$h_\mathbf{w} \leq h_\mathbf{v} + \delta \cdot d_\mathcal{T}(\mathbf{w}, \mathbf{v}). \tag{8.42}$$

After ElementSizeConstruction terminates, the heap is empty. Therefore, we can prove the statement if we prove this invariant.

After removing the first element, this equation is satisfied. Now let us assume the invariant holds for some amount of vertices, and we remove another one; let's say $\mathbf{v}$. Let us pick some arbitrary vertex $\mathbf{w}$ that already got removed. And let us look at the shortest path $\mathbf{v}, \mathbf{u}_1, \ldots, \mathbf{u}_k, \mathbf{w}$ between $\mathbf{v}$ and $\mathbf{w}$ with respect to $\mathcal{T}$. Since $\mathbf{v}$ was removed after $\mathbf{w}$, we get

$$h_\mathbf{w} \leq h_\mathbf{v} \tag{8.43}$$

and

$$h_{\mathbf{w}} \leq h_{\mathbf{v}} + \delta \cdot d_{\mathcal{T}}(\mathbf{w}, \mathbf{v}) \tag{8.44}$$

follows.

Let us now assume all vertices $\mathbf{u}_1, \ldots, \mathbf{u}_k$ on the shortest path have already been removed from the heap. In that case the invariant implies

$$h_{\mathbf{u}_1} \leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_1, \mathbf{w}) \tag{8.45}$$

and by construction (Line 11 in Algorithm 13)

$$h_{\mathbf{v}} \leq h_{\mathbf{u}_1} + \delta \cdot \|\mathbf{v} - \mathbf{u}_1\| = h_{\mathbf{u}_1} + \delta \cdot d_{\mathcal{T}}(\mathbf{v}, \mathbf{u}_1) \tag{8.46}$$

holds. By substituting $h_{\mathbf{u}_1}$ in Eq. (8.46)

$$h_{\mathbf{v}} \leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_1, \mathbf{w}) + \delta \cdot d_{\mathcal{T}}(\mathbf{v}, \mathbf{u}_1) = h_{\mathbf{w}} + d_{\mathcal{T}}(\mathbf{v}, \mathbf{w}) \tag{8.47}$$

follows.

Now let us assume that $\mathbf{u}_i$ is some vertex such that $\mathbf{u}_1, \ldots, \mathbf{u}_i$ already got removed from the heap. Since $\mathbf{w}$ and $\mathbf{u}_i$ have already been removed from the heap, we can use the invariant. Therefore,

$$h_{\mathbf{u}_i} \leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_i, \mathbf{w}) \tag{8.48}$$

follows. Furthermore, by using the above argument, we get

$$h_{\mathbf{v}} \leq h_{\mathbf{u}_i} + \delta \cdot d_{\mathcal{T}}(\mathbf{v}, \mathbf{u}_i). \tag{8.49}$$

By substituting $h_{\mathbf{u}_i}$ in Eq. (8.49)

$$h_{\mathbf{v}} \leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_i, \mathbf{w}) + \delta \cdot d_{\mathcal{T}}(\mathbf{v}, \mathbf{u}_i) = h_{\mathbf{w}} + d_{\mathcal{T}}(\mathbf{v}, \mathbf{w}) \tag{8.50}$$

follows.

Finally, let us assume that $\mathbf{u}_1$ has not already been removed. Then, we can traverse the path backwards from $\mathbf{w}$ to $\mathbf{v}$ and pick $\mathbf{u}_j$ such that $\mathbf{u}_k, \ldots, \mathbf{u}_{j-1}$ already have been removed from the heap. In this case,

$$h_{\mathbf{v}} \leq h_{\mathbf{u}_j} \tag{8.51}$$

follows. Furthermore, by the above argument

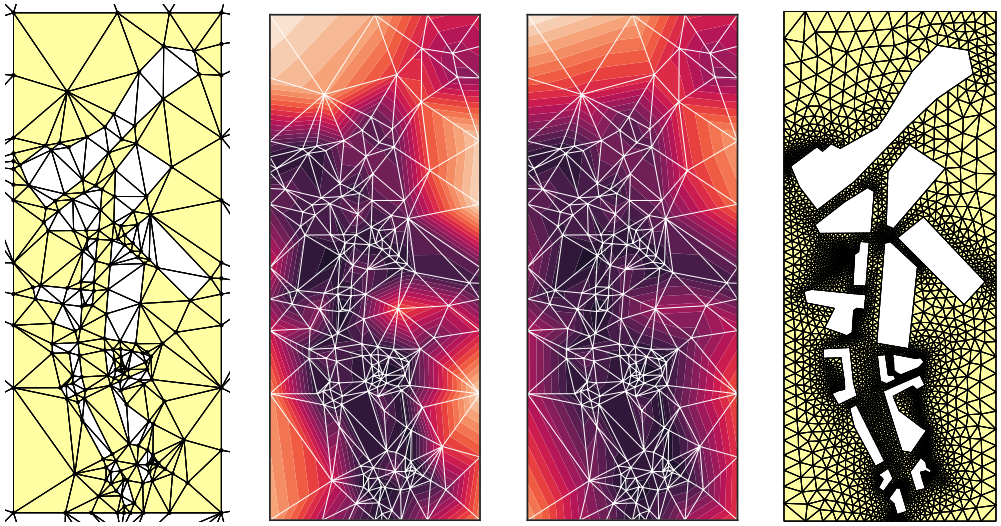
$$h_{\mathbf{u}_j} \leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_j, \mathbf{w}) \tag{8.52}$$

and we can substitute again to get

$$\begin{aligned} h_{\mathbf{v}} \leq h_{\mathbf{u}_j} &\leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_j, \mathbf{w}) \\ &\leq h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_j, \mathbf{w}) + \delta \cdot d_{\mathcal{T}}(\mathbf{u}_j, \mathbf{v}) = h_{\mathbf{w}} + \delta \cdot d_{\mathcal{T}}(\mathbf{v}, \mathbf{w}). \end{aligned} \tag{8.53}$$

Therefore, for all cases the invariant defined by Eq. (8.42) follows. ■

We could replace the iteration over adjacent vertices (Line 9, Algorithm 13) with an iteration over all vertices. This brute force approach would return a truly δ-Lipschitz function. However, we would introduce a quadratic time complexity. Figure 8.19 depicts the construction of an element size function for an urban environment given by a planar straight-line graph $\mathcal{P}$. The contour plot in Fig. 8.19ii shows the local feature size constructed by using Ruppert($\mathcal{P}$). Figure 8.19ii shows the gradient limited element size function after applying Algorithm 13 with $h_u(\mathbf{x}) = \infty$ and $\delta = 0.4$. Applying EikMesh using the constructed element size function generates the mesh depicted in Fig. 8.19iv.



(i) (ii) (iii) (iv)

Figure 8.19: The local feature size $\text{lfs}_{\mathcal{P}}$ of the planar straight-line graph from Section 6.4: the background mesh (i) consists of 193 vertices. The approximation of the local feature size (ii) is limited by ElementSizeConstruction (iii). On the right (iv), the results of EikMesh is displayed using the limited version of the element size function.

8.7 Distance function

The last ingredient worth discussing is the signed distance function d. During the execution of EikMesh, d_Ω has to be evaluated many times for the following reasons: (1) to spot triangles outside of the domain, (2) to compute ∇d_Ω, and if the element size function h depends on d_Ω, (3) we also have to evaluate d_Ω whenever h is evaluated. If we are not careful, d_Ω becomes the computational bottleneck of the meshing algorithm. Therefore, d_Ω must be computed efficiently while maintaining sufficient accuracy. In general, d_Ω can be defined by

$$d_\Omega(\mathbf{x}) = \text{sign}_\Omega(\mathbf{x}) \cdot \min_{y \in \partial\Omega} \|\mathbf{x} - \mathbf{y}\| \tag{8.54}$$

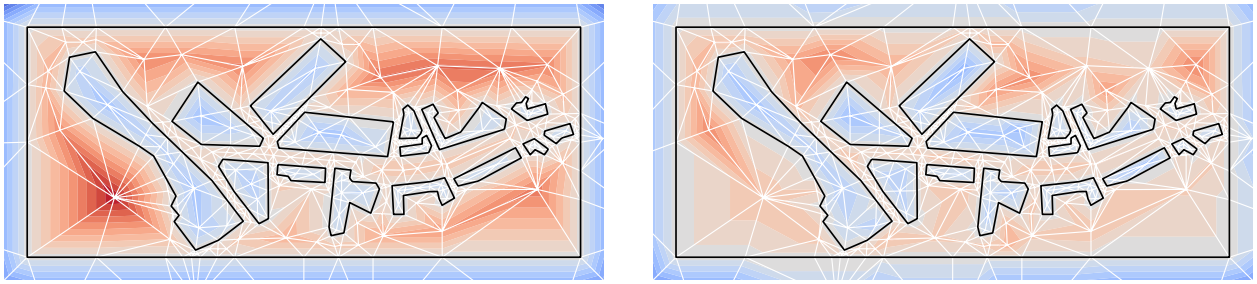
with

$$\text{sign}_\Omega(\mathbf{x}) = \begin{cases} -1, & \text{if } \mathbf{x} \in \Omega \\ +1, & \text{otherwise.} \end{cases} \tag{8.55}$$

Let $\mathcal{P}$ be a segment-bounded planar straight-line graph consisting of the segment-bounding polygon P_0 and holes $P_1, \ldots, P_k$. Furthermore, let d_{P_i} be the signed distance function of the polygon P_i, then

$$d_\Omega(\mathbf{x}) = \max\left\{ d_{P_0}(\mathbf{x}), -\min_{i=1,\ldots k} d_{P_i}(\mathbf{x}) \right\} \tag{8.56}$$

is the implicit representation of the domain. Algorithmically, one could evaluate d_Ω by evaluating all distance functions d_{P_i} whenever a distance has to be computed. In practice, this naive approach is computationally expensive, especially for complex geometries. If the element size function h



(i) (ii)

Figure 8.20: Approximation of d_Ω (Fig. 8.19iv) of the planar straight-line graph from Section 6.4: the background mesh (white lines) consist of 193 vertices. The black lines highlight the level set $d_\Omega = 0$. Blue colors indicate a positive and red colors a negative distance. The brute force approximation (i) is more accurate than using the Fast Marching Method (ii), but for all positions near the boundary $\partial\Omega$, both methods achieve accurate results.

depends on d_Ω, $O(|\mathcal{E}_k|)$ distance evaluations are required for each EikMesh-iteration. Therefore, the distance computation becomes the dominant factor with respect to the run time of EikMesh (and DistMesh).

EikMesh evaluates ∇d_Ω only for boundary vertices. For all other required evaluations of d_Ω, accuracy is less important. Therefore, I evaluate d_Ω for all vertices of the coarse background mesh introduced in the previous section. Furthermore, I use the same interpolation to get approximated values for points enclosed by some triangle τ of the background mesh. The approximated signed distance function for an urban environment is depicted in Fig. 8.20i. This brute force approach still requires n evaluations of d_Ω, where n is the number of vertices of the background mesh.

There are many ways to reduce the required work further. One way is to sort geometric objects like points, line-segments, and polygons in some spatial data structures such as an R-tree [109]. These data structures reduce the number of polygons, which we have to consider. Furthermore, they simplify the evaluation of d_Ω.

Another approach is to reduce the problem to the eikonal equation. Because constrained edges of the background mesh represent all line-segments of the planar straight-line graph, we can identify all vertices $\mathbf{v}$, for which

$$d_\Omega(\mathbf{v}) = 0. \tag{8.57}$$

These vertices are precisely the set of vertices present in the planar straight-line graph $\mathcal{P}$ and its background mesh. Let $\mathcal{V}_0$ be this set. Then the solution $\Phi_{\partial\Omega}(\mathbf{x})$ of the eikonal equation

$$\begin{aligned} \|\Phi_{\partial\Omega}(\mathbf{x})\| &= 1, \\ \forall \mathbf{x} \in \partial\Omega : \Phi_{\partial\Omega}(\mathbf{x}) &= 0. \end{aligned} \tag{8.58}$$

is equal to $-d_\Omega(\mathbf{x})$ for $\mathbf{x}$ in Ω. If $\mathbf{x} \notin \Omega$, then $\Phi_{\partial\Omega}(\mathbf{x}) = d_\Omega(\mathbf{x})$ holds. One can compute an approximation by applying the FastMarchingMethod or some other method on the background mesh, compare Section 9.3. In that case, the propagating wavefront starts at $\mathcal{V}_0$ and merge at the medial axis of Ω. Figure 8.20ii depicts the approximation of $\Phi_{\partial\Omega}$ using the previously constructed background mesh. If one uses a very coarse background mesh, the approximation can be far off. In these cases, the brute force strategy is preferable.

8.8 Examples and discussion

In this section, I show EikMesh's ability to generate high-quality meshes for different artificially and practical domains – I investigate its capability experimentally. Furthermore, I compare EikMesh's generated mesh and intermediate results to those generated by DistMesh introduced in Chapter 7. Additionally, I use one specific example to compare run times between EikMesh and DistMesh. I also want to emphasize that for the more complex geometries of this section, DistMesh fails to construct a mesh that adheres to the domain boundary. Quality measurements are based on the metrics introduced in Section 6.6. One of the artificial test domain is similar to the one I used in Section 7.5, such that a comparison of both algorithms becomes possible. In order to compare run times, I choose another artificial domain such that the performance costs of evaluating d_Ω and h are approximately the same for both algorithms.

I demonstrate the effectiveness and usability of EikMesh by using real-world domains, including domains from the field of interest, that is, pedestrian simulations. For all meshes generated by EikMesh, short boundary edges collapse as described in Section 8.4.2. I stick to $\lambda_1 = 0.1$ and $\lambda_2 = 0.4$. To avoid long boundary edges, I use *edge splits* (see Section 8.4.1) and *virtual edges* (see Section 8.4.1). Edges are split if $\rho_1(\tau) < 0.5$.

Generated meshes are depicted in Figs. 8.21xii, 8.25 to 8.28, 8.29iii and 8.30. Note that EikMesh and DistMesh are implemented in Java using (double precision) standard floating-point arithmetic. For each depicted mesh, *slide points* and *fix points* are highlighted in blue and red, respectively. Any other vertex is black.

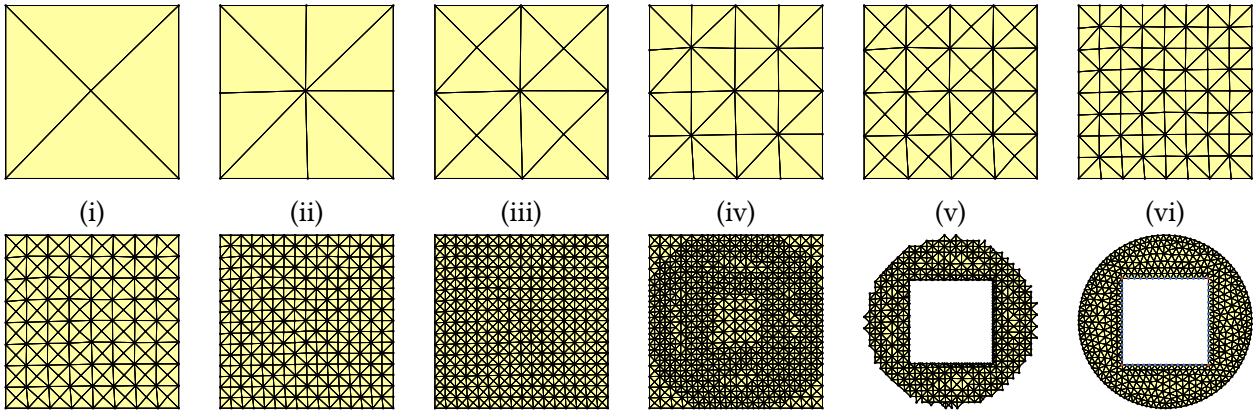
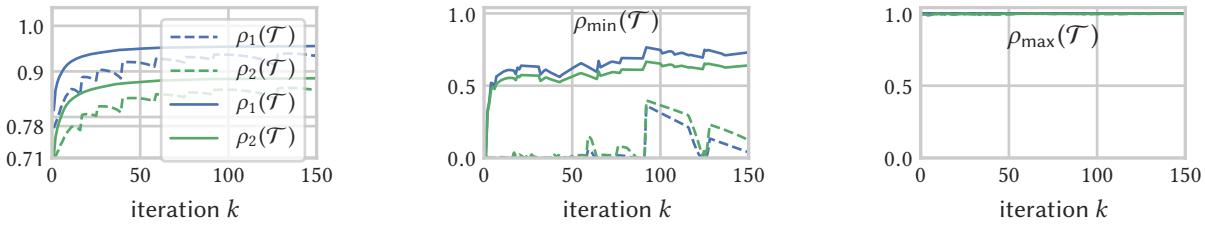


Figure 8.21: The immediate results of EikMesh: the recursive initialization phase (i)-(x), the mesh after removing triangles outside Ω (xi), and after 150 iterations of the smoothing phase have been executed (xii).

8.8.1 Quality comparison

In order to compare the meshes generated by EikMesh and DistMesh, I use the artificial domain used in Section 7.5. The following distance function

$$d_{\text{sub}}(\mathbf{x}) = \max\{d_{\text{circ}}(\mathbf{x}), -d_{\text{rect}}(\mathbf{x})\} \tag{8.59}$$



(i) (ii) (iii)

Figure 8.22: Plots of the mean $\rho_1(\mathcal{T}_k)$, $\rho_2(\mathcal{T}_k)$ (i), minimal $\rho_{1,\min}(\mathcal{T}_k)$, $\rho_{2,\min}(\mathcal{T}_k)$ (ii), and maximum quality $\rho_{1,\max}(\mathcal{T}_k)$, $\rho_{2,\max}(\mathcal{T}_k)$ (iii): solid lines represent qualities achieved by EikMesh. Dashed lines represent qualities achieved by DistMesh. The mean quality ρ_1 of the series of meshes generated by EikMesh approaches 0.96, while the minimum quality stays above 0.48 for $k \geq 5$ and above 0.69 for $k \geq 100$.

defines the domain. It is a circle subtracted by a smaller square, compare Fig. 7.1. Furthermore, I use a 0.2-Lipschitz element size function

$$h(\mathbf{x}) = 0.05 + 0.2 \cdot |d_{\text{sub}}(\mathbf{x})| \,. \tag{8.60}$$

h should lead to high-quality mesh for both, EikMesh and DistMesh. In order to guarantee a fair comparison, I also make sure that both generated meshes consist of approximately the same number of vertices. Additionally, I abandon any approximation of d_Ω and h, and I use 150 improvement iterations. Figure 8.21 depicts immediate results of the initialization phase and the final result of EikMesh.

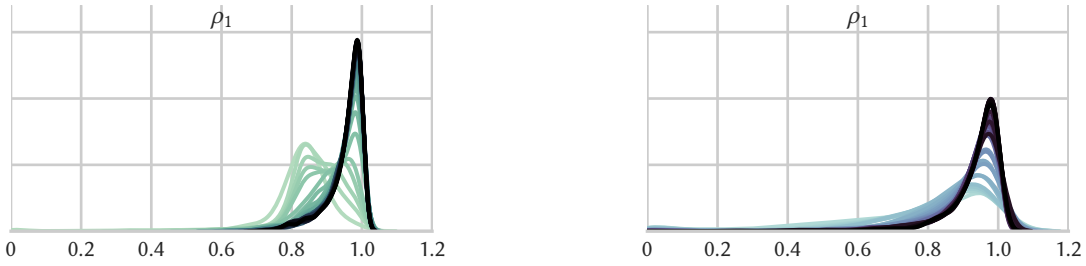
After 10 recursive refinement steps, EikMesh inserts four constrained line-segments and removes all triangles outside of Ω. Inserted constraints hurt the quality of elements intersecting with those constraints, compare Fig. 8.25i. After EikMesh removes elements outside of Ω, there are some critical acute angles at the mesh boundary, that is, acute angles at vertices, which are not fixed points. Those are eliminated by the strategy described in Section 8.2.2.

After 4 improvement iterations, EikMesh increases the minimal quality of the mesh significantly, compare Fig. 8.25ii. Figure 8.25 displays the immediate results of the improvement phase. I picked iteration steps equal to the example in Section 7.5 so that the reader can compare immediate results between both meshing algorithms.

The minimal quality $\rho_{1,\min}$ stays above 0.48 for all iterations $k \geq 5$ and above 0.69 for all iterations $k \geq 100$. EikMesh achieves this improvement using the special treatment of boundary elements discussed in Section 8.4. This can also be observed in Fig. 8.22ii. Maximal element qualities are equal to 1, which is identical to the results of DistMesh. However, the mean qualities ρ_1, ρ_2 stay above the quality values achieved by DistMesh, compare Fig. 8.22i. The convergence is smoother and for almost all consecutive meshes $\mathcal{T}_k, \mathcal{T}_{k+1}$

$$\rho_1(\mathcal{T}_k) < \rho_1(\mathcal{T}_{k+1}) \;\wedge\; \rho_2(\mathcal{T}_k) < \rho_2(\mathcal{T}_{k+1}) \tag{8.61}$$

holds. As I discussed in Chapter 7, this is not true for DistMesh since qualities drop in between two consecutive computation of the Delaunay triangulation, compare Fig. 8.22i.

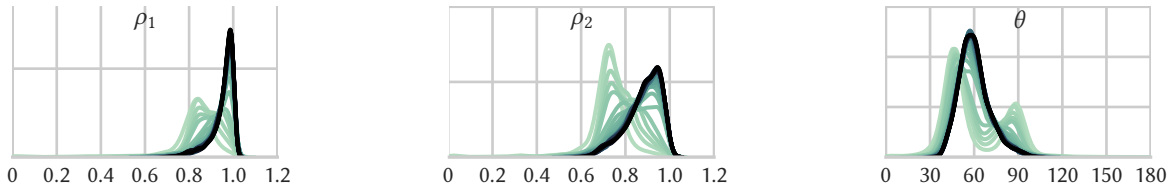


(i) EikMesh (ii) DistMesh

Figure 8.23: Plot of the series of estimation distributions ρ_1, comparing EikMesh (i) with DistMesh (ii): each distribution is estimated using a specific mesh $\mathcal{T}_k$ for $k = 1, 2, 3, 4, 5, 10, 15, \ldots, 145, 150$. A darker curve indicate later iterations, i. e., larger k.

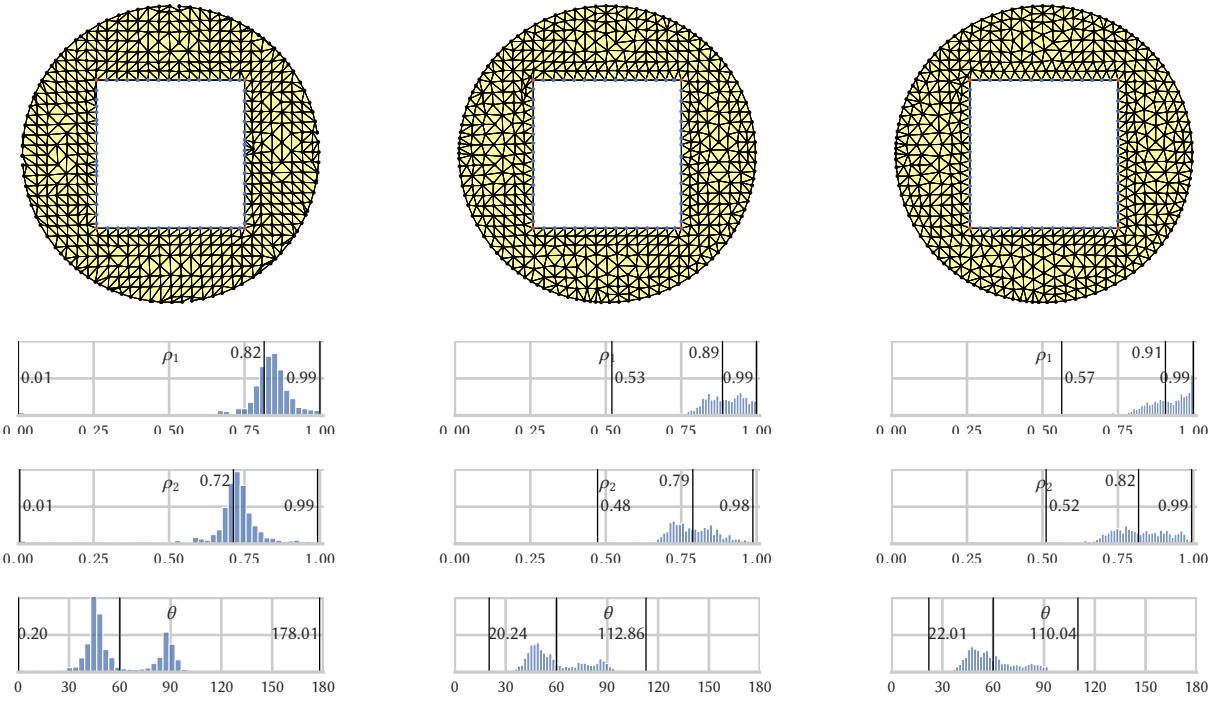
Comparing the series of estimation distributions depicted in Fig. 8.23 reveals that EikMesh starts with a large number of elements τ that satisfy $\rho_1(\tau) \approx 0.83$. Most importantly, the peak for EikMesh (Fig. 8.23i) is higher than the peak for DistMesh (Fig. 8.23ii). Consequently, the proportion of high-quality elements is higher.

The plots of the series of estimation distributions displayed in Fig. 8.24 reveal a similar quality progression during the improvement phase as we observed for DistMesh. However, the peak at 0.96 is higher than the one at 0.94 achieved by DistMesh, compare also Fig. 8.23. In addition to the peak at 0.83 for $\rho_1(\tau), \tau \in \mathcal{T}_0$, we can also observe two peaks at 90° and 45°. The second peak is twice as high. This reflects the new initialization phase of EikMesh. It generates congruent triangle with angles equal to 45°, 90° and 45°. As desired, those two peaks merge step by step into a single peak at 60°. I conclude that EikMesh outperforms DistMesh for this example. The algorithm is effective in eliminating low-quality boundary elements, and it converges fast and smoothly. After 150 improvement iterations, the final mesh consists of overall better elements. At any point in time, the mean quality for the respective mesh is greater than the one achieved by DistMesh.



(i) (ii) (iii)

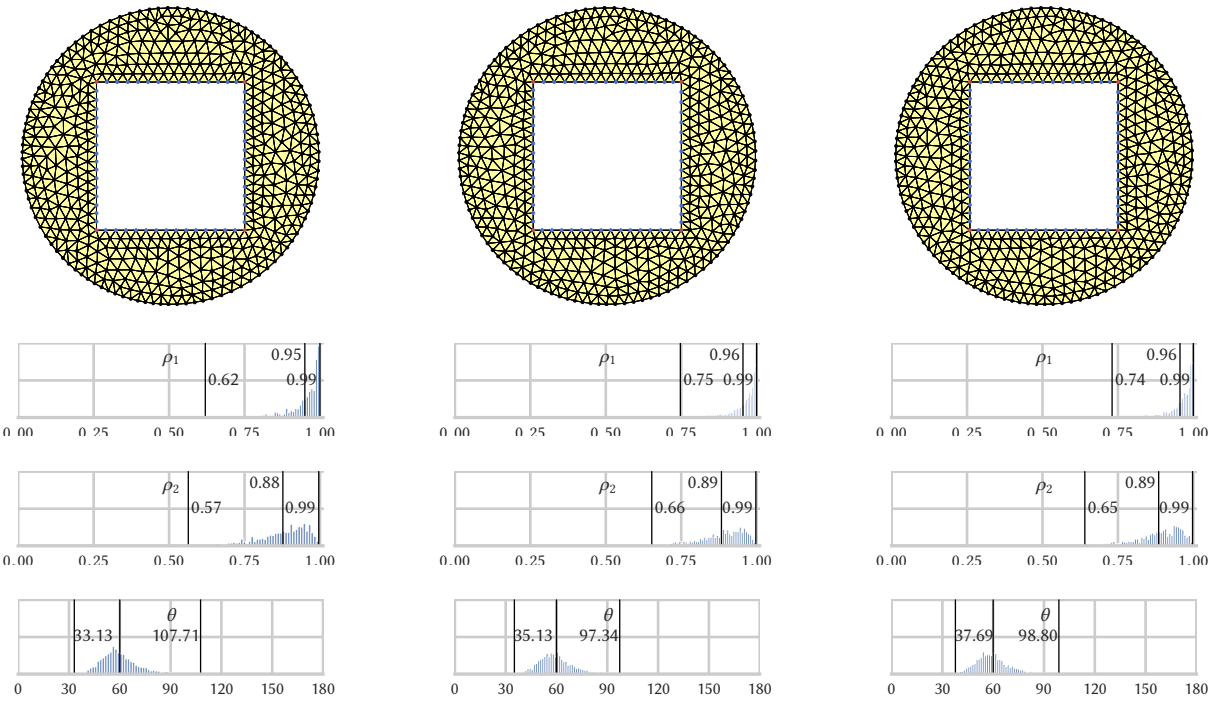
Figure 8.24: Series of estimation distributions ρ_1 (i), ρ_2 (ii) and θ (iii) achieved by EikMesh: each distribution is estimated using a specific mesh $\mathcal{T}_k$ for $k = 1, 2, 3, 4, 5, 10, 15, \ldots, 145, 150$. A darker curve indicate later iterations, i. e., larger k.



(i) $\mathcal{T}_0$

(ii) $\mathcal{T}_4$

(iii) $\mathcal{T}_7$



(iv) $\mathcal{T}_{50}$

(v) $\mathcal{T}_{100}$

(vi) $\mathcal{T}_{150}$

Figure 8.25: The intermediate results of the EikMesh algorithm after its initialization: for iteration $k = 0, 4, 7, 50, 100, 150$ a histogram of qualities ρ_1, ρ_2 and the angles θ of each triangle of $\mathcal{T}_k$ is displayed. The algorithm starts with an initial triangulation constructed by the recursive refinement technique (i). Points move around so that qualities are first harmonized at the boundary (ii)-(iii) and increased for all internal triangles (ii)-(vi). Fix points are highlighted in red, slide points in blue. Minimum, maximum and mean values are highlighted by a vertical black line.

8.8.2 Generated meshes

Airfoil

The airfoil domain is a problem instance from the field of fluid dynamics. It represents the cross-sectional shape of a wing, blade, or sail. Ω is represented by a planar straight-line graph generated by the NACA 4 digit airfoil generator using 100 vertices. The tool can be found on the following website:

http://airfoiltools.com/airfoil/naca4digit.

The mesh is generated by using the distance d_Ω, which gives the distance to the polygon defining the airfoil shape and

$$h(\mathbf{x}) = 0.005 + |d_\Omega(\mathbf{x})| \cdot 0.2. \tag{8.62}$$

Additionally, the enclosing rectangle is defined by

$$\{(-0.3, -0.4), (1.3, -0.4), (1.3, 0.4), (-0.3, 0.4)\}. \tag{8.63}$$

As expected, the mean quality ($\rho_1(\mathcal{T}) \approx 0.96$) is high, compare Fig. 8.26. The smallest angle of the mesh ($\theta_0 \approx 29.15°$) is approximately 30 degrees apart from the optimal angle of 60 degrees. There is some large angle ($\theta_\infty \approx 107.14$), that can be found near the rectangular boundary. However, most of the angle elements are approximately optimal.

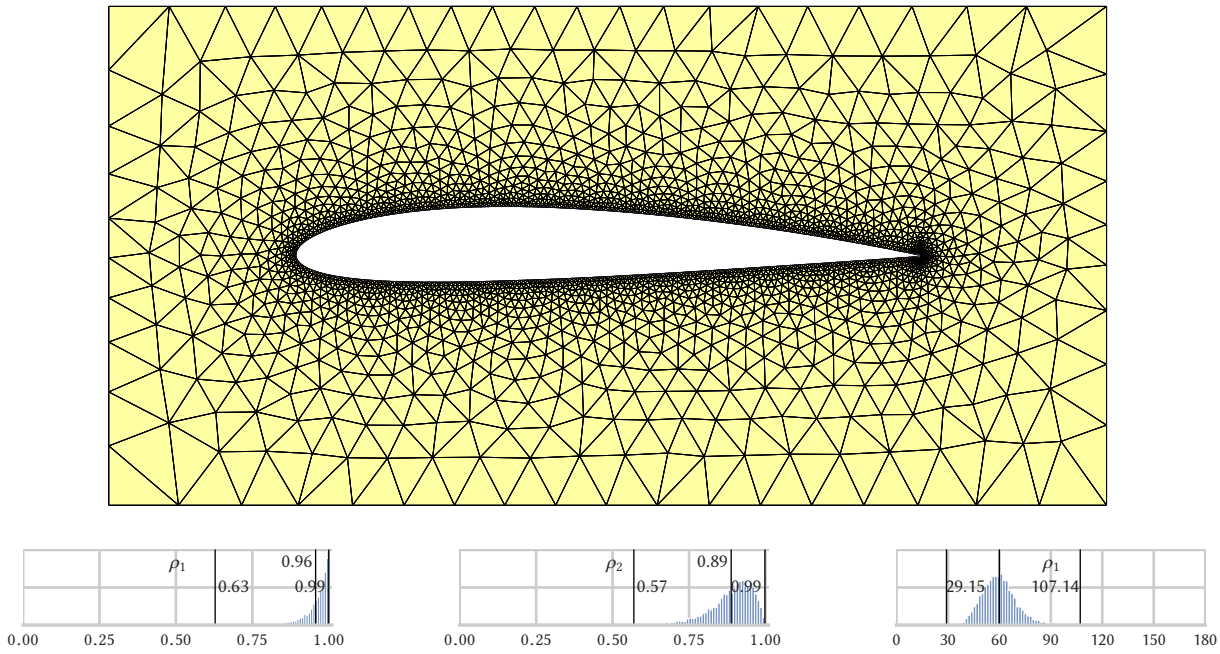
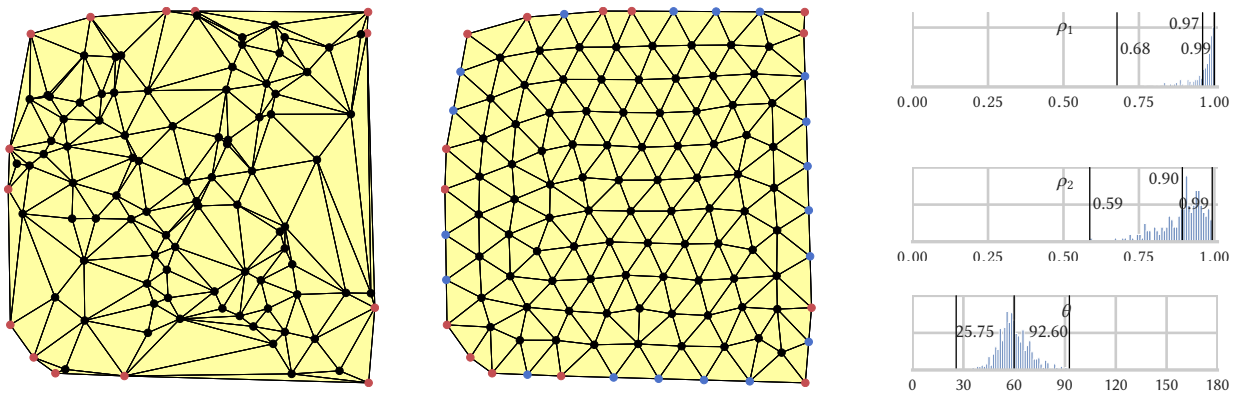


Figure 8.26: The mesh generated by EikMesh using an airfoil domain represented by a planar straight-line graph $\mathcal{P}$ of 100 points: the input is $\mathcal{P}$ and $h(\mathbf{x}) = 0.005 + |d_\Omega(\mathbf{x})| \cdot 0.2$. The histograms of ρ_1, ρ_2 and θ of the generated result are also displayed.



(i) (ii)

Figure 8.27: The mesh generated by EikMesh (ii) using a Delaunay triangulation of 100 random points (i): EikMesh uses $h(\mathbf{x}) = 1.0$. The histograms of ρ_1, ρ_2 and θ of the generated result are also displayed.

Random Delaunay triangulation

The next example illustrates that EikMesh can also improve a given initial triangulation. In this case, the initialization phase is skipped. Figure 8.27i depicts $\mathcal{T}_0$, which is the Delaunay triangulation of 100 randomly chosen points uniformly distributed inside a 10×10 square. Each point of the convex hull of $\mathcal{T}_0$ is marked as a fix point. Because the initialization phase of EikMesh is essential for the resulting mesh resolution, the used edge length function

$$h(\mathbf{x}) = 1.0 \tag{8.64}$$

gives an edge length relative to the smallest current edge length. Only boundary vertices are added due to the edge splitting of long boundary edges. During the improvement phase, 19 sliding vertices are inserted. They slide between two fixed points. Because there is no gradient dependent point projection required, EikMesh works without a distance function d_Ω. The mean quality $\rho_1(\mathcal{T}) \approx 0.97$ of the generated mesh is surprisingly high.

This example also illustrates that we can improve the mesh quality for specific areas of a given mesh. We can imagine that the random Delaunay triangulation is a part of a much larger mesh. If the surrounding elements are of high quality, we can fix all vertices of the boundary of a subdomain. Disabling the splitting mechanism for boundary edges ensures that only the part we want to improve will be changed. Combining this local improvement technique with a partition of the mesh, such as the one presented in Section 8.5, leads to a distributed memory implementation of EikMesh. Each processor can improve its assigned part of the partition in parallel without worrying about the other parts. However, the improvement phase is hindered by the introduced unwanted artificial constraints. To resolve the issue, one could use two or more distinct partitions such that for any vertex or edge, at least for one partition the respective element is not constrained.

Supermarket

The meshed domain of a supermarket, depicted in Fig. 8.28, is the first example that relies on the insertion of constrained line-segments (see Section 8.3.2). It is also a characteristic domain of microscopic pedestrian simulations. Because of the narrow parts of the domain, triangles would overlap with $\mathbb{R}^2 \setminus \Omega$ if we do not use explicit geometry information. Therefore, DistMesh fails to mesh the domain if the element size close to the boundary is too large. It can even fail if h is much smaller than the local feature size. I discussed this problem in Section 7.5. For this more complex geometry, I approximate the distance function d_Ω. The approximation is based on the background mesh introduced in Sections 8.6 and 8.7. I choose a user-defined edge length function

$$h_u(\mathbf{x}) = 0.25 + 0.4 \cdot |d_\Omega(\mathbf{x})| \tag{8.65}$$

and the construction of h described in Section 8.6. Other than DistMesh, EikMesh is able to generate a much more coarse mesh. Using the given edge length function $\rho_1(\mathcal{T}) \approx 0.96$ is almost optimal.

Urban environment

The last example that illustrates the capability of EikMesh is a large-scale domain of a microscopic pedestrian simulation. It is a real-world urban environment (a part of Kaiserslautern in Germany) extracted from Open Street Maps and converted to a planar straight-line graph. For this example, I make use of the background mesh depicted in Figs. 8.29i and 8.29ii. The domain consists of streets surrounded by buildings and narrow passages represented by concave polygons, but there is also much open space. The edge length function is constructed by Algorithm 13 introduced in Section 8.6 using

$$h_u(\mathbf{x}) = \infty \tag{8.66}$$

and $\delta = 0.4$. Therefore, I do not define any user specific requirements but limit the gradient of h such that

$$\max_{\mathbf{x} \in \mathbb{R}^2} |\nabla h(\mathbf{x})| \approx 0.4. \tag{8.67}$$

The mean quality $\rho_1(\mathcal{T})$ of the result $\mathcal{T}$ is approximately 0.92. Due to the geometry's complexity, $\rho_1(\mathcal{T})$ is worse than the value 0.96 estimated by the established optimistic model (see Section 8.6). However, the value is still acceptable. As expected, open areas lead to a coarse mesh resolution and narrow passages to the opposite. Fix points, constrained line-segments, and slide points ensure perfect alignment.

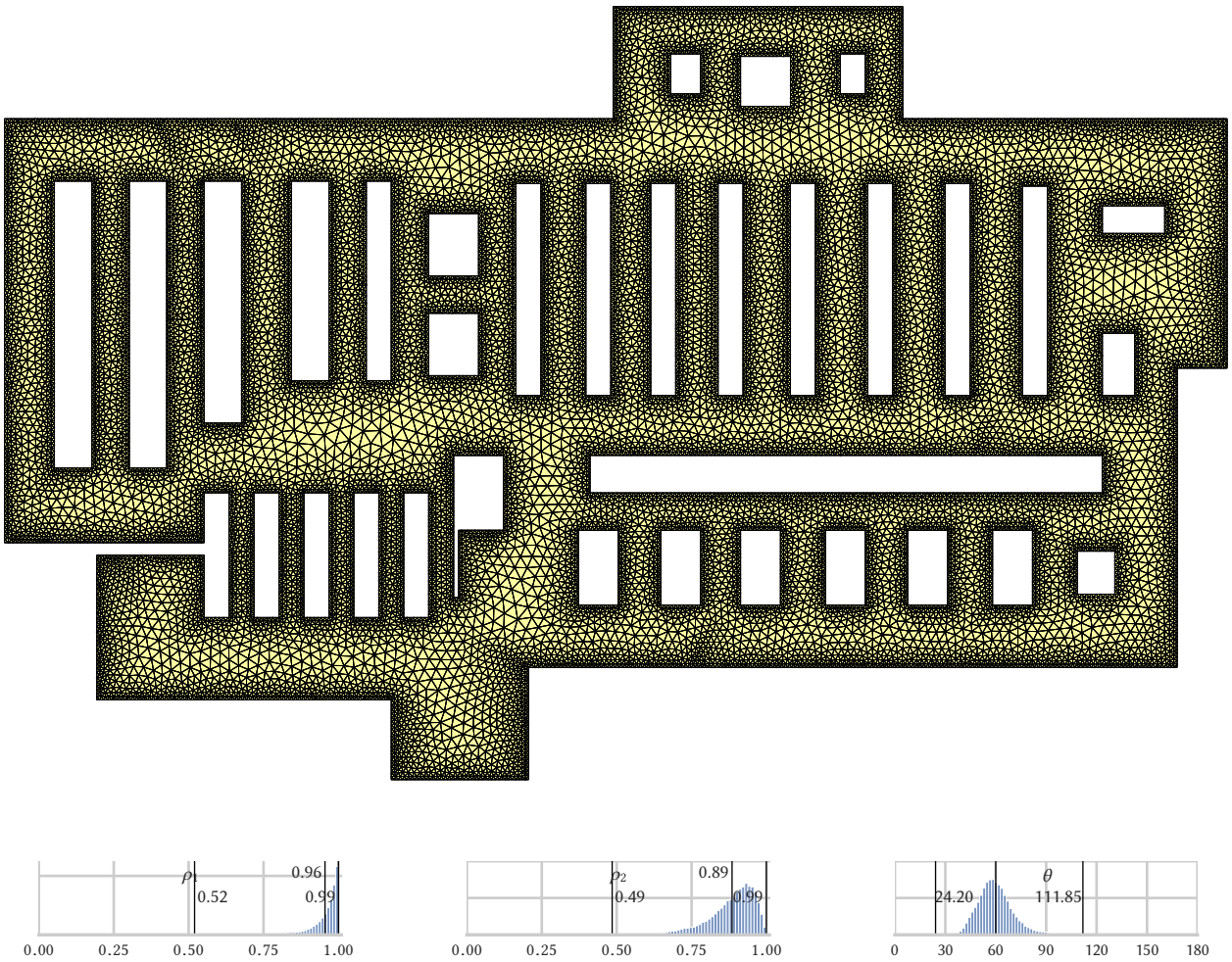


Figure 8.28: A meshed supermarket domain represented by a planar straight-line graph: the mesh is generated by EikMesh. Its histograms of ρ_1, ρ_2 and θ are also depicted. A user specific edge length function $h_u(\mathbf{x}) = 0.25 + 0.4 \cdot |d_\Omega(\mathbf{x})|$ is used and combined by the local feature size lfs and the gradient limiting procedure presented in Section 8.6.

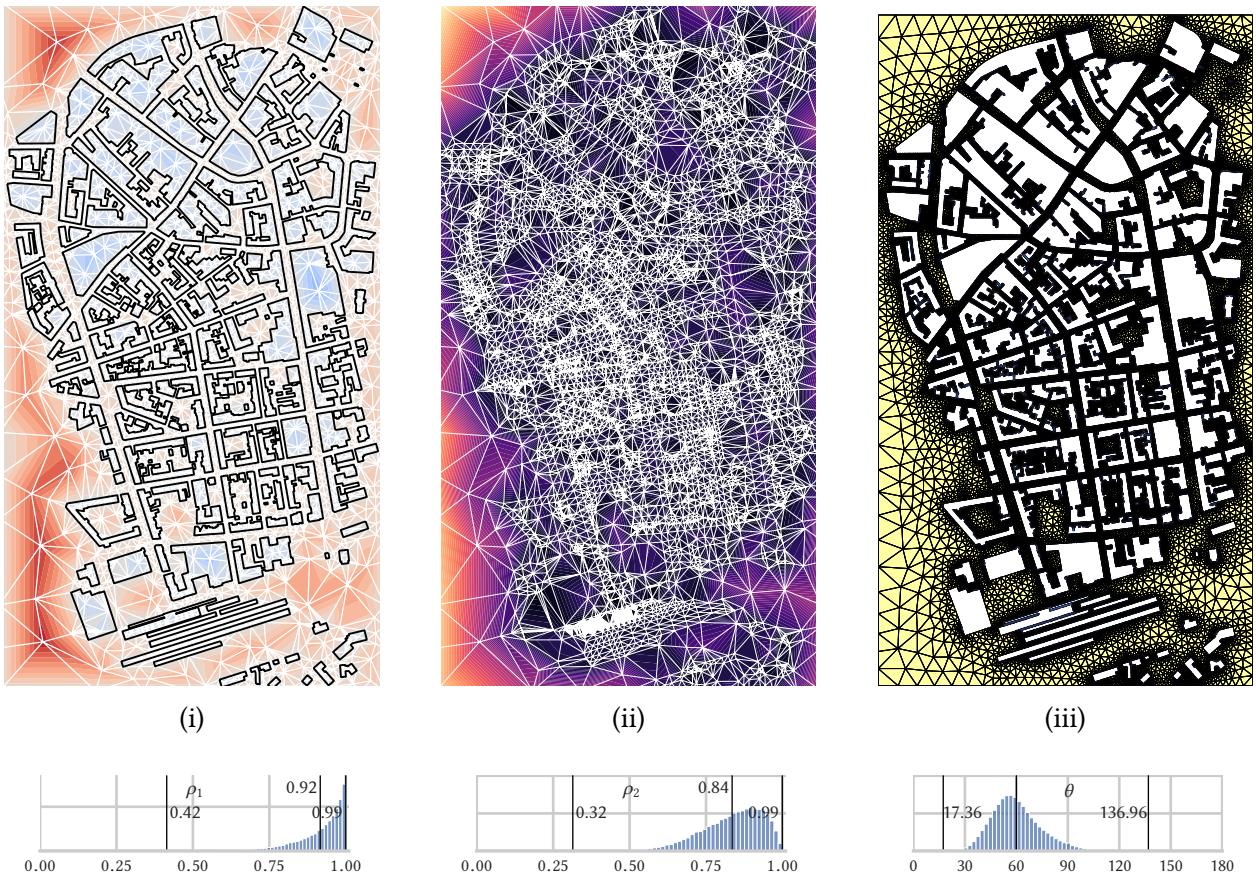


Figure 8.29: A meshed large-scale urban environment: the mesh generated by EikMesh for a large-scale scenario (iii) and its histograms of ρ_1, ρ_2 and θ. d_Ω (i) and h (ii) is approximated by using a coarse background mesh generated by Ruppert. No user specific edge length function is defined.

8.8.3 Execution time comparison

In the last part of this section, I compare the execution times of EikMesh with DistMesh. To do so, I choose a simple geometry defined by

$$d_\Omega(\mathbf{x}) = \text{abs}(0.7 - \|x\|) - 0.3 \tag{8.68}$$

and edge length functions

$$h(\mathbf{x}) = h_{\text{min}} + 0.4 \cdot \max\{-d_\Omega(\mathbf{x}), 0\} \tag{8.69}$$

with $0.2 \geq h_{\text{min}} \geq 0.02$. I use 100 improvement iterations. The exact speedup factors of such comparisons are not very meaningful, but it reveals the transfer of the reduced theoretical complexity to practical examples. Both algorithms are executed on my default workstation: Intel i5-7400 Quad-Core (3.50 GHz), 8 GB DDR4 SDRAM, and a graphics card NVIDIA GeForce GTX 1050 Ti / 4 GB GDDR5 VRAM. I use the Java version 11.0.2.

If we look at the qualities achieved by EikMesh, the first notable observation is that the mean quality ρ_1 is almost independent of $h_{\min}$. If the set of vertices is too small (because $h_{\min}$ is too large) the quality is below 0.95, but for $|V| \geq 2500$, the quality is approximately the same for all generated meshes, compare Fig. 8.30vi.

For DistMesh, the quality decreases with an increasing number of vertices which seems counterintuitive. However, this can be explained by the fixed number of improvement iterations. In my observation, DistMesh also achieves higher qualities if one increases the number of vertices, but it converges slower than EikMesh. For the tested examples, EikMesh achieves a similar quality without increasing the number of vertices. Again EikMesh achieves better qualities and far better minimum qualities, which stay above 0.5, compare Fig. 8.30vii.

The execution time of my EikMesh implementation is smaller than the execution time of my DistMesh implementation, because the sequential part, the Delaunay triangulation computation, is eliminated. Note that parallel algorithms for computing the Delaunay triangulation exist. Still, non-recursive flips (Section 8.2) can exploit parallelism without any complicated synchronization mechanisms. Furthermore, its theoretical time complexity is $O(n)$ instead of $O(n \log(n))$, where n is the number of vertices.

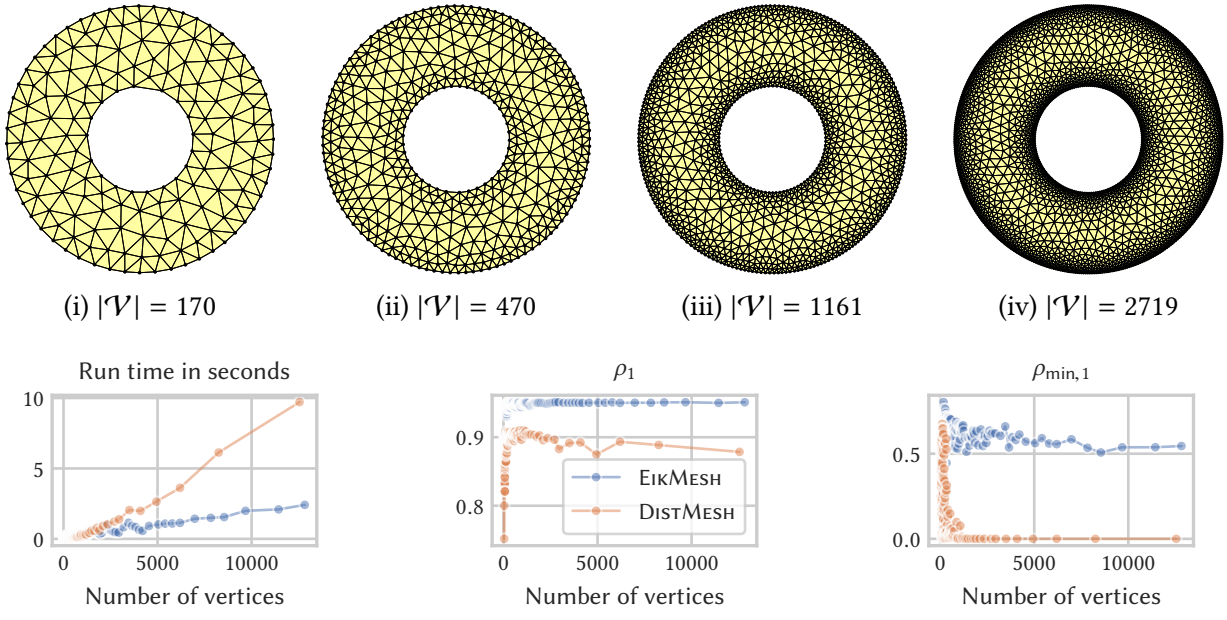


Figure 8.30: Performance comparison between EikMesh and DistMesh: different edge length functions lead to different meshes: the meshes displayed are generated by EikMesh (i)-(iv). Due to the parallel edge flip method, EikMesh runs faster than DistMesh (v). The quality ρ_1 achieved by EikMesh is approximately 0.95 and independent from the number of vertices (vi). For DistMesh, the quality ρ_1 slightly decreases with an increasing number of vertices. $\rho_{1,\min}$ achieved by EikMesh, stays above 0.5 and seems to be independent from the number of vertices. For DistMesh, there is at least one very poorly shaped element for almost any generated mesh.

8.9 Source code

The source code for all discussed algorithms and data structures is part of the open-source simulation framework Vadere [294]. More specifically, it is contained in the independent subproject called `VadereMeshing`. The source code of EikMesh can be found in the `eikmesh` package. `GenEikMesh.java` realizes the improvement phase of EikMesh while `GenUniformRefinementTriangulatorSFC.java` realizes its initialization. `EdgeLengthFunctionApprox.java` implements the element size function construction and `DistanceFunctionApproxBF.java` the construction of the distance function d_Ω.

8.10 Summary

In this chapter, I described EikMesh and all its ingredients. Since it is based on the forced-based smoothing technique of DistMesh, this chapter relied on the previous description of DistMesh.

In Section 8.1, I introduced local mesh operations that are executed by EikMesh. These operations realize simple topological changes that influence a small neighborhood of mesh elements.

In the next section, I showed how a non-recursive edge flip method replaces the computation of the Delaunay triangulation enforcing a slightly constrained vertex movement. This replacement decreases the time complexity of an improvement iteration by a factor of $O(\log(n))$.

I also explained why explicit geometry information such as fix points and constrained line-segments is beneficial for both meshing algorithms. Section 8.3 shows this by an example and explains how particular boundary points called slide points can replace the gradient dependent projection. These slide points guarantee boundary adherence, which is an important property of the constructed mesh, especially in the context of microscopic pedestrian dynamics.

A significant drawback of DistMesh, which I resolved, is the appearance of poorly shaped elements near the boundary. In Section 8.4, I described special local operations for boundary elements. They keep the quality of boundary elements high. I proposed different strategies which emerged from the logic of truss structure analogy of DistMesh.

In Section 8.5, I concluded the description of the core algorithm by presenting a new initialization phase. This phase generates the initial triangulation deterministically. A given triangulation consisting of two triangles is refined recursively. This strategy leads to a lot of high-quality elements and, therefore, generates a good starting point of the improvement procedure. Furthermore, triangles are spatially ordered based on the Sierpinski curve. This ordering is exploited so that objects of the data structure, which represent spatially close mesh elements, are likely to be closely aligned in the main memory. This property reduces the number of CPU stalls and translates to a shorter execution time for many algorithms executed on the mesh.

The following two sections discussed two of EikMesh's required inputs: the element size function and the distance function. I discussed the property of a desirable element size function and how one can compute such a function. I focused the discussion of the distance function on a planar straight-line graph (PLSG), because domains used in the context of microscopic pedestrian simulations can all be represented by PLSGs. Because the evaluation of the distance function d_Ω is a major computational bottleneck for the EikMesh algorithm, I discussed two strategies to approximate d_Ω on a coarse background mesh in Section 8.7.

In Section 8.8, I presented different generated meshes and compared the quality of their elements. Some discussed domains are artificial, others real-world examples from the field of fluid and pedestrian dynamics. I used the artificial examples to compare EikMesh with DistMesh. In the first one, I compared qualities, in the second one execution times. The other examples show EikMesh in practices. Each generated mesh consists of high-quality elements. Mainly the minimum quality stays above 0.4 for all examples, which illustrates the effectiveness of the special treatment of boundary elements. Furthermore, elements align perfectly with the boundary. I also gave an outlook into a distributed memory version of EikMesh by showing that it can be used to improve only small portions of a given mesh.

CHAPTER 9

Navigation field computation

> *"The secret for harvesting from existence the greatest fruitfulness and the greatest enjoyment is: to live dangerously!"*
>
> – Friedrich Nietzsche

Computation of navigation fields demands space discretization. A large number of discretization points lead to high computational costs. Usually, in an offline setting all input parameters are chosen before the simulation starts and they never change during the run. In that case, one can compute static navigation fields before the simulation starts, and their computational cost is less critical. However, in an online setting, where one feeds the running simulation with observations from the real world, new static fields may be required during a run. Consequently, long computation times would slow down the whole simulation. For dynamic navigation fields, a frequent re-computation of navigation fields demands efficient computations for a series of eikonal equations. Using a fine Cartesian grid for space discretization, the computational burden becomes intolerable for large domains and a real-time simulation impossible.

I identified two paths to accelerate the computation of navigation fields. The first one is to find a discretization of the spatial domain that requires a small number of vertices and gives a good approximation of the eikonal equation's solution. For a reasonable element size function h_u, EikMesh provides a high-quality unstructured mesh as a starting point. In Section 9.4.4, I establish a connection between h_u and the curvature of the solution of the eikonal equation. By using this connection, I introduce a new iterative eikonal solver that starts on a coarse mesh, which gets successively refined where needed.

The second path leads to a faster computation of the solution given a specific mesh. One can either choose the best possible solver or, if possible, introduce a new one. In Section 9.5, I introduce the InformedFastIterativeMethod that is specialized to solve a series of similar eikonal equations $\Phi_{\Gamma,0}, \Phi_{\Gamma,1}, \ldots, \Phi_{\Gamma,m}$, where two consecutive solutions $\Phi_{\Gamma,i}, \Phi_{\Gamma,i+1}$ are similar. It is, therefore, suitable to compute dynamic navigation fields.

These paths require a deep understanding of the solving process on an algorithmic level. Therefore, I start this chapter with the propagating wave analogy and a description of *upwind finite difference schemes* for Cartesian grids and unstructured meshes. The review in Section 9.3 completes the discussion of the algorithmic process. It includes ideas, strengths, and weaknesses of existing numerical methods.

9.1 The wavefront propagation analogy

In Section 3.3, I introduced the eikonal equation

$$\begin{aligned} \|\nabla\Phi_\Gamma(\mathbf{x})\| &= f(\mathbf{x})^{-1}, & \mathbf{x} &\in \Omega \\ \Phi_\Gamma(\mathbf{x}) &= 0, & \mathbf{x} &\in \Gamma \\ f(\mathbf{x}) &\geq 0, & \mathbf{x} &\in \Omega, \end{aligned} \tag{9.1}$$

which is a non-linear boundary value problem for a partial differential equation. Applications of the eikonal equation are numerous. It occurs in the field of computer graphics, image processing, computational geometry, physics, robotics, and medical image analysis. For example, in Section 8.6 and Section 8.7, I described how one can compute the local feature size for arbitrary geometries and the distance function d_Ω using the eikonal equation. In physics, the eikonal equation models wavefront propagation.

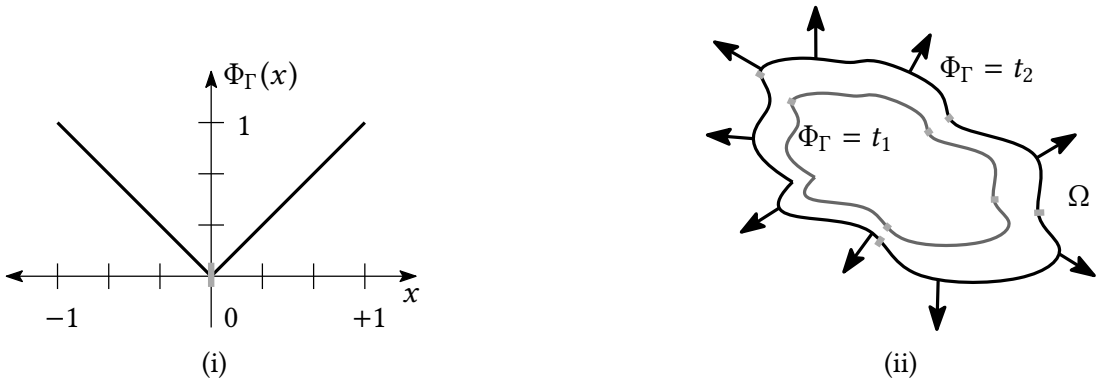
The eikonal equation solution gives us the travel time Φ_Γ of a wavefront propagating over the travel speed field f. For the particular case $f = 1$, the travel time $\Phi_\Gamma(\mathbf{x})$ is identical to the geodesic distance from $\mathbf{x}$ to Γ. Because information flows outwards following $\nabla\Phi_\Gamma$ on the optimal path, larger travel times depend on smaller ones. Consider, for example, the one-dimensional case with $\Omega = [-1, 1], \Gamma = \{0\}$ and the one-dimensional eikonal equation

$$\sqrt{\Phi_\Gamma(x)^2} = f(x)^{-1}. \tag{9.2}$$

Here information flows from 0 towards 1 and -1, compare Fig. 9.1i. Therefore, travel times $\Phi_\Gamma(x)$ for $x > 0$ are independent from $\Phi_\Gamma(x)$ for $x < 0$. Regardless of the travel speed function f, these values ($\Phi_\Gamma(x)$ for $x > 0$) depend on travel times of the left neighborhood, that is,

$$\Phi_\Gamma(x) = \lim_{h \to 0^+} \left(\Phi_\Gamma(x - h) + \frac{h}{f(x - h)} \right). \tag{9.3}$$

Therefore, we can split the wavefront into two independent parts. In higher dimensions multiple independent wave parts can be identified.



(i) (ii)

Figure 9.1: Wave propagation in one-dimensions with $f = 1$ (i), and two-dimensions (ii). One can partition the wavefront into multiple independent parts indicated by the gray cuts.

For a two-dimensional case, let us imagine throwing a stone into a lake. Let's say the stone hits the water at $t_0 = 0$ seconds. The wave created by the impact of the stone propagates outwards. Now let us pause the propagation at two specific moments t_1, t_2 with $t_1 < t_2$. Let us partition the wavefront at t_2, that is,

$$\{\mathbf{x} \in \Omega \mid \Phi_\Gamma(\mathbf{x}) = t_2\} \tag{9.4}$$

into multiple connected parts. Let C_2 be some connected part of this partition. Clearly, the travel time $\Phi_\Gamma(\mathbf{x})$ is smaller than t_2 for all positions $\mathbf{x}$ the wave already passed. Furthermore, for each connected set of points of the partition C_2, we can find a corresponding connected set of points C_1 of the wavefront at t_1 so that its propagation (information flow) covers C_2. We only have to backtrack the propagation. Consequently, we could compute the travel time for points in C_2 using only a portion of the information available at t_1. A sketch of the example is illustrated in Fig. 9.1ii.

This dependency relation is the basis of most efficient solvers. They try to emulate the (parallel) wave propagation, exploit the information flow, and introduce parallelism based on an independent relation.

9.2 Finite difference schemes

Numerical methods utilize *upwind finite difference schemes* to compute an approximation of eikonal equation's viscosity solution. Consider the one-dimensional case above.

9.2.1 Cartesian grid

In order to compute an approximation of the solution of Eq. (9.2), we partition the x-axis into a collection of grid points $x_i = (\Delta x \cdot i)$ with $i = -m, -(m-1), \ldots, (m-1), m$ such that $(m \cdot \Delta x) = 1$. By using a Taylor expansion (neglecting the remainder), we obtain the following system:

$$\begin{aligned} \phi_\Gamma(0) &= 0 \\ \frac{\phi_\Gamma(x_{i+1}) - \phi_\Gamma(x_i)}{\Delta x} &= f(x_i)^{-1} \text{ for } i \geq 0 \\ \frac{\phi_\Gamma(x_{i-1}) - \phi_\Gamma(x_i)}{\Delta x} &= f(x_i)^{-1} \text{ for } i \leq 0, \end{aligned} \tag{9.5}$$

where $\phi_\Gamma(x_i)$ approximates $\Phi_\Gamma(x_i)$. This upwind scheme emulates the flow of information. $\phi_\Gamma(x_1)$ can be computed if $\phi_\Gamma(x_0) = \Phi_\Gamma(0)$ is known, and $\phi_\Gamma(x_2)$ can be computed if we know $\phi_\Gamma(x_1)$ and so on. In one dimension, the wavefront can only approach x_i from one of two possible directions. The system above incorporates our knowledge about the direction. If we neglect this knowledge, we can solve for $\phi_\Gamma(x_i)$ by computing the travel time assuming the front approaches x_i from left and right. The minimum gives us the correct answer:

$$\phi_\Gamma(x_i) = \min\left\{\phi_\Gamma(x_{i-1}) + \frac{\Delta x}{f(x_i)}, \phi_\Gamma(x_{i+1}) + \frac{\Delta x}{f(x_i)}\right\}. \tag{9.6}$$

Therefore, the larger (approximated) gradient defines the value $\phi_\Gamma(x_i)$:

$$\max\left\{\frac{\phi_\Gamma(x_i)-\phi_\Gamma(x_{i-1})}{\Delta x},\frac{\phi_\Gamma(x_i)-\phi_\Gamma(x_{i+1})}{\Delta x}\right\}^2=\frac{1}{f(\mathbf{x}_{i,j})^2}. \tag{9.7}$$

In the two-dimensional case, a similar construction can be used: let $\mathbf{x}_{i,j}=(i\Delta x, j\Delta y)$ with $\Delta x=\Delta y$ be the points of a Cartesian grid, then we can derive an upwind-difference scheme that approximates partial derivatives of $\Phi_\Gamma(\mathbf{x}_{i,j})$ in two dimensions using a similar Taylor expansion:

$$\begin{aligned}\frac{\partial\Phi_\Gamma(\mathbf{x}_{i,j})}{\partial x}&\approx D_{i,j}^{\pm x}\mathbf{x}=\frac{\phi_\Gamma(\mathbf{x}_{i\pm1,j})-\phi_\Gamma(\mathbf{x}_{i,j})}{\pm\Delta x}\\ \frac{\partial\Phi_\Gamma(\mathbf{x}_{i,j})}{\partial y}&\approx D_{i,j}^{\pm y}\mathbf{x}=\frac{\phi_\Gamma(\mathbf{x}_{i,j\pm1})-\phi_\Gamma(\mathbf{x}_{i,j})}{\pm\Delta y}.\end{aligned} \tag{9.8}$$

We can solve the equation by using Godunov's scheme [151]

$$\max\{D_{i,j}^{-x}\mathbf{x},-D_{i,j}^{+x}\mathbf{x}\}^2+\max\{D_{i,j}^{-y}\mathbf{x},-D_{i,j}^{+y}\mathbf{x}\}^2=f(\mathbf{x}_{i,j})^{-2}, \tag{9.9}$$

or the scheme introduced by Osher and Sethian [215]:

$$\max\{D_{i,j}^{-x}\mathbf{x},0\}^2+\min\{D_{i,j}^{+x}\mathbf{x},0\}^2+\max\{D_{i,j}^{-y}\mathbf{x},0\}^2+\min\{D_{i,j}^{+y}\mathbf{x},0\}^2=f(\mathbf{x}_{j,j})^{-2}. \tag{9.10}$$

9.2.2 Unstructured mesh

To derive an upwind-difference scheme on an unstructured mesh $\mathcal{T}$, I follow the general approach presented by Sethian and Vladimirsky [263], that allows for a first- and second-order scheme. Another more geometrically oriented first-order approach was proposed by Kimmel et al. [151]. Since, for an unstructured mesh, there is no natural choice of the coordinate system, Sethian and Vladimirsky proposed to compute the gradient as a linear combination of k directional derivatives, where $\Omega\subset\mathbb{R}^k$. Let us consider a vertex $\mathbf{v}\in\tau$ of a k-simplex τ and its neighbors $\mathbf{v}_1,\ldots,\mathbf{v}_k\in\tau$. I define

$$\mathbf{p}_i=\frac{\mathbf{v}-\mathbf{v}_i}{\|\mathbf{v}-\mathbf{v}_i\|} \tag{9.11}$$

to be the unit vector pointing from $\mathbf{v}_i$ to $\mathbf{v}$. Let $u_i(\mathbf{v})$ be the directional derivatives towards $\mathbf{p}_i$, that is,

$$u_i(\mathbf{v})=\mathbf{p}_i\nabla\phi_\Gamma(\mathbf{v})\in\mathbb{R}. \tag{9.12}$$

Furthermore, let

$$\mathbf{u}=\begin{pmatrix}u_1(\mathbf{v})\\ \vdots\\ u_k(\mathbf{v})\end{pmatrix}\in\mathbb{R}^k\text{ and }\mathbf{P}=\begin{pmatrix}\mathbf{p}_1\\ \vdots\\ \mathbf{p}_k\end{pmatrix}\in\mathbb{R}^{k\times k}. \tag{9.13}$$

Then we generalize Eq. (9.12) to

$$\mathbf{u}=\mathbf{P}\nabla\phi_\Gamma(\mathbf{v}), \tag{9.14}$$

that leads to

$$\mathbf{P}^{-1}\mathbf{u}=\nabla\phi_\Gamma(\mathbf{v}). \tag{9.15}$$

Algorithm 14: SolveEikonal

Input: vertex $\mathbf{v}$
Output: q new value of $\phi_\Gamma(\mathbf{v})$

```
q ← φ_Γ(v);
if v ∉ Γ then
    foreach τ conncted to v do
        q ← min{solution of Eq. (9.18) for v and τ, q};
return q;
```

Following the definition of a k-simplex, $\mathbf{P}$ has to be a non-singular matrix. For example, in the two-dimensional space, τ is a triangle. Therefore, $\mathbf{P}$ consists of two linear independent vectors $\mathbf{p}_1$ and $\mathbf{p}_2$. Finally, we substitute $\nabla\phi_\Gamma(\mathbf{v})$ by $\mathbf{P}^{-1}\mathbf{u}$ in Eq. (9.1) to get

$$\mathbf{u}^T(\mathbf{P}\mathbf{P}^T)^{-1}\mathbf{u} = f(\mathbf{v})^{-2}. \tag{9.16}$$

To obtain the discretized equation, we replace each u_i with the corresponding difference approximation:

$$u_i(\mathbf{v}) \approx a_i\phi_\Gamma(\mathbf{v}) + b_i, \tag{9.17}$$

where b_i linearly depends on values of ϕ_Γ (and possibly of $\nabla\phi_\Gamma(\mathbf{v})$ for higher order schemes) at the vertex around $\mathbf{v}$. Then, the discretized version of Eq. (9.16) for $\mathbf{v}$ is the quadratic equation:

$$(\mathbf{a}^T\mathbf{Q}\mathbf{a})\phi_\Gamma(\mathbf{v})^2 + (2\mathbf{a}^T\mathbf{Q}\mathbf{b})\phi_\Gamma(\mathbf{v}) + (\mathbf{b}^T\mathbf{Q}\mathbf{b}) = f(\mathbf{v})^{-2}, \tag{9.18}$$

where $\mathbf{Q} = (\mathbf{P}\mathbf{P}^T)^{-1}$, $\mathbf{a} = (a_1, \ldots, a_k)^T$ and $\mathbf{b} = (b_1, \ldots, b_k)^T$. We derive a first-order scheme by using

$$u_i(\mathbf{v}) = \frac{\phi_\Gamma(\mathbf{v}) - \phi_\Gamma(\mathbf{v}_i)}{\|\mathbf{v} - \mathbf{v}_i\|} \tag{9.19}$$

such that

$$a_i = \frac{1}{\|\mathbf{v} - \mathbf{v}_i\|}, \quad b_i = -\frac{\phi_\Gamma(\mathbf{v}_i)}{\|\mathbf{v} - \mathbf{v}_i\|}. \tag{9.20}$$

Sethian and Vladimirsky [263] proposed to use $\nabla\phi_\Gamma(\mathbf{v}_i)$ for a second order scheme, that is,

$$u_i(\mathbf{v}) = 2\frac{\phi_\Gamma(\mathbf{v}) - \phi_\Gamma(\mathbf{v}_i)}{\|\mathbf{v} - \mathbf{v}_i\|} - \mathbf{p}_i\nabla\phi_\Gamma(\mathbf{v}_i), \tag{9.21}$$

such that

$$a_i = \frac{2}{\|\mathbf{v} - \mathbf{v}_i\|}, \quad b_i = -2\frac{\phi_\Gamma(\mathbf{v}_i)}{\|\mathbf{v} - \mathbf{v}_i\|} - \mathbf{p}_i\nabla\phi_\Gamma(\mathbf{v}_i). \tag{9.22}$$

$\nabla\phi_\Gamma(\mathbf{v}_i)$ has to be known at the when $\phi_i(\mathbf{v})$ will be computed. If this is not the case, one can fall back to the first-order scheme. One might argue that Φ_Γ is not differentiable everywhere, and to use higher-order schemes, Φ_Γ has to be sufficiently smooth. However, the authors in [263] argue that

> *"the fact that at some points $\nabla\Phi_\Gamma$ is undefined does not prevent us from using this approach: Φ_Γ is differentiable almost everywhere, and characteristics never emanate from the shocks, i. e., no information is created at the shock."*
> – Sethian and Vladimirsky et al. [263]

If $\phi_\Gamma(\mathbf{v}_1), \ldots, \phi_\Gamma(\mathbf{v}_k)$ are known, we can solve Eq. (9.18) that gives us $\phi_\Gamma(\mathbf{v})$ for a specific neighboring simplex. I assume that the wavefront propagates with a constant travel speed $f(\mathbf{v})$ through the simplex τ. To satisfy the upwind criterion, one accepts the computed value $\phi_\Gamma(\mathbf{v})$ if the update is coming from within τ. More precisely, $-\nabla\phi_\Gamma(\mathbf{v})$ has to lie inside the simplex [263]. This is the case if and only if $-\nabla\phi_\Gamma(\mathbf{v})$ is a linear combination of vectors $\mathbf{p}_1, \ldots, \mathbf{p}_k$, where each coefficient is positive.

Lemma 9.1 (gradient direction [263]). *The negated gradient $-\nabla\Phi_\Gamma(\mathbf{v})$ points inside the simplex if and only if all components of $\mathbf{Qu}$ are positive.*

Proof. This can be seen by substituting $\mathbf{u}$ by $\mathbf{P}\nabla\Phi_\Gamma(\mathbf{v})$ using Eq. (9.14):

$$\begin{aligned}\mathbf{Qu} &= \mathbf{QP}\nabla\phi_\Gamma(\mathbf{v}) = (\mathbf{PP}^T)^{-1}\mathbf{P}\nabla\phi_\Gamma(\mathbf{v}) \\ &= (\mathbf{P}^T)^{-1}\mathbf{P}^{-1}\mathbf{P}\nabla\phi_\Gamma(\mathbf{v}) = (\mathbf{P}^T)^{-1}\nabla\phi_\Gamma(\mathbf{v}).\end{aligned} \tag{9.23}$$

Therefore,

$$\mathbf{P}^T\mathbf{Qu} = \mathbf{P}^T(\mathbf{P}^T)^{-1}\nabla\phi_\Gamma(\mathbf{v}) = \nabla\phi_\Gamma(\mathbf{v}). \tag{9.24}$$

Using Eq. (9.24), we can see that $\nabla\phi_\Gamma(\mathbf{v})$ is a linear combination of $\mathbf{p}_1, \ldots, \mathbf{p}_k$. Consequently $-\nabla\phi_\Gamma(\mathbf{v})$ points inside the simplex if and only if all components of $\mathbf{Qu}$ are positive. ■

The construction is used to compute the travel time if the wavefront comes from within a specific k-simplex. However, a vertex $\mathbf{v}$ is surrounded by multiple simplices. Like the previous one dimensional case, we have to find the *defining simplex*, that is, the simplex from which the wavefront arrives first at $\mathbf{v}$. Therefore, one computes $\phi_\Gamma(\mathbf{v})$ for each simplex neighboring $\mathbf{v}$ and picks the smallest value, compare Algorithm 14. This strategy is consistent with Godunov's method on the Cartesian grid.

9.2.3 Dealing with obtuse angles

A crucial assumption for most numerical solvers, which I discuss in the next section, is causality. The update for $\phi_\Gamma(\mathbf{v})$ comes from some *defining triangle* τ connected to $\mathbf{v}$ assuming the wavefront already reached $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \tau$. If the wavefront moves with constant speed $f(\mathbf{v})$ within τ, as we assume, causality is guaranteed for acute triangles. In the extreme case of 90 degrees, the

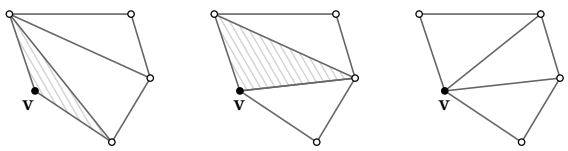


Figure 9.2: Elimination of an obtuse angle at $\mathbf{v}$: virtual simplices that are only used to compute $\Phi_\Gamma(\mathbf{v})$ are introduced to deal with obtuse angles.

Figure 9.3: Causality violation caused by obtuse angles: for obtuse angles (i) it is possible that the wavefront (approximation), indicated by the black line, reaches either $\mathbf{v}_1$ or $\mathbf{v}_2$ but not both before it reaches $\mathbf{v}$. For an acute triangulation this is impossible (ii).

wavefront might arrive at $\mathbf{v}$ and $\mathbf{v}_1$ at the same time, compare Fig. 9.3ii. For larger angles, causality might be violated. In this case, it is possible that the wavefront reaches either $\mathbf{v}_1$ or $\mathbf{v}_2$ but not both before it arrives at $\mathbf{v}$. The situation is illustrated in Fig. 9.3i. To deal with an obtuse angle at $\mathbf{v}$, I replace the simplex τ that consists of an obtuse angle at $\mathbf{v}$ by a set of simplices $\tau_1, \dots, \tau_m$ so that

$$\bigcup_{i=1}^{m} |\tau_i| = |\tau| \ \wedge \ \forall i, j : i \neq j \iff \tau_i \cap \tau_j = \emptyset \tag{9.25}$$

holds. $|\tau|$ is the space covered by τ. Thereby, I follow the idea presented in [263]. Let $\tau = \mathbf{v}\mathbf{v}_1\mathbf{v}_2$ be the simplex with an obtuse angle at $\mathbf{v}$ and $\mathbf{u}$ be the vertex opposite of the edge $\{\mathbf{v}_1, \mathbf{v}_2\}$. Then I replace τ by $\tau_1 = \mathbf{v}\mathbf{v}_1\mathbf{u}$ and $\tau_2 = \mathbf{v}\mathbf{v}_2\mathbf{u}$. I repeat this splitting recursively until all angles at $\mathbf{v}$ are acute, compare Algorithm 15 and Fig. 9.2. The resulting structure is saved so that I do not have to repeat this process.

Algorithm 15: ConstructVirtualSimplices

Input: triangle $\tau = \mathbf{v}\mathbf{v}_1\mathbf{v}_2$, vertex $\mathbf{v} \in \tau$, list $\mathcal{L}$
Output: $\mathcal{L}$ list of virtual simplices

```
if there is an obtuse angle at v then
    τ1 ← vv1u, τ2 ← vv2u;
    L ← L ∪ τ1, L ← L ∪ τ2;
    ConstructVirtualSimplices(τ1,v,L);
    ConstructVirtualSimplices(τ2,v,L);
return L;
```

9.3 Review of numerical methods

There are many different numerical methods to compute the viscosity solution of the eikonal equation 9.1. The most known ones are the RouyAndTourin algorithm [238], the FastMarchingMethod (FMM) [292, 151], the FastSweepingMethod (FSM) [331], the FastIterativeMethod (FIM) [135] and the HeatMethod (HM) [55].

The HeatMethod stands out, because it works inherently differently compared to the other methods. Where the HeatMethod was developed without a specific discretization in mind, all other methods were initially developed to operate on a Cartesian grid. The HeatMethod is based

on a heat diffusion analogy and solves the eikonal equation *indirectly* by combining the heat equation with the Poisson equation. In contrast, the other methods are based on the wavefront analogy and use an upwind-difference scheme to solve the eikonal equation in a *direct* way.

In principle, direct methods only differ in the order, in which the local solution of vertices is computed. The order introduced by the FastMarchingMethod is optimal. But it comes at the cost of an ordered data structure that hinders parallelization. The FastSweepingMethod and the FastIterativeMethod disregard the strict order of the FMM in favor of an unsorted list. Thereby they benefit from parallelization. Because the relaxed order is most undoubtedly suboptimal, the FIM might compute $\phi_\Gamma(\mathbf{v})$ multiple times before it converges. Therefore, we trade 'more work' for parallelism. Each of the direct methods were later extended to support unstructured meshes. We will not find a clear overall winner since the solver's performance depends on the input, that is, the domain Ω and f. I did not conduct an experimental analysis to find the best solver for each situation in this work. Instead, I looked at results in the literature and drew my conclusions based on the domain Ω and travel speed functions f used in the context of pedestrian dynamics.

9.3.1 The Heat Method

Before presenting methods that rely on the upwind difference scheme and compute the solution in a *direct* way, I review another approach: the HeatMethod. Crane et al. [55] proposed a very different technique to compute the geodesic distance *indirectly*. First, they solve the heat equation to compute a vector field X that approximates $\nabla\Phi_\Gamma$. In the second step, they solve the Poisson equation to compute Φ_Γ.

Let us imagine $\Gamma \subset \mathbb{R}^2$ to be a heat source that emits heat particles. Over time, heat particles spread out over the whole domain. If we stop the process after a short amount of time $t \to 0$, particles at distant positions likely traveled on the shortest possible path. Consequently,

$$X = \frac{-\nabla q_\Gamma}{\|\nabla q_\Gamma\|} \approx \frac{\nabla\Phi_\Gamma}{\|\nabla\Phi_\Gamma\|} = \frac{1}{f}, \tag{9.26}$$

where q_Γ is the heat distribution after a short time t and $f = 1$. Then by solving the Euler-Lagrange equation

$$\nabla^2\Phi_\Gamma = \nabla \cdot X, \tag{9.27}$$

Crane et al. [55] suppose to find the closest scalar travel time Φ_Γ.

In general, the vector fields' elements $-\nabla q_\Gamma$ and $\nabla\Phi_\Gamma$ point approximately in the same direction, but their magnitudes are different. Therefore, the authors assume the particular case of the eikonal equation for which

$$f = 1, \tag{9.28}$$

so that

$$\|\nabla\Phi_\Gamma\| = 1. \tag{9.29}$$

Consequently, we can normalize $-\nabla q_\Gamma$ to get an approximation of $\nabla\Phi_\Gamma$.

Computing the geodesic distance on heat flow is an interesting approach. It benefits from well-studied and highly optimized solvers for the heat and Poisson equation, respectively. Crane et al. [55] claims that

> *"geodesic distance is updated an order of magnitude faster than with state-of-the-art methods [such as the FMM], while maintaining a comparable level of accuracy."*
> – Crane et al. [55]

However, the requirement of Eq. (9.28) disqualifies the method for all kind of static and dynamic navigation fields discussed in Chapter 3. Additionally, investigations by Mayr [197] revealed problems with accuracy and robustness of the HeatMethod. The computed approximation of the geodesic distance depends on the choice of many parameters. Mayr points out a missing or unclear theoretical background of the method, especially for closed surfaces. Convergence for non-smooth boundaries is only shown for "best-case examples [where] the impact of the boundary condition is not present. [. . .] These examples do not replace a formal proof of convergence which is missing" [197]. Mayr analyzed the convergence behavior and concluded that

> *"the numerical solution of the HeatMethod does not converge in general".*
> – Mayr [197]

Therefore, the HeatMethod is not suitable to compute navigation fields for pedestrian simulations.

9.3.2 A Gauss-Jacobi method

In [238], Rouy and Tourin introduced a somewhat naive Jacobi-iteration to solve the eikonal equation on a Cartesian grid – it also works on unstructured meshes. Basically, RouyAndTourin solves Eq. (9.9) for iteration i at each grid point (or vertex) by using values computed in the previous iteration. The algorithm stops if the solution converges. The method (Algorithm 16) is slow since each vertex $\mathbf{v}$ has to be revisited several times before the numerical solution settles down [106]. Furthermore, the method does not take advantage of the information propagation of the problem. In the worst-case after each iteration only one vertex converges. Therefore, the

Algorithm 16: RouyAndTourin

Input: triangulation $\mathcal{T}$, spatial destination Γ, spatial domain Ω
Output: ϕ_Γ solution of Eq. (9.1)

$\phi^0_\Gamma(\mathbf{v}) \leftarrow \phi_0(\mathbf{v})$ for all $\mathbf{v} \in \Gamma$;
$\phi^0_\Gamma(\mathbf{v}) \leftarrow \infty$ for all $\mathbf{v} \notin \Gamma$;
$i \leftarrow 0$;
do
  $i \leftarrow i + 1$;
  foreach vertex $\mathbf{v}$ of the mesh $\mathcal{T}$ **in parallel do**
    `// use the values` ϕ^{i-1}_Γ `of the previous iteration`
    $\phi^i_\Gamma(\mathbf{v}) \leftarrow$ SolveEikonal($\mathbf{v}$);
while $\exists \mathbf{v} \in \mathcal{T} : \phi^i_\Gamma(\mathbf{v}) \neq \phi^{i-1}_\Gamma(\mathbf{v})$;
$\phi_\Gamma(\mathbf{v}) \leftarrow \phi^i_\Gamma(\mathbf{v})$ for all $\mathbf{v} \in \mathcal{T}$;
return ϕ_Γ;

theoretical time complexity is $O(n^2)$, where n is the number of vertices. Because the computation of a new value depends only on the values of the previous iteration, we can compute each value in parallel. Even though the method can be implemented for massively parallel machines, it is inefficient in theory and practice.

9.3.3 The Fast Marching Method

The FastMarchingMethod might be one of the most widely applied methods in computational science. One reason for its popularity is that its run time is independent of f and the domain. It was independently developed by Tsitsiklis [292] and Kimmel and Sethian [151] and considers vertices in an order consistent with the way the wavefront propagates, that is, consistent with the Huygens principle. The FMM exploits the monotonicity of Φ_Γ. During the algorithm execution, vertices are divided into three sets:

(i) *unreached*: not yet considered vertices,

(ii) *burning*: vertices of the propagating front,

(iii) *burned*: considered vertices.

Together, burning vertices form the so-called *narrow band*. The FMM enforces the order by a heap $\mathcal{H}$ sorted by the travel time of its elements. The heap represents the narrow band. Consequently, the FMM changes the state of a vertex from *burning* to *burn* only if it is the smallest element in $\mathcal{H}$. When the wavefront reaches a vertex $\mathbf{v}$, the vertex changes its state from *unreached* to *burning* and becomes part of the narrow band $\mathcal{H}$. While $\mathbf{v}$ is *burning*, its value can change whenever a *burning* neighboring burns out, that is, becomes *burned*.

Inserting and removing elements to and from $\mathcal{H}$ requires $O(\log(m))$ time, where $m < n$ is the number of elements in $\mathcal{H}$. Since after each iteration of the while-loop (Line 5 in Algorithm 17) one element is removed from the heap $\mathcal{H}$ and will not be added again, the time complexity of the FMM is $O(n \log(n))$.

Algorithmically, the FMM is very similar to Dijkstra's algorithm [71]. It is inherently sequential and depends on a heap data structure. However, the computation of 'distances' is different. While Dijkstra's algorithm propagates information only across edges, the FMM propagates information across simplices, in our case triangles. Even though other methods such as the FastSweeping-Method are faster in theory, the FastMarchingMethod is still one of the most efficient methods for many practical problems, especially in a single core setup.

9.3.4 The Fast Sweeping Method

The next important method, the FastSweepingMethod (FSM) [331], introduced seven years later, improves the Gauss-Jacobi method (RouyAndTourin algorithm) by using a Gauss-Seidel update scheme. Instead of the parallel update order of RouyAndTourin algorithm or the strict update order consistent with the wave propagation, the FastSweepingMethod uses predefined sweeping directions. For example, in two-dimensions and a Cartesian grid, the update for a grid point can come from top-right, top-left, bottom-left, and bottom-right. In that sense, it is the

Algorithm 17: FastMarchingMethod

Input: triangulation $\mathcal{T}$, spatial destination Γ, spatial domain Ω
Output: ϕ_Γ solution of Eq. (9.1)

$\phi_\Gamma(\mathbf{v}) \leftarrow \phi_0(\mathbf{v})$ for all $\mathbf{v} \in \Gamma$;
$\phi_\Gamma(\mathbf{v}) \leftarrow \infty$ for all $\mathbf{v} \notin \Gamma$;
$\mathcal{B} \leftarrow \emptyset$ // set of burned vertices
$\mathcal{H} \leftarrow \{(\mathbf{v}, \phi_\Gamma(\mathbf{v})) \mid \mathbf{v} \in \Gamma\}$ // set of burning vertices
while $\mathcal{H} \neq \emptyset$ **do**
    $(\mathbf{v}, \phi_\Gamma(\mathbf{v})) \leftarrow \min(\mathcal{H})$;
    foreach neighbor $\mathbf{u}$ of $\mathbf{v}$ with $\mathbf{u} \notin \mathcal{B}$ **do**
        $\mathcal{H} \leftarrow \mathcal{H} \setminus (\mathbf{u}, \phi_\Gamma(\mathbf{u}))$;
        $\phi_\Gamma(\mathbf{u}) \leftarrow$ SolveEikonal($\mathbf{u}$);
        $\mathcal{H} \leftarrow \mathcal{H} \cup (\mathbf{u}, \phi_\Gamma(\mathbf{u}))$;
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \{(\mathbf{v}, \phi_\Gamma(\mathbf{v}))\}$;
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathbf{v}\}$;
return ϕ_Γ;

middle ground between an arbitrary and strict update order. The method is motivated by the observation that information propagates alongside a finite number of characteristics (directions) for a discrete mesh. The alternating Gauss-Seidel update speeds up convergence compared to RouyAndTourin algorithm. Since it does not rely on a sorting data structure, it is, in theory, faster than the FastMarchingMethod.

Let Ω be a square centered at $\Gamma = (0, 0)$ and let $f = 1$ inside Ω, then four sweeps suffice to solve the eikonal equation. In that case, the FSM outperforms the FMM, because the wavefront does not change its direction and each characteristic curve is a straight line, compare Fig. 9.4ii.

In general, the FSM performs well if the maximum curvature for any characteristic curve of the eikonal equation is small. Lemma 9.2 connects this curvature to f. Therefore, for a specific dimension, the number of iterations is independent of the number of vertices but depends on the travel speed function f.

Lemma 9.2 (curvature bound [331]). *The maximum curvature for any characteristic curve of the eikonal equation is bounded by*

$$\max_{\mathbf{x} \in \Omega} \left\| \frac{\nabla f(\mathbf{x})}{f(\mathbf{x})} \right\|. \tag{9.30}$$

Consequently, "the algorithm is optimal in the sense that a finite number of iterations is needed" [331]. The theoretical time complexity of the FSM is $O(n)$. If we refine our mesh further and further, the FSM will eventually be faster than the FMM. However, in practice, the FSM performs poorly for complex geometries, because the hidden constant is large. Therefore, many sweeps are required.

> *"The more complicated the domain is, the better FMM performs with respect to FSM. Indeed, while FMM continually advances the wavefront, FSM has to do another set of sweeps every time the direction of propagation changes."* – Gremaud and Kuster [106]

Algorithm 18: FastSweepingMethod

Input: triangulation $\mathcal{T}$, spatial destination Γ, spatial domain Ω
Output: ϕ_Γ solution of Eq. (9.1)

```
1  φ⁰_Γ(v) ← φ_0(v) for all v ∈ Γ;
2  φ⁰_Γ(v) ← ∞ for all v ∉ Γ;
3  i ← 0;
4  do
5      i ← i + 1;
6      foreach for all sweeping directions do
7          foreach vertex v in order of the sweeping direction do
8              φ^i_Γ(v) ← SolveEikonal(v);
               // use the values φ^i_Γ of the current iteration
9  while ∃v ∈ T : |φ^i_Γ(v) − Φ^{i−1}_Γ(v)| > ε;
10 φ_Γ(v) ← φ^i_Γ(v) for all v ∈ T;
11 return φ_Γ;
```

In [103], we find an excellent experimental study that reveals the method's poor performance for important practical examples. I suggest using the FSM only for specific problems for that we know that the propagating wavefront changes directions only a few times.

9.3.5 The Fast Iterative Method

In [135], Jeong and Whitaker proposed the FastIterativeMethod (FIM) to solve the eikonal equation efficiently on parallel architectures. The FIM combines the advantages of the parallel nature of the RouyAndTourin algorithm with the information propagation of the FastMarchingMethod. The FIM's main idea is to update *burning* vertices of the *narrow band* in parallel without maintaining computationally expensive data structures. Jeong designed the method to run on a GPU. Therefore, he aimed to increase parallelism rather than achieve algorithmic optimality. Thus its worst-case performance may vary depending on the complexity of the input.

For the FIM, the heap $\mathcal{H}$ of the FMM representing the *narrow band* is replaced by an unsorted list $\mathcal{L}$. It utilizes a Gauss-Jacobi update scheme but only for vertices of the narrow band. For each iteration, new values are computed for all vertices in $\mathcal{L}$ in parallel. If the value difference is less than some small threshold ϵ, the neighbors are evaluated and added to the *narrow band* $\mathcal{L}$, if their value is improved (Line 14). Vertices are only removed from the *narrow band*, if they have converged (Algorithm 19, Line 9).

If the angle between the wavefront direction and narrow band's advancing direction is small, vertices converge fast (possibly in a single update). If the angle is large, information propagates through the narrow band, and multiple updates are required. Since new vertices are added during the information propagation within the narrow band, the *narrow band* is in general thicker than the narrow band of the FMM [135]. Furthermore, if the wavefront changes direction, vertices that have already been converged may be added again into the narrow band $\mathcal{L}$ – they *burn* again until new information propagates to them.

Algorithm 19: FastIterativeMethod

Input: triangulation $\mathcal{T}$, spatial destination Γ, spatial domain Ω
Output: ϕ_Γ solution of Eq. (9.1)

$\phi_\Gamma(\mathbf{v}) \leftarrow \phi_0(\mathbf{v})$ for all $\mathbf{v} \in \Gamma$;
$\phi_\Gamma(\mathbf{v}) \leftarrow \infty$ for all $\mathbf{v} \notin \Gamma$;
$\mathcal{L} \leftarrow \{(\mathbf{v}, \phi_\Gamma(\mathbf{v})) \mid \mathbf{v} \in \Gamma\}$ `// set of burning vertices`
while $\mathcal{L} \neq \emptyset$ **do**
    foreach $(\mathbf{v}, \phi_\Gamma(\mathbf{v})) \in \mathcal{L}$ **in parallel do**
        $p \leftarrow \phi_\Gamma(\mathbf{v})$;
        $q \leftarrow$ SolveEikonal($\mathbf{v}$);
        $\phi_\Gamma(\mathbf{v}) \leftarrow q$;
        if $|p - q| < \epsilon$ **then**
            foreach neighbor $\mathbf{u}$ of $\mathbf{v}$ **do**
                if $(\mathbf{u}, \phi_\Gamma(\mathbf{u})) \notin \mathcal{L}$ **then**
                    $p \leftarrow \phi_\Gamma(\mathbf{u})$;
                    $q \leftarrow$ SolveEikonal($\mathbf{u}$);
                    if $p > q$ **then**
                        $\phi_\Gamma(\mathbf{u}) \leftarrow q$;
                        $\mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathbf{u}, \phi_\Gamma(\mathbf{u}))\}$;
            $\mathcal{L} \leftarrow \mathcal{L} \setminus \{(\mathbf{v}, \phi_\Gamma(\mathbf{v}))\}$;
return ϕ_Γ;

Assuming the narrow band contains all the mesh vertices, the method transforms into the RouyAndTourin algorithm. Therefore, the theoretical time complexity is at least $O(n^2)$. Jeong and Whitaker [135] proved the method's convergence but did only provide an experimental performance analysis. Their analysis suggests that the (sequential) FIM outperforms the FSM for all cases but can be slower than the FMM if the wavefront changes its direction several times. Later Gómez et al. [103] reported the same tendency, that is, the FIM is only slower than the FSM for the ideal case mentioned in Section 9.3.4. Similar to the number of sweeps of the FSM, the number of times a vertex is added to the narrow band depends on f. It is independent of the number of vertices. Therefore, the theoretical time complexity of FIM seems also to be $O(n)$ as claimed by [103]. However, to my best knowledge, there is no formal proof for this claim.

Especially for small dimensions, for many problem instances, the heap of the FMM contains only a tiny portion of all vertices. More precisely, the thickness of the narrow band of the FMM is approximately one. Consequently, the (sequential) FMM performs best if the front stays small during the computation. That is the case if the 'length' of each level set

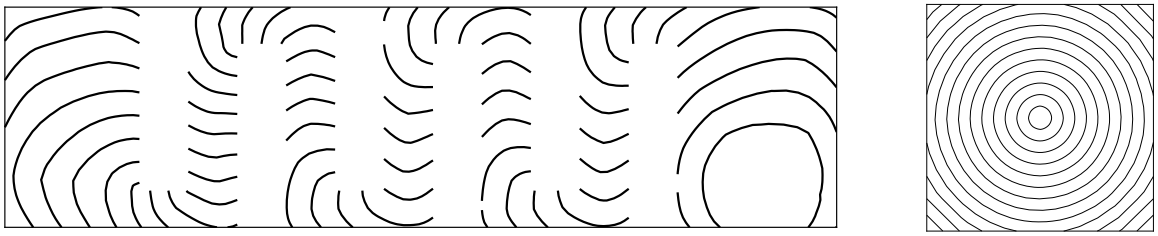
$$L_t = \{\mathbf{x} \in \Omega \mid \Phi_\Gamma(\mathbf{x}) = t\} \tag{9.31}$$

is small. Consider, for example, the multi-barrier domain depicted in Fig. 9.4i. For such cases the FMM outperforms the FIM in a single core setup.

Jeong and Whitaker [135] give us some advice to decide when to use the FIM instead of the FMM for a single-core setup: let c_1 be the cost to solve Eq. (9.18), c_2 be the cost for the heap update operation, $\overline{m}_{\text{FMM}}$ and $\overline{m}_{\text{FIM}}$ be the average operations per node for the FMM and FIM respectively. Furthermore, let $\bar{s}_{\mathcal{H}}$ be the average heap size, then if

$$\overline{m}_{\text{FMM}} < \overline{m}_{\text{FIM}} \left(1 + \frac{c_2}{c_1} \log_2(\bar{s}_{\mathcal{H}})\right) \tag{9.32}$$

the FIM should be chosen. We might predict upper bounds for all required values, but a tight upper bound requires information we might only know after we already solved the equation. Let us, for a moment, ignore the synchronization required whenever we manipulate $\mathcal{L}$. Then, if the number of processing units is larger than the maximal size of the narrow band $\mathcal{L}$, the FIM eventually outperforms the FMM. Therefore, I suggest considering the FIM in a multi-core setup.



(i) (ii)

Figure 9.4: A rather small wavefront scenario (i) and a large wavefront scenario (ii).

9.3.6 Extensions

After their initial presentation, researchers extended the FMM, FSM, and FIM without changing their underlying basis. The current research trend focuses mainly on

(1) supporting additional discretization types such as unstructured meshes or point clouds,

(2) developing parallel algorithms, and

(3) improving the efficiency of solvers.

Supporting new discretization types requires new local solvers. Two years after the introduction of the FastMarchingMethod [292, 151], Sethian and Vladimirsky [263] developed the local solver on unstructured for k-dimensional meshes presented in the previous section. In [73], Droske et al. extended the FMM to work on quadtrees. The quadtree is computed before solving the equation. Covello and Rodrigue [53] proposed an adaptive mesh that is refined as the solution is computed. The idea is to refine in areas where curvature κ of Φ_Γ is large. To compute the local curvature around some point $\mathbf{x}$, the travel time in that region has to be known. Therefore, the proposed algorithm

(1) rolls back the narrow band if the curvature is too high,

(2) refines the mesh,

(3) and continues the FMM.

Two years earlier the same authors developed a generalized front marching algorithm for vertices located on highly distorted meshes or vertices that are randomly distributed [52]. In [224, 225], Qian et al. extended the FastSweepingMethod to two- and three-dimensional unstructured triangular meshes. Similarly, Fu et al. [87, 88] extended the FIM to support the same types of spatial discretization. In the introduction of the FMM, Tsitsiklis [292] already proposed a $O(n)$ approach using a bucket data structure along with his $O(n \log(n))$ Dijkstra-like method similar to [151].

Kim [150] introduced the GroupMarchingMethod (GMM) which can be seen as an extension of the FMM. The algorithm was originally designed for Cartesian grids. Instead of only updating one point of the narrow band, a subset G of points is updated. A two-pass update within G ensures that information propagates through the subset. The GMM ensures that all points in G are independent from all *burning* and *unreached* point not in G. If $h_{\min}$ is the smallest edge of the mesh or Cartesian grid, we know that the distance between each pair of points is at least $h_{\min}$. Let $\phi_\Gamma(\mathbf{v}_{\min})$ be is the smallest travel time within the narrow band, then

$$\phi_\Gamma(\mathbf{v}) \geq \phi_\Gamma(\mathbf{v}_{\min}) + \frac{h_{\min}}{f_{\max}} \text{ with } f_{\max} = \max_{\mathbf{x}\in\Omega} f(\mathbf{x}) \tag{9.33}$$

holds for all *unreached* points $\mathbf{v}$. Consequently, we can update all points of following subset G of the narrow band

$$G = \left\{ \mathbf{v} \in \mathcal{H} \;\middle|\; \phi_\Gamma(\mathbf{v}) \leq \phi_\Gamma(\mathbf{v}_{\min}) + \frac{h_{\min}}{f_{\max}} \text{ with } f_{\max} \right\} \tag{9.34}$$

without worrying about violating causality. The GMM updates in each iteration $i = 0, 1, \ldots$ a group G of points (contained in the current narrow band) for that

$$i \cdot \frac{h_{\min}}{f_{\max}} \leq \phi_\Gamma(\mathbf{v}) \leq (i+1) \cdot \frac{h_{\min}}{f_{\max}} \tag{9.35}$$

holds. Like the FIM, the GMM replaces the heap $\mathcal{H}$ by a linear list $\mathcal{L}$. For each iteration, the GMM traverses this list in reversed and normal order, and each point with a small enough current travel time ϕ_Γ is updated (twice). This ensures that information within the group G propagates to all members of the group. Kim [150] numerically verified that the GMM performs with the computation cost $\mathcal{O}(n)$. The author presented experimental results for (stepwise) linear travel speed functions but, to the best of my knowledge, there is no formal proof. To find G, we have to filter $\mathcal{L}$ by iterating over it. In the worst case $\mathcal{L}$ contains $\mathcal{O}(n)$ elements. Therefore, it seems that we get rid of the $\log(n)$ factor by introducing a new one, such that the time complexity is $\mathcal{O}(n^2)$. Even if the GMM is a $\mathcal{O}(n)$ method, for most travel speed functions, "it has a high overhead in the form of keeping track of the group and determining which members of the group are to be updated" [140].

Jones et al. [140] proposed the SimplifiedFastMarchingMethod (SFMM), a very simple but nonetheless effective improvement of the FMM. Instead of removing and reinserting elements to the narrow band if their value is updated (Lines 8 and 10 of Algorithm 17), the authors suggest to just insert a new element. Therefore, they allow multiple elements for the same vertex contained in $\mathcal{H}$, and a vertex might be *burning* and *burned* at the same time. Since the smallest element will be popped first, all consecutive elements can be ignored. In other words, if a vertex $\mathbf{v}$ is *burned*, all elements $(\mathbf{v}, \phi_\Gamma(\mathbf{v}))$ in $\mathcal{H}$ will be ignored. SFMM avoids the decrease key operation of the heap and executes a push operation instead. Both have the same theoretical time complexity of $\mathcal{O}(\log(n))$. However, the hidden constant for the decrease key operation is larger. In practice, the SFMM runs faster even though its adaptation slightly increases the heap size $\mathcal{H}$ [103].

The UntidyFastMarchingMethod (UFMM) [324, 230] is equivalent to Algorithm 17 but the heap is replaced by the untidy priority queue which reduces the time complexity to $\mathcal{O}(n)$. The method assumes that f is bounded. The sorted untidy priority queue consists of multiple unsorted buckets. Each bucket contains an (unsorted) list but guarantees that all values $\phi_\Gamma(\mathbf{v})$ of its elements $\mathbf{v}$ are within a narrow interval. Given a specific travel time, the corresponding bucket can be addressed in $\mathcal{O}(1)$ time. The sorted queue is circular, meaning that the first bucket represents the last bucket if it becomes empty. Popping the smallest element from the queue is equal to popping some arbitrary element from the first bucket. Since this element will most certainly not be the smallest element, additional errors are introduced. Therefore, the result is not identical to the one computed by the FMM. It accumulates additional errors. However, they are bounded by the same order. Compared to the bucket based approach of Tsitsiklis, the UFMM does not require a new type of discretization.

In 2003, Herrmann [121] introduced a domain decomposition parallelization of the FastMarchingMethod. The idea is to assign each spatial sub-domain to a specific processing unit. Each processor manages its own priority queue, which is essentially a set of the partitioned narrow band. Vertices at the boundary of each sub-domain have to be copied and synchronized. Processors might idle if the wavefront had not reached a specific sub-domain yet or already propagated completely over it. A rollback mechanism revokes all previously accepted vertices' valid status

whenever the value of a ghost vertex decreases. For large wavefronts, this approach achieves reasonable speedups. On that basis, Yang and Stern [322] implemented a highly scalable massively parallel FMM. They exploit the independence of front characteristics so that a newly accepted point received from a neighboring sub-domain only causes a rollback for vertices influenced by its characteristics. The authors achieved strong scalability using up to 65 536 processors. There study does not include speed functions different from $f = 1$ or complex geometries.

Another interesting approach to parallelize the FMM was suggested by Breuß et al. [36]. They focused on a domain-decomposition-free parallelization. Their idea is to use multiple wavefronts in parallel but instead of partitioning the space, they partition the set of initial vertices $\mathbf{v} \in \Gamma$. More precisely, the authors partition the initial narrow band into multiple sets (partial narrow bands) so that the union of all sets gives us the set of the initial narrow band and the intersection of all sets is empty. Then each processor executes the FMM by using one of these sets. Whenever a processor recognizes that another wavefront computed a smaller value $\phi_\Gamma(\mathbf{v})$ it stops the front at $\mathbf{v}$. Compared to Herrmann's approach, the authors report better speedups for multiple examples.

Zhao [330] introduced the first two parallel implementations of the FastSweepingMethod, one for shared memory systems and the other for distributed memory systems. Their shared memory implementation parallelizes the sweeps in different directions. The distributed memory approach uses domain decomposition to assign each domain to different processors. The authors balance the convergence of sub-domains and the number of information exchanges that causes a restart of the sub-domain sweeping. The works of Detrixhe et al. [61] build on top of Zhao's [330] work. They use Cuthill-Mckee ordering to cluster grid points on diagonal lines for sweeping. Since points on such clusters can be updated concurrently, they improve the performance of the method. The performance of this method does not reach a plateau with an increase in the number of threads. Therefore, it is suited for parallel architectures such as the GPU [272]. Recently, Shrestha and Senocak [272] developed a multi-GPU implementation of this method using CUDA and OpenACC.

Regarding parallelization, most work is done for the FastIterativeMethod. This is no surprise, because compared to the FSM and FMM, the FIM has a higher parallel potential. In its introduction, [135] a GPU implementation was already presented. Hong and Jeong [125] extended it to a multi-GPU implementation. In [9] the authors introduce a multi-level parallel approach for heterogeneous and hierarchical architectures, thereby bringing the FIM from the GPU back to multi-core shared-memory systems. They use a Cartesian grid for discretization. Genellari and Haase [89, 90] introduced a CUDA implementation of the FIM that operates on unstructured meshes in the three-dimensional space. Hong and Jeong's [124] goal was to bring the FIM to shared-memory systems with up to 32 threads. They extended the FIM in two ways: their LockFreeFastIterativeMethod version minimizes the synchronization required, especially whenever the narrow band is manipulated. Therefore, it improves scalability in a shared memory multi-core system. Their second approach GOFastIterativeMethod focuses on reducing the number of updates of the FIM, that is, the number of times SolveEikonal (Algorithm 14) is called. In a first step, they solve the eikonal equation on a coarse Cartesian grid using the standard FIM. Each grid point represents a block of the fine Cartesian grid. Based on this solution, they construct a heuristic to control the membership of the narrow band, that is, the order in which blocks are updated. In the second step, they solve the same equation on the fine grid using the constructed heuristic. They showed that for simple geometries and travel speed func-

tions f LockFreeFastIterativeMethod is preferable over GOFastIterativeMethod. Their lock-free version runs 80 times faster than a single-threaded FastMarchingMethod using 32 threads. In case of simple geometries and a complex travel speed functions f GOFastIterativeMethod outperforms LockFreeFastIterativeMethod. Note that Hong and Jeong [124] used a Cartesian grid. However, implementing GOFastIterativeMethod on an unstructured mesh is straightforward.

9.3.7 Conclusion

This review of methods shows that it is not obvious which solver performs best in practice. First, we have to assess our hardware configuration. If we are only using a single thread, the SimplifiedFastMarchingMethod is a safe choice [103]. It performs well for all domains Ω and travel speed functions f. If the geometry is simple, and f is constant, the FSM might be superior. In a multi-core hardware setup, the FastIterativeMethod and its extensions seem to be most promising. In pedestrian dynamics we have to deal with complex geometries and travel speed functions f. Therefore, the curvature of characteristic curves can be high. Consequently, I decided to stick to the narrow band based approaches, that is, the FastMarchingMethod and its simple extension the SFMM as well as the FastIterativeMethod and its lock-free extension the LockFreeFastIterativeMethod. The review also shows the importance of the order, in which vertices are updated. Furthermore, I found little work concerning discretization – it is almost always assumed as part of the input.

9.4 Mesh resolution control

To combat high computational costs, I introduce mesh resolution control techniques. Researchers in pedestrian dynamics are interested in trajectories of agents rather than the solution of the eikonal equation. Therefore, a sparse discretization is favorable to a fine one if the simulation yield similar trajectories. Since agents move to their next position evaluating the navigation field locally, for example, by computing $\nabla\phi_\Gamma$, it is essential that the gradient of the travel time $\nabla\Phi_\Gamma$ is accurately approximated by $\nabla\phi_\Gamma$. If its direction and magnitude does not change at all, a coarse mesh leads to similar results compared to a fine one, compare Fig. 9.5.

9.4.1 Curvature of the travel time

If we think of the eikonal equation's solution as a two-dimensional surface $\mathcal{S}$ in three-dimensional space, that is, $\Phi_\Gamma(\mathbf{x})$ is the z-coordinate of $\mathcal{S}$, we require a mesh that can approximate $\mathcal{S}$ accurately. Figure 9.8 illustrates such a surface. Intuitively, the mesh has to be fine if the surface is curved.

Definition 9.1 (curvature of a curve)**.** Let $g(z) = (x(z), y(z))$ be a smooth curve in the plane, then

$$\kappa_g(\mathbf{x}) = \frac{1}{r} \tag{9.36}$$

is the curvature of the curve at $\mathbf{x}$, where $\mathbf{x} = (x(z), y(z))$ for some z and r is the radius of the circle, which 'best' approximates the curve at $\mathbf{x}$.

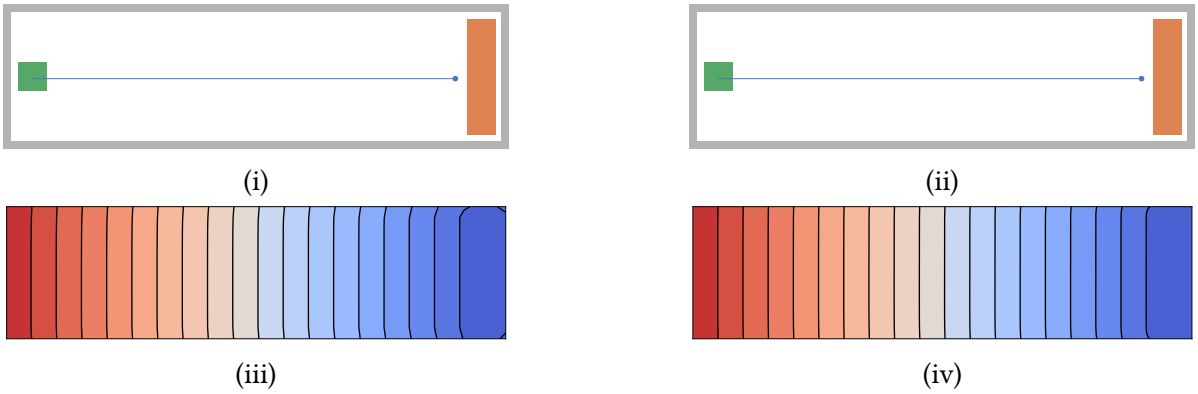


Figure 9.5: Trajectory comparison for different navigation fields: on the left, the navigation field ($n = 549$) is rather coarse, while on the right it is very fine ($n = 19898$). However, the resulting trajectory of the simulation using an optimal steps model is identical. The curvature κ_{Φ_Γ} in this example is small almost everywhere.

Definition 9.2 (normal curvature)**.** Let $\mathcal{S}$ be a smooth surface and Π be a plane that contains the unit normal vector $\mathbf{n}_\mathbf{x}$ at a point $\mathbf{x} \in \mathcal{S}$. Then $\kappa_g(\mathbf{x})$ is the *normal curvature*, where the curve g is the intersection of $\mathcal{S}$ and Π.

Definition 9.3 (mean curvature)**.** Let $\mathcal{S}$ be a smooth surface, then

$$H_\mathcal{S}(\mathbf{x}) = \frac{\kappa_{g,\max}(\mathbf{x}) + \kappa_{g,\min}(\mathbf{x})}{2} \tag{9.37}$$

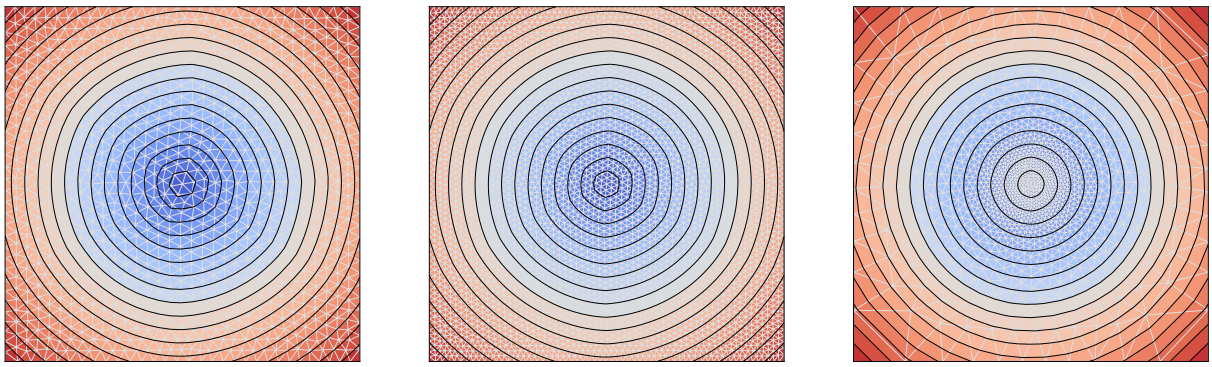
is the *mean curvature* of $\mathcal{S}$ at $\mathbf{x}$, where $\kappa_{g,\max}(\mathbf{x})$ is the maximal and $\kappa_{g,\min}(\mathbf{x})$ the minimal curvature.

One the one hand, if the *mean curvature* $H_\mathcal{S}(\mathbf{x})$ at $\mathbf{x}$ is large, the mesh resolution at $\mathbf{x}$ should be high. On the other hand, one can achieve accurate trajectories for coarse navigation fields if the curvature $H_\mathcal{S}(\mathbf{x})$ is small. The example depicted in Fig. 9.5 confirms this observation. For both navigation fields (iii) and (iv) the resulting trajectories of the simulations (i) and (ii) are identical. A good approximation of Φ_Γ gives us a good approximation of its curvature and vise versa. Since we assume the wavefront propagates through a simplex with constant speed, a high grid resolution is also required at areas where ∇f is large. Otherwise, f is poorly approximated and thus Φ_Γ.

Let us look at a point source example where $\Gamma = (0, 0)$, the domain is a square $\Omega = [-1; 1] \times [-1; 1]$ and $f = 1$ is constant. In that case, the mean curvature $H_\mathcal{S}(\mathbf{x}, \Phi_\Gamma(\mathbf{x}))$ at $(\mathbf{x}, \Phi_\Gamma(\mathbf{x}))$ is equal to the curvature of the circle of radius divided by two:

$$H_\mathcal{S}(\mathbf{x}, \Phi_\Gamma(\mathbf{x})) = \frac{1}{2 \cdot \|\mathbf{x}\|}. \tag{9.38}$$

Therefore, to increase our approximation accuracy, the mesh resolution should increase towards $(0, 0)$. Figure 9.6 illustrates how a mesh (iii) generated by EikMesh improves the solution while keeping the number of vertices small. The relative error is smaller if we use a more sophisticated



(i) (ii) (iii)

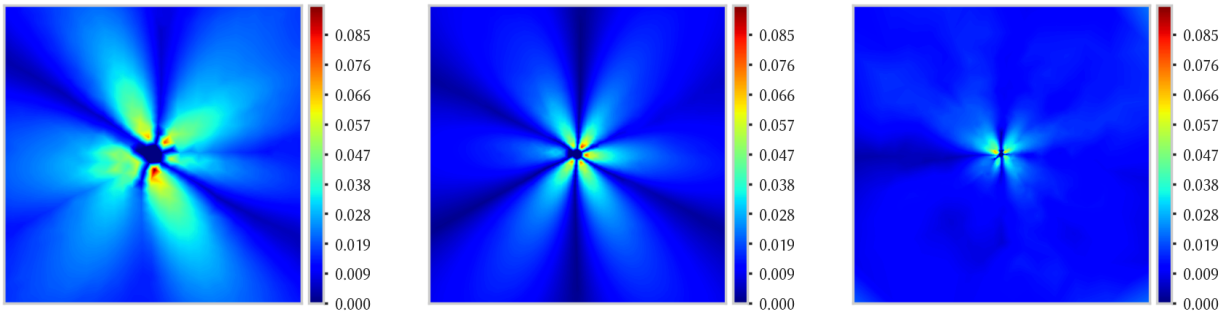
Figure 9.6: Solution of the point source example using the FMM on different unstructured meshes generated by EikMesh. I choose different element size functions h resulting in a different amount of vertices n: $h = 0.1, n = 1086$ (i), $h = 0.05, n = 4217$ (ii), $h(\mathbf{x}) = 0.025 + 0.3 \cdot \|\mathbf{x}\|, n = 1259$ (iii).

element size function while the mesh consists of $n = 1086$ vertices compared to the one with 4217 vertices, see Fig. 9.6.

For the point source example, I knew the *mean curvature* H_S beforehand, and I could incorporate information into the element size function h of EikMesh. In Section 8.6, I showed how one could construct a δ-Lipschitz element size function appropriate for any planar straight-line graph by computing the local feature size, and I left out the user-defined part of the function h_u. In fact, for the example above, I used

$$h_u(\mathbf{x}) = 0.025 + 0.6 \cdot H_S(\mathbf{x}, \Phi_\Gamma(\mathbf{x}))^{-1}. \tag{9.39}$$

However, in general, we have no knowledge of H_S. I want to introduce one solution to the problem that relies on curvature estimations. The idea is to solve the eikonal equation on a coarse mesh, estimates the curvature of its surface, and constructs a new mesh based on the estimated curvature.



(i) (ii) (iii)

Figure 9.7: Relative errors $|\phi_\Gamma(\mathbf{v}) - \Phi_\Gamma(\mathbf{v})|/\Phi_\Gamma(\mathbf{v})$ for meshes depicted in Fig. 9.6: $h = 0.1, n = 1086$ (i), $h = 0.05, n = 4217$ (ii), $h(\mathbf{x}) = 0.025 + 0.3 \cdot \|\mathbf{x}\|, n = 1259$ (iii).

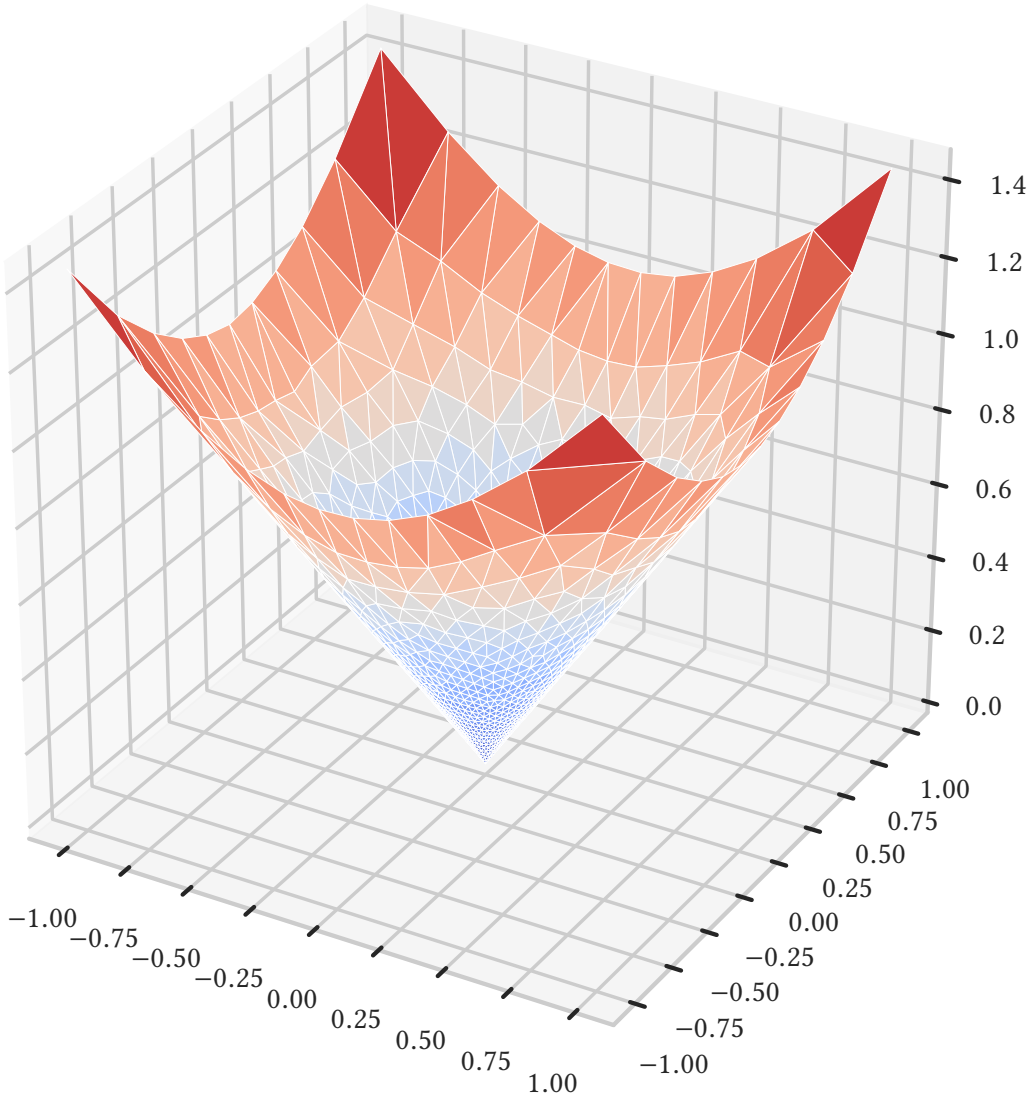


Figure 9.8: Surface of ϕ_Γ using the unstructured mesh illustrated in Fig. 9.6iii.

9.4.2 Curvature estimation

Let $\mathbf{v}_i = (x, y, \phi_\Gamma(x, y)) \in \mathbb{R}^3$ be the vertices of our lifted mesh $\mathcal{T}$ and $e_{ij} = \{\mathbf{v}_i, \mathbf{v}_j\}$, $\tau = \mathbf{v}_i\mathbf{v}_j\mathbf{v}_k$ the lifted edges and triangles respectively. Let us replace the sharp edge e_{ij} with a cylindrical bend of radius r connecting the two neighboring triangles smoothly. The maximal curvature on this bend is $1/r$ in the direction perpendicular to e_{ij}. It is zero in the direction of the edge. Consequently, the mean curvature on the cylinder is

$$\left(\frac{1}{r} + 0\right) \cdot \frac{1}{2} = \frac{1}{2r} \tag{9.40}$$

and its area is

$$\alpha_{ij} r \|\mathbf{v}_j - \mathbf{v}_j\|, \tag{9.41}$$

where α_{ij} is the angle (radian measure) between the two normals of the neighboring triangles – it is also called *dihedral angle* [18]. An estimation of the mean curvature of the cylindrical bend is defined by the integral over the entire cylindrical bend B_{ij}, that is,

$$\int_{B_{ij}} \kappa = \frac{1}{2}\alpha_{ij}\|\mathbf{v}_j - \mathbf{v}_j\|. \tag{9.42}$$

Let $\mathbf{v}$ be the vertex of a triangle and let $\mathbf{v}_1, \ldots, \mathbf{v}_m$ be the ordered neighborhood vertices of $\mathbf{v}$, then

$$\widetilde{H}_{\mathcal{T}}(\mathbf{v}) = \frac{1}{2}\sum_{i=1}^{m} \|\mathbf{v}_j - \mathbf{v}_j\| \cdot \alpha_{ij} \tag{9.43}$$

is the *mean curvature* at $\mathbf{v}$. Note that other authors, such as Alboul et al. [11], already introduced and used these formulas.

9.4.3 An iterative eikonal solver

In order to compute an accurate solution of Φ_Γ without any knowledge of Ω, Γ and f, I introduce the iterative Algorithm 20. Initially, I construct a mesh $\mathcal{T}_0$ using EikMesh with $h_u = \infty$. Then, I alternate between computing ϕ_Γ and refining the mesh based on

$$h_u(\mathbf{x}) = h_{\min} + \lambda \cdot \frac{1}{\widetilde{H}_{\mathcal{T}_i}(\mathbf{x})}, \tag{9.44}$$

where λ controls the influence of $\widetilde{H}_{\mathcal{T}_i}(\mathbf{x})$.

Multiple refinement techniques, such as techniques provided by Rivara [236], are applicable. I decided to use a modified version of the RGB-Subdivision proposed by Puppo et al. [223] because their subdivision scheme supports dynamic selective refinement and coarsening. Furthermore, it generates conforming meshes at all intermediate steps. The property to refine a mesh locally and undo the refinement dynamically can be especially useful for generating dynamic navigation fields, for which f and therefore H_S changes over time, see Section 9.4.4. Since my underlying data structure represents a conforming mesh, the second property simplifies the implementation based on my doubly-connected edge list. The presented RGB-Subdivision assigns a level $L(e)$ to

Algorithm 20: IterativEikonalSolver

Input: triangulation $\mathcal{T}$, planar straight-line graph $\mathcal{P}$, spatial domain Ω
Output: ϕ_Γ solution of Eq. (9.1)

$d_\Omega \leftarrow$ construct distance function (Section 8.7);
$h_\Omega \leftarrow$ construct geometry dependent element size function (Section 8.6);
$\mathcal{T}_0 \leftarrow$ EikMesh(d_Ω,h_Ω,$\mathcal{P}$);
$i \leftarrow 0$;
do
  $\phi_{\Gamma,i} \leftarrow$ solution of Eq. (9.1) on $\mathcal{T}_i$;
  compute $\widetilde{H}_{\mathcal{T}_i}$;
  construct h_u based on $\widetilde{H}_{\mathcal{T}_i}$;
  $\mathcal{T}_{i+1} \leftarrow$ Refine($\mathcal{T}_i$,h_u);
  $i \leftarrow i + 1$;
while $\mathcal{T}_i \neq \mathcal{T}_{i+1}$;
return $\phi_{\Gamma,i}$;

each edge of the mesh. For the initial mesh, the level is 0 for each edge. It increases by 1 whenever an edge is split. In my version, an edge e is split only if $L(e) = i - 1$, where i is the iteration index. The reason for stopping the refinement after one split is to avoid unnecessary splits, thus recompute and integrate improvements as soon as possible. Furthermore, by restricting the number of splits, I keep the number of green triangles high and mesh quality. A green triangle of the RGB-Subdivision is the result of three splits of an original green triangle. It inherits the quality of its parent (all triangles of $\mathcal{T}_0$ are green triangles), compare Fig. 9.9.

I chose two examples to demonstrate the algorithm, one with shocks and another with a complex travel speed function f. Remember that shocks are created at positions where the wavefront collapses on itself, that is, positions where Φ_Γ is not differentiable. In the context of navigation fields, at shocks agents choose different path to navigate to the same destination. The first domain is an L-shape planar straight-line graph $\mathcal{P}$ containing a hole with $\Gamma = \partial\Omega$, such that the wavefront moves with $f = 1$ speed from the domain boundary towards the medial axis. For the second, I choose $\Omega = [-1; 1] \times [-1; 1]$, $\Gamma = (0, 0)$ and

$$f(x, y) = 0.8 \cdot \sin(2\pi \cdot x) \cdot \sin(2\pi \cdot y) + 1. \tag{9.45}$$

For both examples, the parameter $\lambda = 0.3$ controls how much the curvature influences the ele-

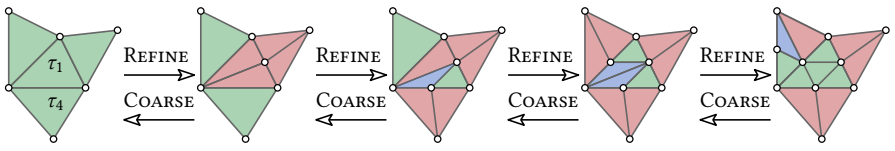
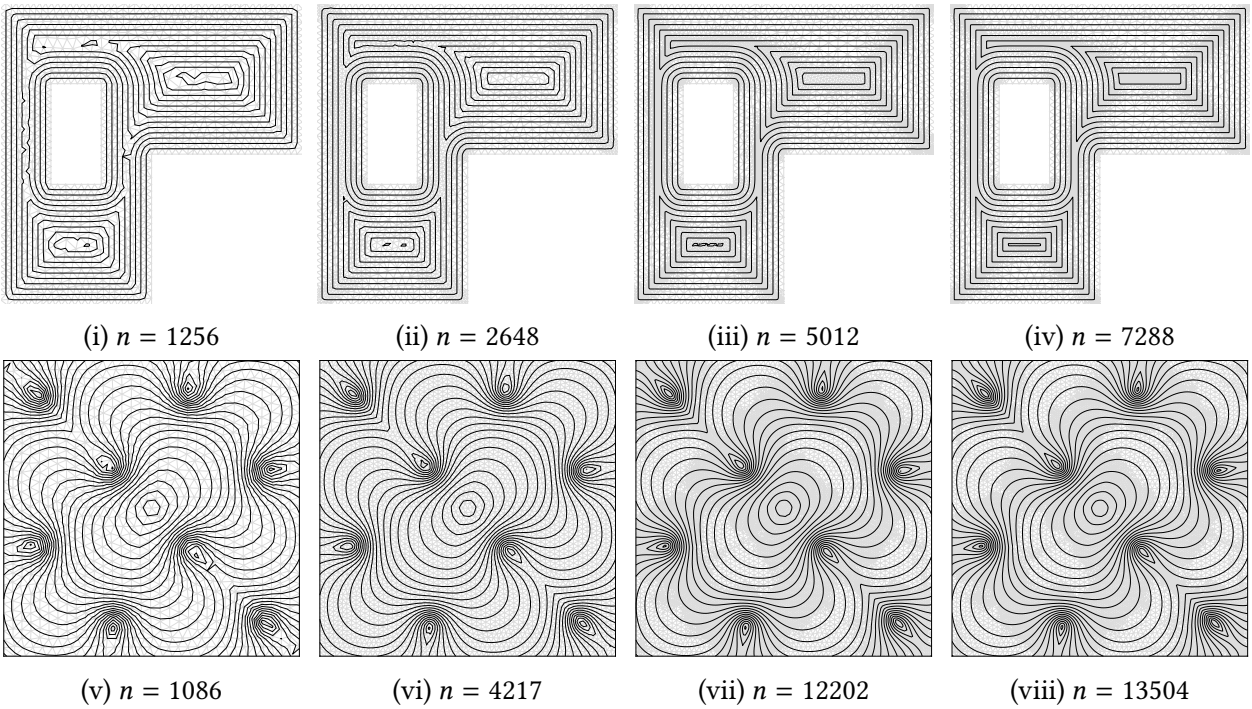


Figure 9.9: RGB-Subdivision example: splitting a green triangle leads to two red triangles, and splitting a red triangle results in a green and blue triangle. Two neighboring blue triangles at the same level become green if the edge they share is flipped. At any point, the mesh remains conforming. SplitEdge and FlipEdge are the only mesh operations required to refine the mesh.



(i) $n = 1256$ (ii) $n = 2648$ (iii) $n = 5012$ (iv) $n = 7288$

(v) $n = 1086$ (vi) $n = 4217$ (vii) $n = 12202$ (viii) $n = 13504$

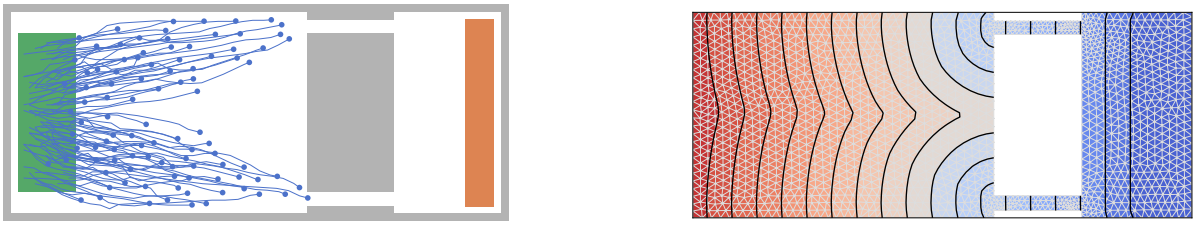
Figure 9.10: Solution of the eikonal equation computed by IterativEikonalSolver: after four iterations the algorithm terminates. Especially at shocks (i) and position where ∇f is large (v), the solution $\phi_{\Gamma,0}$ poorly approximates Φ_Γ for the initial mesh. The final results (iv and viii) $\phi_{\Gamma,3}$ are a much better approximations.

ment size. After 4 iterations IterativEikonalSolver terminates. For each iteration the accuracy of the solution increases, compare Fig. 9.10. The mesh refinement stops earlier, where the estimated curvature is small.

9.4.4 Mesh resolution control in pedestrian dynamics

Algorithm 20 can deal with very little knowledge about Ω and f. It is a general way to achieve accurate results while keeping the mesh number vertices small. However, solving the eikonal equation and refining the mesh multiple times is computationally expensive. Since fast computation is one of my goals, it is preferable to use a suitable element length function h_u, which incorporates information about f. In that way, we keep the number of mesh manipulations at a minimum. Consequently, one has to look at the concrete application more closely.

In the case of navigation fields for pedestrian dynamics, most models use a travel speed function that, in the static case, depends on the obstacle density or distance to obstacles and, in the dynamic case, on the density, speed, or flow of pedestrians. The reader might want to revisit Sections 3.4 and 3.5, where I listed different travel speed functions. I assume sparsely distributed pedestrians do not influence medium-scale navigation of pedestrians while a packed crowd does. This assumption is consistent with the travel speed function used in pedestrian simulations to this day. Since bottlenecks are the major source for congestion, ∇f might be large at jams but will be approximately zero at open spaces. Therefore, the mesh has to be fine at crowded areas



(i) (ii)

Figure 9.11: The snapshot (i) and static floor field (ii) of a two-path scenario of 14 m height: agents of the top half enter the top entry while agents below navigate through the bottom entry. The reason is that the propagating wavefront is first divided into two parts and merges the middle.

close to the domain boundary $\partial\Omega$.

Shocks are the second reason for a high mesh resolution since, at those areas, agents 'decide' which way to go. For example, in the situation depicted in Fig. 9.11, the shock at the vertical center of the domain divides agents into agents walking through the top and agents walking through the bottom bottleneck. For dynamic navigation fields shocks move and deform over the simulation time. Refining the mesh computationally effectively at moving shocks in general is a topic I am not covering in this thesis. In the discussion (Chapter 10), I give some ideas to tackle this challenge.

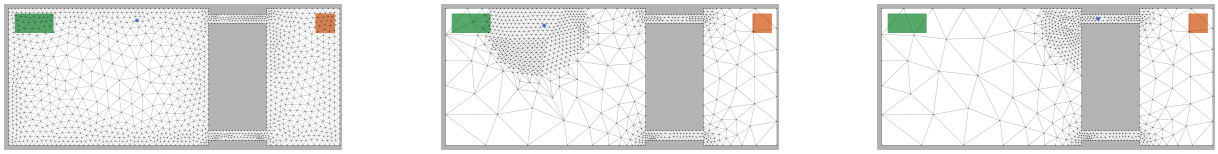
I suggest two resolution control strategies. First of all, I propose a static global strategy using EikMesh with a user-defined element size function which depends on the distance d_Ω to the domain boundary $\partial\Omega$:

$$h_u(\mathbf{x}) = \min\{h_{\min} + \delta \cdot |d_\Omega(\mathbf{x})|, h_{\max}\}. \tag{9.46}$$

I construct the mesh once before the simulation starts. Therefore, the time required for the construction of the mesh is not critical. $h_{\min}$ and δ control the mesh resolution and influence the mesh quality. Note that I use the ElementSizeConstruction introduced in Section 8.6.2 to guarantee a high mesh quality.

The second dynamic and local strategy is only useful to compute dynamic navigation fields. First, I construct a coarse mesh by applying EikMesh with $h_u(\mathbf{x}) = \infty$. During the simulation run, I dynamically coarsen and refine the mesh based on the RGB-Subdivision proposed by Puppo et al. [223]. I make sure that edges within a specific radius r_h of an agent are smaller than some threshold $h_{\max}$. To reduce the computational cost, I choose r_h larger than necessary such that the mesh manipulation is only required after multiple eikonal equations have been solved.

Manipulating the mesh using the dynamic strategy comes at a cost. The coarsening and refinement require additional computation time. We have to manipulate the mesh, but we also have to re-compute specific geometrical measures to solve the eikonal equation, such as the triangle's angles. Note that I only compute those measures once to speed up the dynamic navigation field computation in case of a fixed mesh. Furthermore, the local feature size (see Section 8.6) at bottlenecks is already relatively small such that the mesh generated by EikMesh is fine near jams regardless of h_u. Additionally, the element quality drops for blue and red triangles, thus the overall mesh quality, compare Fig. 9.9. However, for large domains, the dynamic strategy can lead to a lower number of vertices. Furthermore, if future scenarios lead to islands of packed crowds in open space, this approach might be required for accurate results.



(i) (ii) (iii)

Figure 9.12: Two different refinement strategies: the static refinement (i) computes a mesh once while the dynamic refinement (ii)-(iii) manipulates the mesh near agents every m time steps. For the static mesh I use h_u defined in Eq. (9.46) with $h_{\min} = 0.5$, $h_{\max} = 2.0$, and $\delta = 0.4$. In case of the dynamic refinement, $h_u = \infty$ and a triangle is refined if its midpoint is closer than $r_h = 7.0$ and some of its edges is longer than 0.5

9.5 The Informed Fast Iterative Method

In Section 9.3 I, introduced and discussed state-of-the-art solvers for the eikonal equation. In this section, I develop a new solver, the InformedFastIterativeMethod (IFIM), that is specialized to solve a series of eikonal equations, for which the travel speed function f changes not too much.

Dynamic navigation fields require frequent re-computation since the travel speed function f changes over time and, therefore, the travel time Φ_Γ. The time is discretized into discrete time steps $t_i = i\Delta t$ and we have to compute $\Phi_{\Gamma,0}, \ldots, \Phi_{\Gamma,k}$ consecutively using travel speed functions $f_0, \ldots, f_k$. Dynamic navigation fields impose a heavy workload, and to enable real-time simulations, the computation time of one solution of the eikonal equation has to be smaller than Δt. However, we enter new territory for dynamic navigation fields since we compute a series of similar solutions of eikonal equations. To my best knowledge, the 'similarity property' has not been considered or exploited yet.

9.5.1 Informed wavefront propagation

Since f_i depends on the dynamics of the simulation, that is, the agents' position, consecutive travel speed functions f_i, f_{i+1} and, therefore, consecutive travel times $\Phi_{\Gamma,i}, \Phi_{\Gamma,i+1}$ are "similar". The idea behind the InformedFastIterativeMethod is to incorporate information of $\phi_{\Gamma,i}$ into the computation of $\phi_{\Gamma,i+1}$, so that, the narrow band of the FastIterativeMethod stays as narrow as possible and almost all vertices converge in one update. In that case, the resulting algorithm combines the advantages of the FastFarchingMethod, that is, a minimal amount of updates, and the FastIterativeMethod, that is, highly parallel propagation of the narrow band.

The similarity I am most interested in is the similarity of the wave propagation order, that is, the order, in which the wave arrives at the vertices of the mesh. Let me define this similarity more formally.

Definition 9.4 (defining simplex)**.** Let $\phi_{\Gamma,i+1}$ be the approximation of the solution of the $(i+1)^{\text{th}}$ eikonal equation and let $\mathbf{v} \in \mathcal{T}$ be some vertex of the mesh. Then

$$\tau = \mathbf{v}\mathbf{u}_1\mathbf{u}_2$$

is the *defining simplex* of $\mathbf{v}$ if and only if $\mathbf{v}$ received its value $\phi_{\Gamma,i+1}(\mathbf{v})$ from $\mathbf{u}_1$ and $\mathbf{u}_2$. In other words, the wavefront arrived at $\mathbf{v}$ coming from within τ.

Definition 9.5 (defining vertex)**.** Let $\mathcal{V}$ be the set of vertices of the underlying mesh $\mathcal{T}$, then

$$\pi_i : \mathcal{V} \to \mathcal{V} \times \mathcal{V} \tag{9.47}$$

is the *defining vertex relation* of the i^{th} eikonal equation, such that,

$$\pi_i(\mathbf{v}) = \{\mathbf{u}_1, \mathbf{u}_2\} \tag{9.48}$$

if and only if $\tau = \mathbf{v}\mathbf{u}_1\mathbf{u}_2$ is the *defining simplex* of $\mathbf{v}$.

Definition 9.6 (defining vertex set)**.** Let $\pi_i : \mathcal{V} \to \mathcal{V} \times \mathcal{V}$ be the *defining vertex relation* and $\mathbf{v}$ be a vertex in $\mathcal{V}$. Then $\mathcal{V}_{\mathbf{v},i}$ is the *defining vertex set* of $\mathbf{v}$ (of the i^{th} eikonal equation) defined recursively by

$$\mathbf{v} \in \mathcal{V}_{\mathbf{v},i} \ \wedge \ \mathbf{u} \in \mathcal{V}_{\mathbf{v},i} \Rightarrow \pi_i(\mathbf{u}) \subseteq \mathcal{V}_{\mathbf{v},i}. \tag{9.49}$$

Lemma 9.3 (defining acyclic graph)**.** *Let π_i be the defining vertex relation, then $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$ with*

$$(\mathbf{v}, \mathbf{u}) \in \mathcal{E}_i \iff \mathbf{u} \in \pi_i(\mathbf{v}) \tag{9.50}$$

is the defining directed acyclic graph DAG (of the i^{th} eikonal equation).

The *defining simplex* is the triangle or edge from within the wave arrives first at a vertex $\mathbf{v}$. If it is an edge, there is only one *defining vertex* such that $|\pi_i(\mathbf{v})| = 1$ follows. Also note that $\pi_{i+1}(\mathbf{v}) = \emptyset$ if and only if $\mathbf{v} \in \Gamma$. The defining graph $\mathcal{G}_i$ is acyclic by the nature of the wave propagation – information flows outwards, and therefore, never back and forth. Looking at two different wave propagations on the same mesh, I call the wave propagation order identical if the resulting dependency graphs are identical. The following *similarity metric* counts the number of edges different in two defining graphs and gives us a measure of how similar the wave propagation order is:

Definition 9.7 (similarity metric)**.** Let $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$, $\mathcal{G}_{i+1} = (\mathcal{V}, E_{i+1})$ be two defining graphs, then

$$D(\mathcal{G}_i, \mathcal{G}_{i+1}) = \frac{|\mathcal{E}_i \setminus \mathcal{E}_{i+1}| + |\mathcal{E}_{i+1} \setminus \mathcal{E}_i|}{|\mathcal{E}_{i+1}| + |\mathcal{E}_{i+1}|} \tag{9.51}$$

gives us the similarity of two defining graphs.

For a moment, let us assume we know the solution $\phi_{\Gamma,i+1}$ before we solve the equation. For each vertex $\mathbf{v}$ of the underlying mesh $\mathcal{T}$, we can identify its *defining simplex*. Furthermore, we can construct the *defining vertex relation* π_{i+1} and the *defining graph* $\mathcal{G}_{i+1}$. We know that necessary information flows from a vertex $\mathbf{v}$ to $\mathbf{u}$ if and only if there exists a path from $\mathbf{v}$ to $\mathbf{u}$ on $\mathcal{G}_{i+1}$. Therefore, we can compute $\phi_{\Gamma,i+1}(\mathbf{v})$ by solving the equation only for the *defining vertex set* $\mathcal{V}_{\mathbf{v},i+1} \subseteq \mathcal{V}$. More importantly, we can identify unnecessary computations, that is, any computation inconsistent with the wavefront propagation order given by $\mathcal{G}_{i+1}$. Because we know that we get the correct value $\phi_{\Gamma,i+1}(\mathbf{v})$ if and only if we already computed the values for vertices in $\mathcal{V}_{\mathbf{v},i+1} \setminus \{\mathbf{v}\}$, we can postpone the computation $\phi_{\Gamma,i+1}(\mathbf{v})$ until those vertices have their final value.

Algorithmically, the InformedFastIterativeMethod (IFIM) keeps its narrow band $\mathcal{L}$ as narrow as possible, by only adding vertices to $\mathcal{L}$, if their *defining vertices* are already *burned*, compare Line 16 of Algorithm 21. Remember, for the FIM, the same vertex can be added to, and

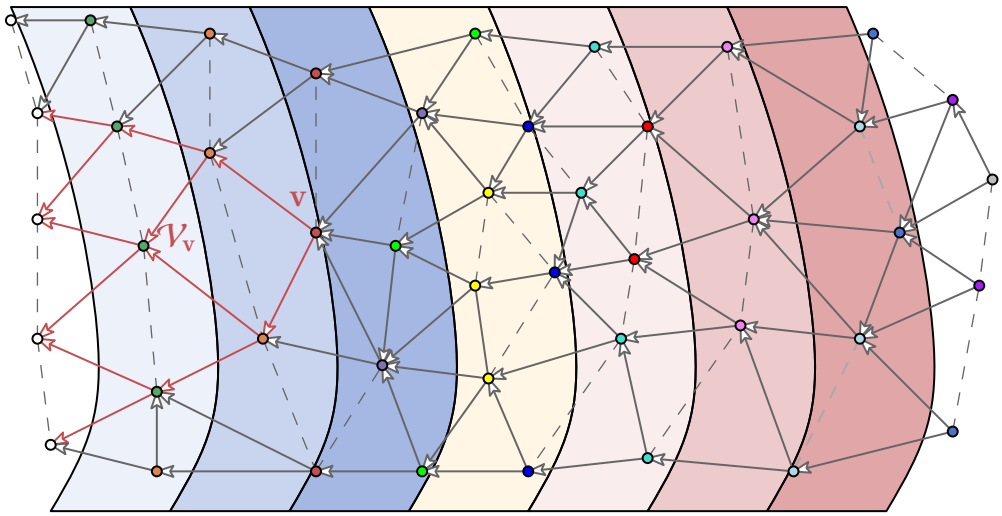


Figure 9.13: Example of a defining graph: the wavefront propagates from left to right. Each vertex color encodes a level starting from left (level 0) to the right (level 13). The arrows indicate the edges of the defining graph. Red arrows belong to the subgraph that connects all vertices in $\mathcal{V}_\mathbf{v}$.

removed from the narrow band multiple times. This happens whenever the propagating wavefront changes its direction and might cause a chain reaction – vertices of $\mathcal{L}$ might be updated which causes neighboring vertices to be added to the narrow band. Interestingly, this can only happen if a vertex $\mathbf{v}$ is added to the narrow band $\mathcal{L}$, before $\pi_{i+1}(\mathbf{v})$ received their final values. In that case $\phi_{\Gamma,i+1}(\mathbf{v})$ is too large and will be decreased by its *defining vertices* $\pi_{i+1}(\mathbf{v})$ later on.

Now let us drop our assumption that we know the solution $\phi_{\Gamma,i+1}$ before we actually solve the equation. Since this is not the case and we neither know $\phi_{\Gamma,i+1}$ nor π_{i+1}, I approximate π_{i+1} by π_i. My assumption is that most edges present in $\mathcal{G}_i$ are also present in $\mathcal{G}_{i+1}$, that is, $D(\mathcal{G}_i, \mathcal{G}_{i+1}) \approx 0$. This is the case if gradients of two consecutive solutions point in a similar direction. It is important to note that identical graphs do not imply identical gradient directions. Therefore, even if gradients change, the graph might be unaffected.

One might look at the dynamic travel speed function to decide if this assumption is valid. Instead, my argument looks at the pedestrian behavior researchers in the field want to model: a change in the direction of the gradient at $\mathbf{x}$ means that agents close to $\mathbf{x}$ move in a different direction than the agents previously located around $\mathbf{x}$. If the agents' movement direction changed strongly constantly at a certain area, it would mean that agents, moving towards Γ, constantly change their medium-scale navigation. Therefore, they would constantly use very different paths towards their destination which would result in chaotic behavior which is a very questionable situation.

Constructing π_{i+1} while solving for $\phi_{\Gamma,i+1}$ does not introduce any substantial additional computational costs. Each time $\phi_{\Gamma,i+1}(\mathbf{v})$ is decreased by an update coming from $\mathbf{u}_1, \mathbf{u}_2$, I update π_{i+1} accordingly so that

$$\pi_{i+1}(\mathbf{v}) = \{\mathbf{u}_1, \mathbf{u}_2\}\,. \tag{9.52}$$

Testing if both *defining vertices* $\mathbf{u}_1, \mathbf{u}_2$ are already *burned* is implemented by comparing a flag.

Both operations require $O(1)$ time and additional overall $O(n)$ memory.

Jeong et al. [135] proved that the FIM is consistent with the corresponding Eq. (9.1). Their proof is based on the assumption that each vertex is added to the narrow band at least once. Lemma 9.4 provides the same property for the InformedFastIterativeMethod.

Lemma 9.4. *After the IFIM terminates, each vertex has been added to the narrow band at least once.*

Proof. Let's suppose that at some point during the computation, a vertex $\mathbf{v}$ has not been added to the narrow band because its defining vertices has not been *burned* out yet. Therefore, there is some neighbor $\mathbf{u}_1$ of $\mathbf{v}$ that has not been *burned* yet. Either it is *burning*, that is, it is part of the narrow band, or it is *unreached*. If the neighbor $\mathbf{u}_1$ is part of the narrow band, it will *burn* out eventually. The same happens for the second defining vertex $\mathbf{u}_2$ thus $\mathbf{v}$ will be considered and added to the narrow band since both of its defining vertices *burned*. If $\mathbf{u}_1$ or $\mathbf{u}_2$ is *unreached*, they have not yet been added to the narrow band. By induction over $\mathcal{G}_i$, we eventually end up at some vertex $\mathbf{v}'$, for which one of the defining vertices is *burned* and one is *burning*. The *burning* one is part of the narrow band, and when it changes from *burning* to *burned*, $\mathbf{v}'$ will be added to the narrow band eventually. Using this induction argument, it is clear that $\mathbf{v}$ will be added as well. ∎

If two consecutive defining graphs differ, the IFIM might add a vertex $\mathbf{v}$ too late to its narrow band $\mathcal{L}$. In that case, $\phi_{\Gamma,i+1}(\mathbf{v})$ already has its final value, but the wave propagation is stalled until its defining vertices of the previous iteration *burn* out. The difference between adding vertices too early and too late is that in the former case, wrong information propagates, while in the latter case, correct information is held back. Since the wavefront propagates further without integrating $\phi_{\Gamma,i+1}(\mathbf{v})$, it will lead to wrong values for all $\mathcal{V}_{\mathbf{v},i+1}$ visited by the wavefront before $\mathbf{v}$ will eventually be added to the narrow band. In that sense, some vertices in $\mathcal{V}_{\mathbf{v},i+1}$ are added too early to the narrow band. Therefore, adding vertices too late to the narrow band is similar to adding vertices too early. Consequently, it is not guaranteed that IFIM outperforms the FIM in any case.

The FIM can be seen as a special case of the IFIM where each defining graph is the same, and each graph imposes the breadth-first wavefront propagation order of the FIM.

9.5.2 Partial wave propagation

So far, I only described how to incorporate the wavefront propagation order of the previous computation into the next one. However, we can go one step further and incorporate travel times as well. Let's look at the simple scenario depicted in Fig. 9.5, and let's imagine a group of agents move from the left to the right destination. Since f does not change for all positions distant from agents, consecutive wavefront propagations are identical to the point, where they arrive at the crowd. The idea behind partial wave propagation is to exploit partially equal wavefront propagations to reduce the number of required updates even further. More precisely, if $\mathcal{V}_{\mathbf{v},i} = \mathcal{V}_{\mathbf{v},i+1}$ and $f_i(\mathbf{v}) = f_{i+1}(\mathbf{v})$ for all vertices in $\mathcal{V}_{\mathbf{v},i}$ we know that $\phi_{\Gamma,i}(\mathbf{v}) = \phi_{\Gamma,i+1}(\mathbf{v})$ and we can save computational costs.

Lemma 9.5. *Let $\mathbf{v} \in \mathcal{T}$ be a vertex of the underlying mesh and $\phi_{\Gamma,i}$ be the approximated solution of the eikonal equation of the previous time step. Let $\phi_{\Gamma,i+1}$ be the values computed before some update*

Algorithm 21: InformedFastIterativeMethod

Input: triangulation $\mathcal{T}$, spatial destination Γ, spatial domain Ω, defining vertex relation π_i of iteration i, solution of the eikonal equation of iteration i $\phi_{\Gamma,i}$

Output: $\phi_{\Gamma,i+1}$ solution of Eq. (9.1), defining vertex relation π_{i+1} of iteration $i+1$

$\phi_{\Gamma,i+1}(\mathbf{v}) \leftarrow \phi_0(\mathbf{v})$ for all $\mathbf{v} \in \Gamma$;
$\phi_{\Gamma,i+1}(\mathbf{v}) \leftarrow \infty$ for all $\mathbf{v} \notin \Gamma$;
$\mathcal{L} \leftarrow \{(\mathbf{v}, \phi_{\Gamma,i+1}(\mathbf{v})) \mid \mathbf{v} \in \Gamma\}$ `// set of burning vertices`
$\mathcal{B} \leftarrow \mathcal{V}_0$ `// set of burned vertices`
while $\mathcal{L} \neq \emptyset$ **do**
    foreach $(\mathbf{v}, \phi_{\Gamma,i+1}(\mathbf{v})) \in \mathcal{L}$ **in parallel do**
        $p \leftarrow \phi_{\Gamma,i+1}(\mathbf{v})$;
        if $\mathbf{v}$ requires an update (Lemma 9.5) **then**
            $(q, \mathbf{u}_1, \mathbf{u}_2) \leftarrow$ SolveEikonal($\mathbf{v}$);
            $\pi_{i+1}(\mathbf{v}) \leftarrow \{\mathbf{u}_1, \mathbf{u}_2\}$;
        else
            $q \leftarrow \phi_{\Gamma,i}(\mathbf{v})$;
            $\pi_{i+1}(\mathbf{v}) \leftarrow \pi_i(\mathbf{v})$;
        $\phi_{\Gamma,i+1}(\mathbf{v}) \leftarrow q$;
        if $|p - q| < \epsilon$ **then**
            $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathbf{v}\}$;
            foreach neighbor $\mathbf{u}$ of $\mathbf{v}$ **do**
                if $(\mathbf{u}, \phi_{\Gamma,i+1}(\mathbf{u})) \notin \mathcal{L} \wedge \pi_i(\mathbf{u}) \subset \mathcal{B}$ **then**
                    $p \leftarrow \phi_{\Gamma,i+1}(\mathbf{u})$;
                    if $\mathbf{u}$ requires an update (Lemma 9.5) **then**
                        $(q, \mathbf{v}_1, \mathbf{v}_2) \leftarrow$ SolveEikonal($\mathbf{u}$);
                        $\pi_{i+1}(\mathbf{v}) \leftarrow \{\mathbf{v}_1, \mathbf{v}_2\}$;
                    else
                        $q \leftarrow \phi_{\Gamma,i}(\mathbf{u})$;
                        $\pi_{i+1}(\mathbf{u}) \leftarrow \pi_i(\mathbf{u})$;
                    if $p > q$ **then**
                        $\phi_{\Gamma,i+1}(\mathbf{u}) \leftarrow q$;
                        $\mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathbf{u}, \phi_{\Gamma,i+1}(\mathbf{u}))\}$;
                        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\mathbf{u}\}$;
            $\mathcal{L} \leftarrow \mathcal{L} \setminus \{(\mathbf{v}, \phi_{\Gamma,i+1}(\mathbf{v}))\}$;
return $\phi_{\Gamma,i+1}, \pi_{i+1}$;

of $\mathbf{v}$*. Then this update will yield* $\phi_{\Gamma,i}(\mathbf{v})$ *if for all neighbors* $\mathbf{u}$ *of* $\mathbf{v}$

$$[\phi_{\Gamma,i}(\mathbf{u}) < \phi_{\Gamma,i}(\mathbf{v}) \Rightarrow \phi_{\Gamma,i}(\mathbf{u}) = \phi_{\Gamma,i+1}(\mathbf{u})] \ \wedge \tag{9.53}$$

$$[\phi_{\Gamma,i}(\mathbf{u}) \geq \phi_{\Gamma,i}(\mathbf{v}) \Rightarrow (\phi_{\Gamma,i}(\mathbf{u}) \leq \phi_{\Gamma,i+1}(\mathbf{u}) \ \vee \ \phi_{\Gamma,i+1}(\mathbf{u}) \geq \phi_{\Gamma,i+1}(\mathbf{v}))] \tag{9.54}$$

holds.

Proof. Equation (9.53) ensures that values of the defining vertices $\pi_i(\mathbf{v})$ are identical for i and $i+1$. However, $\pi_i(\mathbf{v}) \neq \pi_{i+1}(\mathbf{v})$ is possible. This can happen if the travel time of some neighbor $\mathbf{u} \notin \pi_i(\mathbf{v})$ decreases, that is, if

$$\phi_{\Gamma,i+1}(\mathbf{u}) < \phi_{\Gamma,i}(\mathbf{u}). \tag{9.55}$$

Equation (9.53) ensures that this is not the case for neighbors, for which the travel time was already smaller than the travel time at $\mathbf{v}$. Equation (9.54) ensures that for all other neighbors the value increases or it decreases not too much. Even if the value decreases, if $\phi_{\Gamma,i+1}(\mathbf{u}) \geq \phi_{\Gamma,i+1}(\mathbf{v})$ holds, the wavefront arrived at $\mathbf{v}$ earlier than at $\mathbf{u}$ and therefore $\mathbf{u} \notin \pi_{i+1}(\mathbf{v})$. ■

Before updating the travel time at a vertex $\mathbf{v}$ by calling SolveEikonal($\mathbf{v}$), the IFIM tests if the update is necessary by applying Lemma 9.5, compare Lines 8 and 20 of Algorithm 21. And if it is not necessary, the IFIM copies the old value, compare Lines 12 and 24. The highlighted parts of Algorithm 21 indicate the difference between FIM and IFIM.

If the *defining vertex set* of some vertex changes in any way, it is almost certain that we can not use the old travel time value for this vertex. The reason is that any change will propagate through the set and will only vanish due to numeric inaccuracy or if multiple changes cancel each other out which is unlikely to happen. Therefore, if many values at vertices close to Γ change, it is likely that we have to recompute everything.

Another strategy which saves computation time if we apply the FastMarchingMethod is to *burn* all vertices which are unaffected before starting the wave propagation. Let

$$\mathcal{V}_{f,i+1} = \{\mathbf{v} \in \mathcal{V} : f_i(\mathbf{v}) \neq f_{i+1}(\mathbf{v})\} \tag{9.56}$$

then we know that for each vertex $\mathbf{u}$,

$$\phi_{\Gamma,i}(\mathbf{u}) \leq \min_{\mathbf{v} \in \mathcal{V}_{f,i+1}} \phi_{\Gamma,i}(\mathbf{v}) \Rightarrow \phi_{\Gamma,i+1}(\mathbf{u}) = \phi_{\Gamma,i}(\mathbf{u}) \tag{9.57}$$

holds. Therefore, we could copy all those values, initialize the narrow band by adding all neighbors of *burned* vertices, and start the wave propagation from that point.

As mentioned in the review above, the LockFreeFastIterativeMethod requires less synchronization compared to FastIterativeMethod and is especially suitable in a shared-memory hardware setup. My proposed extensions also transform the LockFreeFastIterativeMethod and other extensions of the FastIterativeMethod (FIM) into 'informed' versions. For the LockFreeFastIterativeMethod the exact same code changes, depicted in Algorithm 21, were required.

9.5.3 Natural wavefront propagation

In this subsection, I discuss an alternative approach for cases where consecutive defining graphs are identical. Note that, other than the previous algorithm, I neither implemented nor tested the following one. I want to look back to the water wave analogy, which was the starting point of this chapter. In the real physical world, the wave moves perfectly with time through the water. Let's imagine nature is a computing machine with endless resources. Then, nature computes each level set of travel time in parallel, and each computation requires the same amount of time. In that sense, nature achieves perfect parallelism. At any point during the computation, there is no vertex, for which the machine could additionally start the computation since information is still missing.

Assuming we know the defining vertex relation π_i, we can emulate the natural wavefront propagation to achieve perfect parallelism for the discrete case. Continuous level sets translate to *defining levels* $\mathcal{V}_0, \dots, \mathcal{V}_k$ of the *defining acyclic graph* $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$. The levels are a partition of $\mathcal{V}$.

Definition 9.8 (defining level)**.** Let $\pi_i : \mathcal{V} \to \mathcal{V} \times \mathcal{V}$ be the *defining vertex relation* of the i^{th} eikonal solution. Then $\mathbf{v} \in \mathcal{V}$ is an element of the *defining level* $\mathcal{V}_0$ if and only if it is an element of Γ. For $l > 0$ a vertex $\mathbf{v} \in \mathcal{V}$ is an element of the *defining level* $\mathcal{V}_l$ if and only if

$$\pi_i(\mathbf{v}) \cap \mathcal{V}_{l-1} \neq \emptyset \ \wedge \ \pi_i(\mathbf{v}) \subseteq \bigcup_{j=0}^{l-1} \mathcal{V}_j \tag{9.58}$$

holds.

We can compute the travel time of each vertex of a *defining level* $\mathcal{V}_l$ in parallel as soon as the travel times of all vertices in $\mathcal{V}_{l-1}$ have been computed. Let $\chi : \mathcal{V} \to \mathbb{N}_0$ be the level a vertex $\mathbf{v}$. We can construct the level partition by using π_i, since the level $\chi(\mathbf{v})$ is inductively defined by

$$\begin{aligned} \chi(\mathbf{v}) &= 0 \iff \mathbf{v} \in \Gamma, \\ \chi(\mathbf{v}) &= \max_{\mathbf{u} \in \pi_i(\mathbf{v})} \chi(\mathbf{u}) + 1. \end{aligned} \tag{9.59}$$

Algorithm 22: NaturalMarchingMethod

Input: defining levels $\mathcal{V}_0, \dots, \mathcal{V}_k$, triangulation $\mathcal{T}$
Output: ϕ_Γ solution of Eq. (9.1)

```
φ_Γ(v) ← φ_0(v) for all v ∈ V_0;
φ_Γ(v) ← ∞ for all v ∉ V_0;
foreach level l = 1, ..., k do
    foreach v ∈ V_l in parallel do
        φ_Γ(v) ← SolveEikonal(v)
return φ_Γ;
```

The resulting NaturalMarchingMethod, illustrated in Algorithm 22, might be a powerful alternative if many consecutive defining graphs are identical. It especially suits a single instruction multiple data (SIMD) hardware setup since the algorithm can be implemented such that the execution path divergence is small. The parallel time complexity of the NaturalMarchingMethod is $O(k)$.

9.6 Experimental comparison

In this section, I analyze the performance of the InformedFastIterativeMethod (IFIM) and show how dynamic navigation fields can be effectively used for medium-scale navigation. To do so, I compare the IFIM to the FastIterativeMethod (FIM) and the FastMarchingMethod (FMM). Instead of measuring the run time, which highly depends on an optimal implementation and the hardware setup, I measure the number of times a vertex is updated and removed from the narrow band. Therefore, I compare workloads instead of computation times.

Let u be the overall number of times a vertex (that received a new value) has been removed from the narrow band. The FastMarchingMethod (FMM) requires a minimal amount of updates since each vertex is removed from the narrow band exactly once. Consequently, it requires

$$u_{\text{FMM}} = |\mathcal{V} \setminus \mathcal{V}_0| \tag{9.60}$$

updates. In general, one can expect a small number of updates if the imposed update order is consistent with the natural wavefront propagation. In the case of the FIM, this is the case if the natural wave propagation order is similar to a breath first-order. If the IFIM is able to consistently 'learn' and approximates the natural wavefront propagation order, it will require fewer updates than the FIM.

9.6.1 Examples

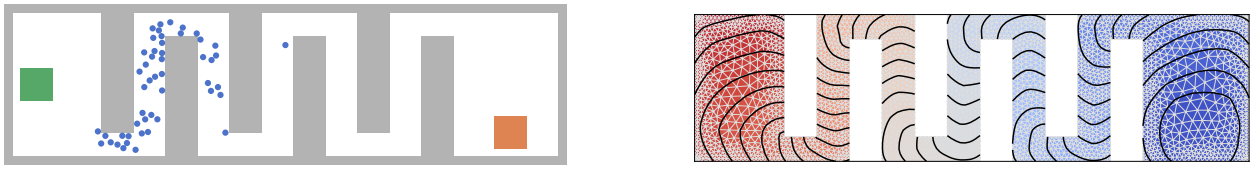
For each of the following scenarios, I use the travel speed function

$$f(\mathbf{x}) = \begin{cases} 1/(1 + h_{\mathcal{W}} \cdot \rho_{\mathcal{W}}(\mathbf{x}) + h_{\mathcal{A}} \cdot \rho_{\mathcal{A}}(\mathbf{x})) & \text{if } \mathbf{x} \in \Omega \\ 0 & \text{else,} \end{cases} \tag{9.61}$$

which we introduced in [165]. This travel speed function depends on the obstacle and pedestrian density. $h_{\mathcal{A}}$, $h_{\mathcal{W}}$ control the influence of the pedestrian $\rho_{\mathcal{A}}$ and obstacle $\rho_{\mathcal{W}}$ density. I discussed the effect in Section 3.5. It leads to the avoidance of crowded areas at the cost of longer travel distances. Therefore, agents choose faster paths over shorter ones. For all simulations, I use a time step size equal to $\Delta t = 0.4$ seconds. Each dynamic navigation field is computed on a static mesh making use of the static *refinement strategy* discussed in Section 9.4. If not stated otherwise, everything is executed sequentially.

Corridor

The first 35×10 square meter scenario is a shock free corridor. 50 agents walk from the left to the right. They have to change directions multiple times. A snapshot and the navigation field as it resulted 15 s into the simulation is depicted in Fig. 9.14.



(i) (ii)

Figure 9.14: Snapshot (i) and dynamic floor field (ii) 15 s into the simulation: blue agents walk from left (the green source) to right (the orange destination Γ). Here I use an optimal steps model a travel speed function that depends on the pedestrian density.

Although the wavefront changes direction multiple times, the FIM performs well since the wavefront propagation order is similar to a breadth-first search on the mesh $\mathcal{T}$ because the mesh aligns with the change in direction. If we look at the number of updates in Fig. 9.15i, it is evident that the FIM performs almost optimally, since it requires approximately $u_{\text{FMM}} = 2{,}975$ updates for each time step.

However, the same is true for the IFIM. In addition, the IFIM requires fewer updates until the agents come close to their destination. We can clearly see how the partial wave propagation prevents almost all updates at the start of the simulation and after the first eikonal equation has been solved. Figure 9.16i shows the narrow band size for each narrow band manipulation for the 75^{th} time step. The narrow band size for the IFIM stays smaller at all times, but the difference is not too large.

Bottlenecks

The second 35×15 square meter scenario consists of two bottlenecks. It was already introduced in Section 9.4.4, compare Fig. 9.11. During the execution of solvers, the wavefront is divided into two parts. These parts merge and consequently cause a shock. Therefore, the wavefront propagation order diverges from the breath-first search.

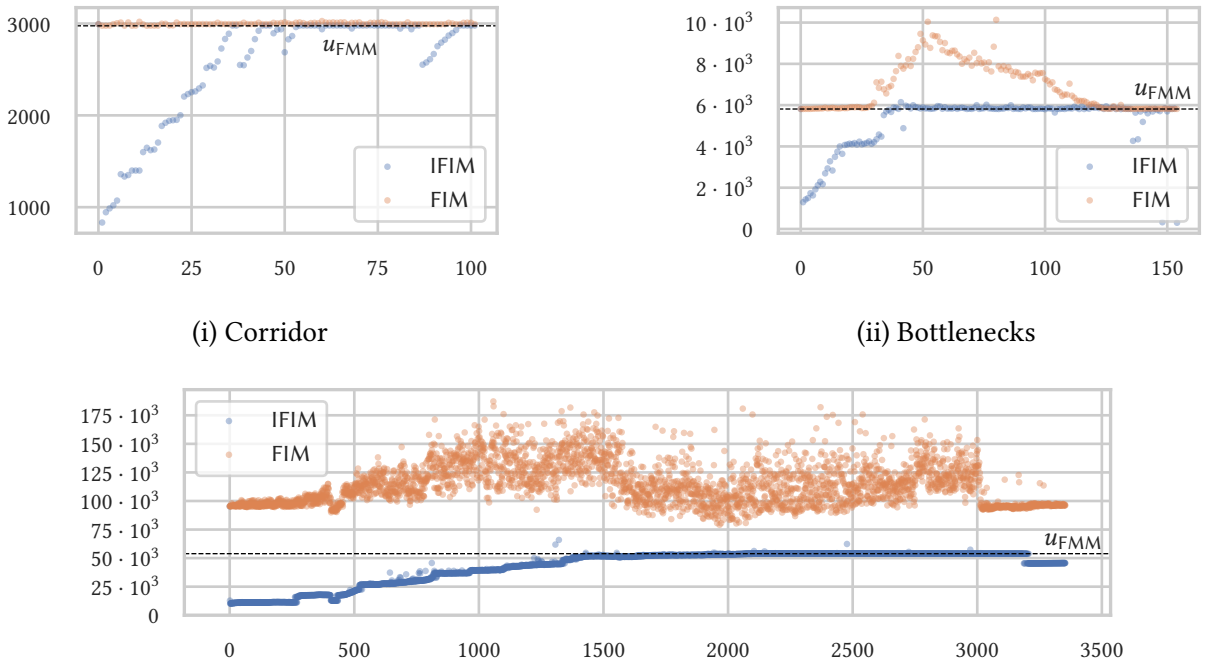
We can observe the effect in the number of required updates illustrated in Fig. 9.15ii. At the beginning of the simulation, the FIM performs almost optimally. But as soon as the shock moves and deforms, its performance suffers up to a point where the number of required updates is almost doubled compared to $u_{\text{FMM}} = 5{,}812$.

In comparison, the IFIM requires more or less the optimal amount of updates for all time steps except the first one. While for the FIM, the number of updates increases after the 40^{th} time step, the IFIM 'learns' and adapts its update order accordingly.

The narrow band size of both methods for the 75^{th} time step is depicted in Fig. 9.16ii. It shows that the narrow band size of the FIM exceeds the narrow band size of the IFIM. At one point, $\mathcal{L}$ is three times as large. Consequently, the IFIM requires fewer manipulations of its narrow band data structure.

Richard-Wagner-Straße

The last scenario is a large-scale example. It is the Richard-Wagner-Straße in Kaiserslautern flooded with 10,000 agents walking from top to the bottom, compare Fig. 9.17. The domain is



(i) Corridor

(ii) Bottlenecks

(iii) Richard-Wagner-Straße

Figure 9.15: The number of times a vertex gets removed from the narrow band for each simulation time step t_j, $t_j \cdot \Delta t$ seconds into the simulation: for the corridor scenario (i), the bottleneck scenario (ii), and the Richard-Wagner-Straße (iii) the IFIM requires less removals than the FIM, that is, approximately the same amount as the FMM.

228×560 square meters large. This scenario is inspired by protest marches that often take place in this part of Kaiserslautern. Every second, 20 agents spawn at random positions inside the green rectangle. Each agent walks towards the orange rectangle Γ at the bottom. Note that I already introduced its geometry in Chapter 6 (see Fig. 6.3) and displayed a mesh generated by EikMesh (see Fig. 8.19) in Chapter 8. Since there are multiple combinations of streets leading to Γ, each solution $\phi_{\Gamma,0}, \ldots, \phi_{\Gamma,k}$ contains multiple shocks.

For each time step, the FIM requires between 75,000 and 193,000 updates. This is almost four times the optimum $u_{\text{FMM}} = 53{,}888$, compare Fig. 9.15iii. In contrast, the IFIM never requires more than approximately u_{FMM} number of updates.

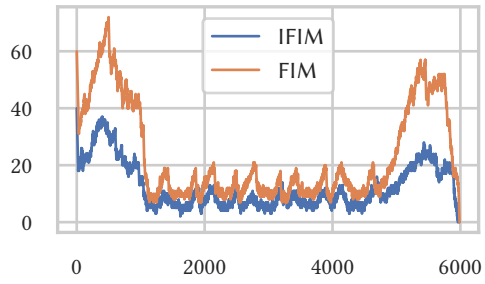
The narrow band sizes at time step 2,000 match this trend. The maximum narrow band size of the FIM is almost three times of the maximum narrow band size of the IFIM, compare Fig. 9.16iii.

On my default hardware setup using only one thread, the IFIM requires on average 139 milliseconds to solve an eikonal equation. The average run time drops to 85 milliseconds using two threads. With 150 milliseconds on average, the FMM performs almost equally well. The reason is that the narrow band of the method remains relatively small and the heap operations are therefore computationally inexpensive. However, the execution time of the IFIM can be decreased further by using additional processors.

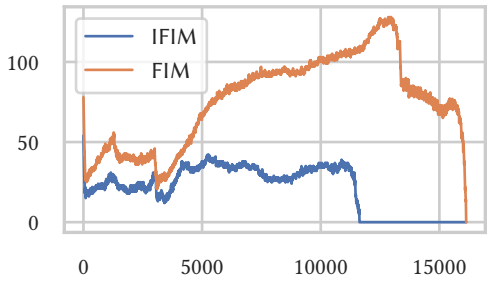
9.6.2 Conclusion

In summary, the IFIM outperforms the FIM for each example. The difference increases with the number of shocks, that is, with the number of distinct detours to the destination. But even if this number is rather low, the IFIM performs better, and since it does not introduce any significant computational work, the IFIM is preferable to the FIM for dynamic navigation field computation. Furthermore, the narrow band sizes indicate that there is enough parallel potential. Since the number of updates of the IFIM seems to be always close to the optimum u_{FMM}, there is also no reason to use the FMM over the IFIM. In the case of static navigation fields, the FMM or some of its extensions might be superior. One might want to apply the FMM to compute $\phi_{\Gamma,0}$ to learn the first defining graph and then apply the IFIM for all consecutive computations. Because the number of updates required by the IFIM is close to u_{FMM} and its parallel potential is similar to the FIM, the IFIM combines the advantages of the FIM and FMM.

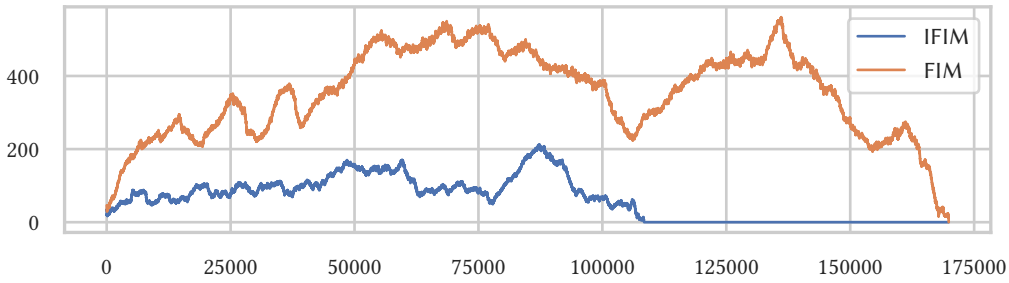
The effect of using a dynamic navigation field over a static one can be observed in Fig. 9.18. At first, agents choose the shortest path along the main street. After a while, the main street becomes more and more crowded, and at some point in time, agents start to use a slightly longer detour. Eventually, the detour becomes crowded as well, and agents start using even longer paths to Γ. More and more side streets are flooded with agents, compare Figs. 9.18i to 9.18iv. When



(i) Corridor (t_{50})



(ii) Bottlenecks (t_{75})

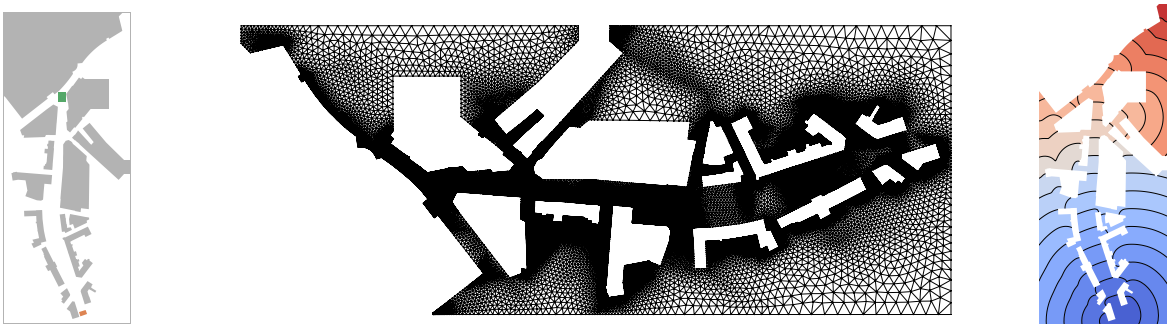


(iii) Richard-Wagner-Straße (t_{2000})

Figure 9.16: Narrow band size comparison of the FIM and IFIM for specific time steps t_j at time $t_j \cdot \Delta t$ into the simulation: for t_{50} the narrow band sizes for the corridor scenario are similar (i). For the bottleneck scenario (ii), and the Richard-Wagner-Straße (iii) this is no longer the case. Overall, fewer manipulations of the narrow band data structure are necessary using the IFIM.

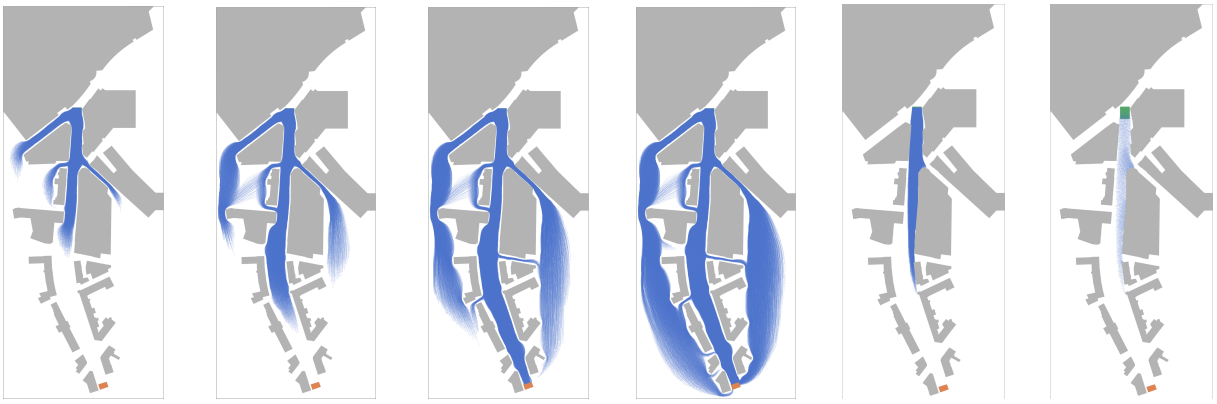
using a static navigation field, agents only use the main street to reach Γ. Since agents avoid each other only locally, they get stuck if a large crowd is in front, compare Fig. 9.18vi. And since they stick to the shortest path, they tend to use only a small part of the street. This effect can be observed in Fig. 9.18v. In comparison, the medium-scale navigation imposed by the dynamic navigation field leads to a more even distribution of agents for all possible paths.

There is still a lot of work necessary to find and calibrate suitable travel speed functions. I think there exists no function, which ideally models pedestrian behavior for all scenarios. However, in the case of the Richard-Wagner-Straße, the dynamic navigation field computed with the travel speed function presented in [165], results in more realistic pedestrian behavior than the static one. Thus, this travel speed functions seems to be a reasonable start.



(i) (ii) (iii)

Figure 9.17: The scenario (i), the underlying mesh (ii), and $\phi_{\Gamma,0}$ (iii) of the Richard-Wagner-Straße scenario: agents spawn inside the green rectangle and navigate to the orange area at the bottom. Apart from using the main street, there are multiple detours an agent chooses from.



(i) (ii) (iii) (iv) (v) (vi)

Figure 9.18: Snapshots of agent trajectories 125 s (i), 196 s (ii), 280 s (iii), and 363 s (iv) into the simulation of the Richard-Wagner-Straße scenario using a dynamic navigation field: trajectory plots (v) and agents (vi) after 393 s into the simulation using a static navigation field for the same scenario.

9.7 Source code

The source code for all discussed algorithms and data structures is part of the open-source simulation framework Vadere [294]. More specifically, numerical solvers are contained in the `solver` package of the `VadereSimulator` subproject. Class names indicate the solver it implements. For example, `MeshEikonalSolverIFIM.java.java` implements InformedFastIterativeMethod. I offer implementations of the LockFreeInformedFastIterativeMethod and implementations for the FastMarchingMethod, FastIterativeMethod on Cartesian grids and unstructured triangular meshes.

9.8 Summary

In this chapter, I discussed the computation of static and dynamic navigation fields for microscopic pedestrian simulation. I pursued two paths: (1) establish a connection between the mesh resolution and the eikonal equation and (2) develop an effective solver for dynamic navigation field computation. This required a deep understanding of the solving process on an algorithmic level.

Therefore, in the first three sections, I presented and compared state-of-the-art solvers and techniques to solve the eikonal equation. To deepen my and the reader's understanding of the problem, I started with a physical wave propagation analogy. In Section 9.2, I presented well-known finite difference schemes for Cartesian grids and unstructured meshes. In the following section, I also discussed and compared numerical methods such as the FastMarchingMethod and their extensions. This review establishes criteria to decide which numerical method fits the problem instance.

In Section 9.4, I established a connection between the mesh resolution and the curvature of the travel time surface. I used a curvature estimation technique in order to develop an iterative numerical method that solves the eikonal equation. The method starts with a coarse high-quality mesh generated by EikMesh, computes the travel time, estimates its curvature, and refines the mesh at highly curved areas. This process is repeated until no further mesh refinements are required. Besides this general approach, I connected ∇f to the mesh resolution and exploited the application, in my case, pedestrian simulation, to generate one static mesh or multiple dynamic meshes that change during a simulation run.

In Section 9.5, I developed an adaptation of the FastIterativeMethod (FIM), called InformedFastIterativeMethod (IFIM). It 'learns' and uses the wavefront propagation order to reduce the workload of dynamic navigation field computations. It outperforms the FIM and the FastMarchingMethod for the dynamic navigation field computation. The InformedFastIterativeMethod performs exceptionally well if the travel speed function f does not change too much over time. There might be many more applications outside of pedestrian dynamics that I am willing to discover in the future. Apart from the IFIM, this section also discusses the natural wavefront propagation and how it might lead to even more effective computations optimized for single instruction multiple data hardware architectures.

Section 9.6 concludes the chapter by comparing the newly developed IFIM against other methods. Additionally, I showed the effect of using a dynamic over a static navigation field. Three

different scenarios revealed the strength of the IFIM. In conclusion, there is no reason to use any other method for dynamic navigation field computation. The IFIM has the same parallel potential as the FIM and has approximately the same workload compared to the FMM.

In Section 9.7 I gave references to my source code.

CHAPTER 10

Discussion

"Attention is the rarest and purest form of generosity."

– Simone Weil

This final chapter summarizes my thesis, reviews what has been accomplished, reveals open questions, and gives an outlook on possible future directions.

10.1 Summary

This work aimed to enable realistic large-scale microscopic pedestrian simulations. Instead of designing a new simulation model, I focused on efficient algorithms to enhance the computation of existing ones. In the introduction (Chapter 1), I motivated my work and gave preparing information and advice to the reader.

In the first chapter (Chapter 2) of Part I, I reviewed the principle of most known microscopic models and the hierarchical structure they implement. I discussed the advantages and disadvantages and assessed the model's potential for large-scale simulations. Furthermore, I argued that many important models rely on either *floor field* or *navigation fields* and that medium- and long-range navigation realized by dynamic navigation fields is a robust wayfinding technique, especially for complex geometries. I suggested some improvements for optimal steps models at the end of Section 2.2.6.

In Chapter 3, I established the assumed relation between the cognitive map of a person and an agent's navigation field. Many researchers assume that pedestrians navigate through the environment by using an *optimal path*, but I pointed out that the term *optimal* is ambiguous. For many essential scenarios, modelers assume that agents move on the shortest or least time-consuming path. In that case, navigation fields offer a robust solution to the wayfinding problem. I introduced the mathematical framework of navigation fields and presented examples of static and dynamic navigation fields established in the community. Using the navigation field for the Behavioral Heuristics Model, I showcased that many models can benefit from a sophisticated computation of the destination direction using navigation fields.

Part II starts (Chapter 4) by differentiating between the inhomogeneity of the decision-making process and the homogeneous locomotion. I argued that the degree of homogeneity maps to the

degree of execution path divergence of the model's implementation. Consequently, I identified locomotion to be the model part that benefits the most from parallelism. The chapter ends with a review of existing parallel implementations of locomotion models.

In Chapter 5, I combined and developed efficient parallel algorithms to parallelize optimal steps models. I advocated against model changes for the sake of parallelism, because the model would lose essential properties. My algorithm (ParallelEventDrivenUpdate) introduces parallelism to optimal steps models and is especially suitable for single instruction multiple data architectures. I presented the degree of parallelism ParallelEventDrivenUpdate achieves. Additionally, I compared computation times of ParallelUpdate and ParallelEventDrivenUpdate, both executed on a graphics processing unit (GPU). ParallelUpdate is a former attempt to introduce parallelism but compromises optimal steps models for the sake of it. It acted as a baseline for the performance evaluation.

In Part III, I focused on the second computationally expensive part that has to be efficiently computed to enable large-scale simulations, i. e., navigation fields. I followed two ways: (1) a reduction of the problem size, i. e., the workload, and (2) the development of a new specialized and more efficient numerical method. To reduce the problem size, I proposed to discretize the spatial domain by high-quality meshes with a localized resolution. Since pedestrian simulations are sensitive to geometrical changes, I decided to use unstructured triangular meshes that adhere to the boundary domain. In Chapters 6 and 7, I introduced known meshing algorithms to generate these meshes and established metrics to compare them. Since DistMesh (Chapter 7) performs exceptionally well and is based on a straightforward physical analogy, I decided to use it for mesh generation. I showcased its performance and its flaws: an arbitrarily small minimal element quality, elements that do not align with the geometry, and high computational costs.

I tackled all these issues by introducing an adaptation called EikMesh. It can deal with an implicit geometry definition and with two-dimensional segment-bounded planar straight-line graphs (Chapter 8). EikMesh supports geometrical constraints, avoids Delaunay triangulation computations, and exploits a localized memory order of mesh elements. Compared to DistMesh, the time complexity is reduced, and its parallel potential increased. Its output always adheres to the boundary, and the quality of the worst mesh element is significantly increased for all test cases. Additionally, it generates meshes of higher quality and can deal with inputs, for which DistMesh fails to construct a proper mesh.

In the last chapter (Chapter 9), I focused on the actual computation of navigation fields and a suitable user-defined element size function h. I described how the eikonal equation is solved on Cartesian grids and triangular meshes. Furthermore, I proposed a localized mesh resolution that depends on the curvature of the eikonal equation's solution Φ. Since one has to compute Φ before knowing its curvature, I introduced an iterative eikonal solver that solves the equation on a series of more and more refined meshes. Different numerical methods promise efficiency. In practice, their performance relies on the problem instance. For example, methods inspired by the FastSweepingMethod only perform well for simple cases. The FastMarchingMethod performs well for all instances but operates inherently sequential. Therefore, I decided to look closer into the FastIterativeMethod, which is designed for massively parallel hardware, but its performance also suffers from turns of the propagating wavefront. InformedFastIterativeMethod eliminates the drawback of the FastIterativeMethod if consecutive similar eikonal equations have to be computed. Therefore, it suits the requirements for dynamic

navigation field computation. The new method 'learns' the propagation direction of the next wavefront by analyzing the current one. I showed that for n vertices, the InformedFastIterativeMethod requires almost exactly n updates, which is optimal. In addition, it inherits the parallel potential of the FastIterativeMethod.

10.2 Conclusion and outlook

In this thesis, I presented essential algorithms and data structures to parallelize locomotion models and to compute eikonal equations efficiently. Unstructured meshes with localized resolutions and parallel identification of independent agents led to efficient large-scale navigation field-based microscopic simulations. Despite their sequential nature, I was able to parallelize optimal steps models by exploiting data independence in the form of "unaware" agents. Model compromises were unnecessary, but it is also clear that reducing the agent's influence radius increases the number of agents that can be updated in parallel. This model modification is worth studying in future works.

In my experimental study, the portion of independent agents was relatively low. But since the identification of independent agents is computationally inexpensive, reasonable speedups were achievable. The GPU implementation showcased the parallel potential of ParallelEventDrivenUpdate for a system with many processing units. It achieved a massive speedup compared to the single-threaded version. Additional experimental studies are required to analyze the run time and scalability for different hardware systems.

With EikMesh we can construct high-quality unstructured triangular meshes that adhere to the domain boundary. I emphasized its parallel potential, because we got rid of the computation of Delaunay triangulations. The force-based improvement step was already executed in parallel, and edge flips can even be executed on the GPU, compare [208]. What is missing is a distributed memory implementation. One idea I had in mind was to partition the mesh into multiple connected sub-meshes and perform the improvement phase of EikMesh for each sub-domain independently. We could fix each sub-domain's boundary vertices, but we would need to change the partition during the phase to avoid artifacts. Alternatively, we can handle the boundary of each sub-domain by communicating the required information. A partition could also limit the execution of the improvement phase to an area, where elements are poorly shaped. These concepts are worth studying and might yield interesting results for the computational geometry community.

Using the proposed user-defined element size function, EikMesh increases the mesh resolution at geometrically important areas such as bottlenecks. I showed that the curvature is a good estimator for the mesh resolution and presented an iterative algorithm to solve the eikonal equation, assuming we have no special knowledge about the travel speed function f. In the future, I want to integrate this technique for dynamic navigation field computations. Because of the similarity of consecutive eikonal equations, I estimate that only a few mesh adaptations between successive computations are required to guarantee good accuracy. I also presented an alternative: mesh refinement at crowded areas. It is a first consideration and future work is necessary to test its potential. The method requires mesh coarsening and refinement periodically. In my implementation, it still drains too many resources to be executed for each time step. Further investigations and developments towards a dynamic mesh are required. The reader might ask why I did not

use EikMesh for the dynamic mesh resolution adaptation? There is potential. One idea I had in mind was to generate a first mesh with a higher resolution at crowded areas and then, from time to time, only execute the improvement phase of EikMesh. Using an appropriate dynamically changing element size h would lead to a mesh that 'moves' with the crowd. However, the devil lies in the details, and I could not test this idea in the scope of my thesis.

Iteratively solving the eikonal equation on a more and more refined mesh is a robust method to generate the initial mesh for dynamic navigation field computation. However, we have to solve the equation multiple times. In future works, I aim to enhance the method such that the refinement is executed while the wavefront propagates through the domain. In that case, the refinement would take place near the current *narrow band* of the numerical solver. Instead of solving the equation multiple times, we would solve it multiple times for some vertices but avoid a complete re-computation. It is an open question if such a technique led to better performance.

The new InformedFastIterativeMethod is an improvement over existing methods if one has to compute multiple similar eikonal equations. It is suitable to compute dynamic navigation fields since it exploits the similar direction of consecutive propagating wavefronts by 'learning' the *(defining) vertices* the wave comes from. In its current implementation, I assume that the underlying mesh did not change and whenever it does, I 'unlearn' everything. Here is room for improvement. Instead of *defining vertices*, we could 'learn' and remember *directions*, that is, approximations of $\nabla\Phi_{\Gamma,i}$ and identify the *defining vertices* (of $\nabla\Phi_{\Gamma,i+1}$) based on $\nabla\Phi_{\Gamma,i}$. Identifying the *defining vertices* would require additional arithmetic operations but might still lead to a faster computation overall. Furthermore, we could use this as a fall-back strategy, i. e., whenever *defining vertices* are unavailable. A GPU implementation for the InformedFastIterativeMethod is missing, but since it is very similar to the FastIterativeMethod it should be straightforward. If f can also be computed on the GPU, we could eliminate most of the memory transfers between the host (CPU) and the device (GPU) during pedestrian simulations. A more extensive performance analysis comparing different massively parallel solvers on different hardware settings also belongs to future works.

For a problem instance, where the wavefront direction stays similar for multiple consecutive eikonal equations, the NaturalMarchingMethod should be considered. The development of InformedFastIterativeMethod led me to this concept at the end of my work. There was no time left to implement and test it, but in my opinion, it might be a powerful technique. It has the potential to work exceptionally well on massively parallel hardware systems.

I showed the influence of the dynamic navigation field in a large-scale setting by an example. I also showed that in some cases, a higher mesh resolution does not necessarily change the result, that is, the agents' trajectory. However, for a curved Φ, the mesh resolution matters. My proposed curvature dependent element size function is a step in the right direction. In the future, we have to investigate in more detail what effect the mesh resolutions on the simulation output has.

As my last words, I want to stress that numerous fields might find relevant ideas in my work. Besides topics connected to large-scale pedestrian simulation, I looked closely into unstructured two-dimensional mesh generation and numerical methods to solve the eikonal equation. I hope that researchers outside of the pedestrian dynamics community dealing with these two topics also find fruitful ideas in this thesis.

Bibliography

[1] IntelliJ. URL https://www.jetbrains.com/idea/.

[2] Ipe. URL http://ipe.otfried.org/.

[3] MacTeX. URL https://www.tug.org/mactex/.

[4] Apache Maven. URL https://maven.apache.org/.

[5] TeXstudio. URL https://www.texstudio.org/.

[6] OpenSteer, 2004. URL http://opensteer.sourceforge.net/.

[7] Vadere crowd simulation, 2016. URL https://gitlab.lrz.de/vadere/vadere.

[8] S2UCRE, 2017. URL www.s2ucre.de.

[9] David Abramson, Michael Lees, Valeria Krzhizhanovskaya, Jack Dongarra, Peter M.A. Sloot, Florian Dang, and Nahid Emad. Fast iterative method in solving eikonal equations: a multi-level parallel approach. *Procedia Computer Science*, 29:1859–1869, 2014. ISSN 1877-0509. doi: 10.1016/j.procs.2014.05.170.

[10] Salem F. Adra, Simon Coakley, Mariam Kiran, and Phil McMinn. An agent-based software platform for modelling systems biology, 2008.

[11] Lyuba Alboul and Rudolf M.J. van Damme. Polyhedral metrics in surface reconstruction. In *Proc. Math. of Surfaces VI, Clarendon press*, pages 171–200, 2 1996. ISBN 0-19-851198-1.

[12] Michael P. Allen and Dominic J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, Inc., USA, 2nd edition, 2017. ISBN 0198803206.

[13] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. *ACM Trans. Graph.*, 24(3): 617–625, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073238.

[14] N. Amenta and M. Bern. Surface reconstruction by Voronoi filtering. *Discrete & Computational Geometry*, 22:481 – 504, 1999. doi: 10.1007/PL00009475.

[15] Erik Andresen, David Haensel, Mohcine Chraibi, and Armin Seyfried. Wayfinding and cognitive maps for pedestrian models. In Victor L. Knoop and Winnie Daamen, editors, *Traffic and Granular Flow '15*, pages 249–256, Cham, 2016. Springer International Publishing. ISBN 978-3-319-33482-0.

[16] Gianluca Antonini, Michel Bierlaire, and Mats Weber. Discrete choice models of pedestrian walking behavior. *Transportation Research Part B: Methodological*, 40(8):667–687, 2006. doi: 10.1016/j.trb.2005.09.006.

[17] Adrain F. Aveni. The not-so-lonely crowd: friendship groups in collective behavior. *Sociometry*, 40(1):96–99, 1977. doi: 10.2307/3033551.

[18] Jakob Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Curvature in Triangle Meshes*, pages 143–158. Springer London, London, 2012. ISBN 978-1-4471-4075-7. doi: 10.1007/978-1-4471-4075-7_8.

[19] Stefania Bandini, Giancarlo Mauri, and Giuseppe Vizzari. Supporting action-at-a-distance in situated cellular agents. *Fundam. Inf.*, 69(3): 251–271, August 2005. ISSN 0169-2968.

[20] Stefania Bandini, Federico Rubagotti, Giuseppe Vizzari, and Kenichiro Shimura. A cellular automata based model for pedestrian and group dynamics: motivations and first experiments. In *Parallel Computing Technologies*, pages 125–139, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23178-0.

[21] Jörn Behrens and Michael Bader. Efficiency considerations in triangular adaptive mesh refinement. *Philosophical Transactions of the Royal Society A*, 367:4577–4589, October 2009. Theme Issue 'Mesh generation and mesh adaptation for large-scale Earth-system modelling'.

[22] Nicola Bellomo and Abdelghani Bellouquid. On multiscale models of pedestrian crowds from mesoscopic to macroscopic. *Communications in Mathematical Sciences*, 13(7):1649–1664, 2015. ISSN 1945-0796. doi: 10.4310/cms.2015.v13.n7.a1.

[23] Nicola Bellomo, Benedetto Piccoli, and Andrea Tosin. Modeling crowd dynamics from a complex system viewpoint. *Mathematical Models and Methods in Applied Sciences*, 22(supp02):1230004–1–1230004–29, 2012. doi: 10.1142/S0218202512300049.

[24] Aniket Bera, Sujeong Kim, and Dinesh Manocha. Online parameter learning for data-driven crowd simulation and content generation. *Computers and Graphics*, 55:68–79, 2016. doi: 10.1016/j.cag.2015.10.009.

[25] Aniket Bera, Sujeong Kim, and Dinesh Manocha. Modeling trajectory-level behaviors using time varying pedestrian movement dynamics. *Collective Dynamics*, 2018. doi: 10.17815/CD.2018.15.

[26] Jur Berg, Stephen J. Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. *Springer Tracts in Advanced Robotics*, 70: 3–19, 2011. ISSN 1610-742X. doi: 10.1007/978-3-642-19457-3_1.

[27] Marshall Bern and Paul Plassmann. Chapter 6 - Mesh generation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 291–332. North-Holland, Amsterdam, 2000. ISBN 978-0-444-82537-7. doi: 10.1016/B978-044482537-7/50007-3.

[28] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384–409, 1994. ISSN 0022-0000. doi: 10.1016/S0022-0000(05)80059-5.

[29] Daniel H. Biedermann, Carolin Torchiani, Peter M. Kielar, David Willems, Oliver Handel, Stefan Ruzika, and André Borrmann. A hybrid and multiscale approach to model and simulate mobility in the context of public events. *Transportation Research Procedia*, 19:350–363, 2016. ISSN 2352-1465. doi: 10.1016/j.trpro.2016.12.094. Transforming Urban Mobility. mobil.TUM 2016. International Scientific Conference on Mobility and Transport. Conference Proceedings.

[30] Ted D. Blacker and Michael B. Stephenson. Paving: a new approach to automated quadrilateral mesh generation. *International Journal for Numerical Methods in Engineering*, 32(4):811–847, 1991. doi: 10.1002/nme.1620320410.

[31] Victor J. Blue and Jeffrey L. Adler. Cellular automata microsimulation for modeling bi-directional pedestrian walkways. *Transportation Research Part B: Methodological*, 35:293–312, 2001. doi: 10.1016/S0191-2615(99)00052-1.

[32] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of SIGGRAPH 95*, 1995.

[33] Nino Boccara. *Modeling Complex Systems*. Graduate Texts in Physics. Springer, New York, 2nd edition, 2010.

[34] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the Delaunay tree. *Theoretical Computer Science*, 112 (2):339 – 354, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90024-N.

[35] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 01 1981. ISSN 0010-4620. doi: 10.1093/comjnl/24.2.162.

[36] M. Breuß, E. Cristiani, P. Gwosdek, and O. Voge. A domain-decomposition-free parallelisation of the Fast Marching Method. *Applied Mathematics and Computation*, 2009.

[37] Kevin Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223 – 228, 1979. ISSN 0020-0190. doi: 10.1016/0020-0190(79)90074-7.

[38] Jakub Bujas, Dawid Dworak, Wojciech Turek, and Aleksander Byrski. High-performance computing framework with desynchronized information propagation for large-scale simulations. *Journal of Computational Science*, 32:70 – 86, 2019. ISSN 1877-7503. doi: 10.1016/j.jocs.2018.09.004.

[39] Hans-Joachim Bungartz, Stefan Zimmer, Martin Buchholz, and Dirk Pflüger. *Modeling and Simulation: An Application-Oriented Introduction*. Springer Undergraduate Texts in Mathematics and Technology. Springer, Berlin Heidelberg, 2014. doi: 10.1007/978-3-642-39524-6.

[40] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295:507–525, 2001. doi: 10.1016/S0378-4371(01)00141-8.

[41] Bernard Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10(4):377–409, Dec 1993. ISSN 1432-0444. doi: 10.1007/BF02573985.

[42] Siu-Wing Cheng and Tamal K. Dey. Quality meshing with weighted Delaunay refinement. *SIAM Journal on Computing*, 33(1):69–93, 2003. doi: 10.1137/S0097539703418808.

[43] Siu-Wing Cheng, Tamal K. Dey, and Joshua A. Levine. A practical Delaunay meshing algorithm for a large class of domains. In *Proceedings of the 16th International Meshing Roundtable*, pages 477–494, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-75103-8.

[44] Siu-Wing Cheng, Tamal K. Dey, and Edgar A. Ramos. Delaunay refinement for piecewise smooth complexes. *Discrete & Computational Geometry*, 43(1):121–166, Jan 2010. ISSN 1432-0444. doi: 10.1007/s00454-008-9109-3.

[45] L. Paul Chew. Guaranteed-quality triangular meshes. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, 1989.

[46] L. Paul Chew. Guaranteed-quality Delaunay meshing in 3d (short version). In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pages 391–393, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918789. doi: 10.1145/262839.263018.

[47] Woo-Young Choi, Dae-Young Kwak, Il-Heon Son, and Yong-Taek Im. Tetrahedral mesh generation based on advancing front technique and optimization scheme. *International Journal for Numerical Methods in Engineering*, 58(12):1857–1872, 2003. doi: 10.1002/nme.840.

[48] Mohcine Chraibi, Armin Seyfried, and Andreas Schadschneider. Generalized Centrifugal-force Model for pedestrian dynamics. *Physical Review E*, 82(4):046111, 2010. doi: 10.1103/PhysRevE.82.046111.

[49] Mohcine Chraibi, Ulrich Kemloh, Andreas Schadschneider, and Armin Seyfried. Force-based models of pedestrian dynamics. *Networks and Heterogeneous Media*, 6(3):425–442, 2011. doi: 10.3934/nhm.2011.6.425.

[50] James S. Coleman and John James. The equilibrium size distribution of freely-forming groups. *Sociometry*, 24(1):36–45, 1961.

[51] B. Cosenza, G. Cordasco, R. De Chiara, and V. Scarano. Distributed load balancing for parallel agent-based simulations. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 62–69, Feb 2011. doi: 10.1109/PDP.2011.22.

[52] Paul Covello and Garry Rodrigue. A generalized front marching algorithm for the solution of the eikonal equation. *Journal of Computational and Applied Mathematics*, 156(2):371 – 388, 2003. ISSN 0377-0427. doi: 10.1016/S0377-0427(03)00360-1.

[53] Paul Covello and Garry Rodrigue. Solving the eikonal equation on an adaptive mesh. *Applied Mathematics and Computation*, 166(3):678 – 695, 2005. ISSN 0096-3003. doi: 10.1016/j.amc.2004.06.061.

[54] Michael G. Crandall and Pierre-Louis Lions. Viscosity solutions of Hamilton-Jacobi equations. *Transactions of the American Mathematical Society*, 277(1):1–42, 1983. doi: 10.2307/1999343.

[55] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in heat: a new approach to computing distance based on heat flow. *ACM Transactions on Graphics*, 32(5), 2013. doi: 10.1145/2516971.2516977.

[56] Emiliano Cristiani, Benedetto Piccoli, and Andrea Tosin. Multiscale modeling of granular flows with application to crowd dynamics. *Multiscale Modeling & Simulation*, 9(1):155–182, 1 2011. ISSN 1540-3467. doi: 10.1137/100797515.

[57] Sean Curtis and Dinesh Manocha. Pedestrian simulation using geometric reasoning in velocity space. In *Pedestrian and Evacuation Dynamics 2012*, pages 875–890. Springer International Publishing, 2014. doi: 10.1007/978-3-319-02447-9_73.

[58] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3 edition, 2008.

[59] P. Degond, C. Appert-Rolland, M. Moussaïd, J. Pettré, and G. Theraulaz. A hierarchy of heuristic-based models of crowd dynamics. *Journal of Statistical Physics*, 152(6):1033–1068, 2013. doi: 10.1007/s10955-013-0805-x.

[60] Boris Delaunay. Sur la sphère vide. *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk*, 7:793–800, 1934.

[61] Miles Detrixhe, Frederic Gibou, and Chohong Min. A parallel fast sweeping method for the eikonal equation. *Journal of Computational Physics*, 237:46–55, 2013. doi: 10.1016/j.jcp.2012.11.042.

[62] Olivier Devillers. The Delaunay Hierarchy. *International Journal of Foundations of Computer Science*, 13(02):163–180, 2002. doi: 10.1142/S0129054102001035.

[63] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry*, SCG '01, pages 106–114, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113357X. doi: 10.1145/378583.378643.

[64] Luc Devroye, Christophe Lemaire, and Jean-Michel Moreau. Expected time analysis for Delaunay point location. *Computational Geometry*, 29(2):61 – 89, 2004. ISSN 0925-7721. doi: 10.1016/j.comgeo.2004.02.002.

[65] M. T. Dickerson and R. S. Drysdale. Fixed-radius near neighbors search algorithms for points and segments. *Inf. Process. Lett.*, 35(5): 269–273, August 1990. ISSN 0020-0190. doi: 10.1016/0020-0190(90)90056-4.

[66] M. T. Dickerson, R. L. Drysdale, and J. R. Sack. Simple algorithms for enumerating interpoint distances and finding k nearest neighbors. *Internat. J. Comput. Geom. Appl.*, 2(3):221–239, 1992.

[67] Matthew T. Dickerson and R. L. Scot Drysdale. Enumerating k distances for n points in the plane. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, SCG '91, pages 234–238, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914260. doi: 10.1145/109648.109674.

[68] Felix Dietrich and Gerta Köster. Gradient Navigation Model for pedestrian dynamics. *Physical Review E*, 89(6):062801, 2014. doi: 10.1103/PhysRevE.89.062801.

[69] Felix Dietrich, Florian Albrecht, and Gerta Köster. Surrogate models for bottleneck scenarios. In *Proceedings of the 8th International Conference on Pedestrian and Evacuation Dynamics (PED2016)*, Hefei, China, 2016.

[70] Felix Dietrich, Florian Künzner, Tobias Neckel, Gerta Köster, and Hans-Joachim Bungartz. Fast and flexible uncertainty quantification through a data-driven surrogate model. *International Journal for Uncertainty Quantification*, 8:175–192, 2018. doi: 10.1615/Int.J.UncertaintyQuantification.2018021975.

[71] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. doi: 10.1007/BF01386390.

[72] H. Dong, M. Zhou, Q. Wang, X. Yang, and F. Wang. State-of-the-art pedestrian and evacuation dynamics. *IEEE Transactions on Intelligent Transportation Systems*, 21(5):1849–1866, 2020. doi: 10.1109/TITS.2019.2915014.

[73] Marc Droske, Bernhard Meyer, Martin Rumpf, and Carlo Schaller. An adaptive level set method for medical image segmentation. In *Information Processing in Medical Imaging*, pages 416–422, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45729-9.

[74] Dorine Duives, Winnie Daamen, and Serge Hoogendoorn. Anticipation behavior upstream of a bottleneck. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 43–50, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.007.

[75] Dorine C. Duives, Winnie Daamen, and Serge P. Hoogendoorn. State-of-the-art crowd motion simulation models. *Transportation Research Part C: Emerging Technologies*, 37(0):193–209, 2013. doi: 10.1016/j.trc.2013.02.005.

[76] Dorine C. Duives, Winnie Daamen, and Serge P. Hoogendoorn. Continuum modelling of pedestrian flows - part 2: sensitivity analysis featuring crowd movement phenomena. *Physica A: Statistical Mechanics and its Applications*, 447:36–48, 2016. doi: 10.1016/j.physa.2015.11.025.

[77] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, Mar 1996. ISSN 1432-0541. doi: 10.1007/BF01975867.

[78] Arne D. Ekstrom, Michael J. Kahana, Jeremy B. Caplan, Tony A. Fields, Eve A. Isham, Ehren L. Newman, and Itzhak Fried. Cellular networks underlying human spatial navigation. *Nature*, 425(6954):184–188, 2003.

[79] Colin Ellard. *Where am I? Why we can find our way to the moon but get lost in the mall.* HarperCollins Publishers, 2009. ISBN 9781554683949.

[80] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient GPU implementation for large scale individual-based simulation of collective behavior. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 51–58, Oct 2009. doi: 10.1109/HiBi.2009.11.

[81] Claudio Feliciani and Katsuhiro Nishinari. An improved cellular automata model to simulate the behavior of high density crowd and validation by experimental data. *Physica A: Statistical Mechanics and its Applications*, 451:135–148, 2016. doi: 10.1016/j.physa.2016.01.057.

[82] David A. Field. Qualitative measures for initial meshes. *International Journal for Numerical Methods in Engineering*, 47(4):887–906, 2000. doi: 10.1002/(SICI)1097-0207(20000210)47:4<887::AID-NME804>3.0.CO;2-H.

[83] P. Fiorini and Z. Shiller. Motion planning in dynamic environments using the relative velocity paradigm. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, volume 1, pages 560–565, 1993. doi: 10.1109/ROBOT.1993.292038.

[84] S Fortune. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM. ISBN 0-89791-194-6. doi: 10.1145/10515.10549.

[85] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1):153, Nov 1987. ISSN 1432-0541. doi: 10.1007/BF01840357.

[86] Libi Fu, Weiguo Song, and Siuming Lo. A fuzzy-theory-based behavioral model for studying pedestrian evacuation from a single-exit room. *Physics Letters A*, 380(34):2619 – 2627, 2016. ISSN 0375-9601. doi: 10.1016/j.physleta.2016.06.011.

[87] Zhisong Fu, Won-Ki Jeong, Yongsheng Pan, Robert M. Kirby, and Ross T. Whitaker. A fast iterative method for solving the eikonal equation on triangulated surfaces. *SIAM J. Sci. Comput.*, 33(5):2468–2488, oct 2011. doi: 10.1137/100788951.

[88] Zhisong Fu, Robert Michael Kirby, and Ross T. Whitaker. A fast iterative method for solving the eikonal equation on tetrahedral domains. *SIAM journal on scientific computing: a publication of the Society for Industrial and Applied Mathematics*, 35 5:c473–c494, 2013.

[89] D. Ganellari and G. Haase. Fast many-core solvers for the eikonal equations in cardiovascular simulations. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 278–285, July 2016. doi: 10.1109/HPCSim.2016.7568347.

[90] Daniel Ganellari, Gundolf Haase, and Gerhard Zumbusch. A massively parallel eikonal solver on unstructured meshes. *Computing and Visualization in Science*, Feb 2018. ISSN 1433-0369. doi: 10.1007/s00791-018-0288-z.

[91] John Alan George. *Computer Implementation of the Finite Element Method.* PhD thesis, Stanford University, Stanford, CA, USA, 1971. AAI7205916.

[92] Roland Geraerts and Mark H. Overmars. The corridor map method: a general framework for real-time high-quality path planning. *Computer Animation and Virtual Worlds*, 18(2):107–119, 2007. doi: 10.1002/cav.166.

[93] R. Ghanem, H. Owhadi, and D. Higdon, editors. *Handbook of uncertainty quantification.* Cham : Springer International Publishing, 2017. doi: 10.1007/978-3-319-12385-1.

[94] Gerd Gigerenzer. Why heuristics work. *Perspectives on Psychological Science*, 3(1):20–29, 2008. doi: 10.1111/j.1745-6916.2008.00058.x.

[95] Gerd Gigerenzer, Peter M. Todd, and A.B.C. Research Group. *Simple Heuristics That Make Us Smart.* Oxford University Press, Oxford, 1999.

[96] Peter G. Gipps and Bertil S. Marksjö. A micro-simulation model for pedestrian flows. *Mathematics and Computers in Simulation*, 27(2–3): 95–105, 1985. doi: 10.1016/0378-4754(85)90027-8.

[97] Git Contributors. Git, 2015. URL https://www.git-scm.com/.

[98] Marion Gödel, Rainer Fischer, and Gerta Köster. Applying Bayesian inversion with Markov Chain Monte Carlo to Pedestrian Dynamics. In *UNCECOMP 2019, 3rd ECCOMAS Thematic Conference on Uncertainty Quantification in Computational Sciences and Engineering*, 2019. doi: 10.7712/120219.6322.18561.

[99] Marion Gödel, Rainer Fischer, and Gerta Köster. Towards inferring input parameters from measurements: Bayesian inversion for a bottleneck scenario. In *Traffic and Granular Flow 2019*, pages 93–102, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55973-1.

[100] Marion Gödel, Gerta Köster, Daniel Lehmberg, Manfred Gruber, Angelika Kneidl, and Florian Sesser. Can we learn where people go? *Collective Dynamics*, 2020. doi: 10.17815/CD.2020.43.

[101] Daniel G. Goldstein and Gerd Gigerenzer. Models of ecological rationality: the recognition heuristic. *Psychological Review*, 109(1):75–90, 2002. doi: 10.1037/0033-295X.109.1.75.

[102] Reginald G. Golledge, R. Daniel Jacobson, Robert Kitchin, and Mark Blades. Cognitive maps, spatial abilities, and human wayfinding. *Geographical review of Japan, Series B.*, 73(2):93–104, 2000. doi: 10.4157/grj1984b.73.93.

[103] Javier V. Gómez, David Álvarez, Santiago Garrido, and Luis Moreno. Fast methods for eikonal equations: an experimental survey. *CoRR*, abs/1506.03771, 2015.

[104] Arne Graf. Automated routing in pedestrian dynamics. Master's thesis, Fachhochschule Aachen, 2015. URL http://juser.fz-juelich.de/record/276318.

[105] Simon Green. Particle simulation using CUDA, May 2010. URL https://developer.download.nvidia.com/assets/cuda/files/particles.pdf.

[106] Pierre A. Gremaud and Christopher M. Kuster. Computational study of fast methods for the eikonal equation. *SIAM Journal on Scientific Computing*, 27(6):1803–1816, 2006. doi: 10.1137/040605655.

[107] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Trans. Graph.*, 4(2):74–123, April 1985. ISSN 0730-0301. doi: 10.1145/282918.282923.

[108] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1):381–413, Jun 1992. ISSN 1432-0541. doi: 10.1007/BF01758770.

[109] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of Annual Meeting*, volume 14, pages 47–57. Association for Computing Machinery, 1984. doi: 10.1145/971697.602266.

[110] Stephen. J. Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. ClearPath: highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 177–187. ACM, 2009. doi: 10.1145/1599470.1599494.

[111] E.T. Hall. *The Silent language.* Doubleday, 1959.

[112] E.T. Hall. *The Hidden Dimension.* Anchor, 1990. ISBN 0385084765.

[113] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'10, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2.

[114] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.

[115] Dirk Hartmann. Adaptive pedestrian dynamics based on geodesics. *New Journal of Physics*, 12:043032, 2010. doi: 10.1088/1367-2630/12/4/043032.

[116] Dirk Hartmann, Jana Mille, Alexander Pfaffinger, and Christian Royer. Dynamic medium scale navigation using dynamic floor fields. In *Pedestrian and Evacuation Dynamics 2012*, pages 1237–1249. Springer International Publishing, 2014. doi: 10.1007/978-3-319-02447-9-102.

[117] Dirk Helbing. A fluid dynamic model for the movement of pedestrians. *Complex Systems*, 6:391–415, 1992.

[118] Dirk Helbing and Péter Molnár. Social Force Model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, 1995. doi: 10.1103/PhysRevE.51.4282.

[119] Dirk Helbing, Illés Farkas, and Tamás Vicsek. Simulating dynamical features of escape panic. *Nature*, 407:487–490, 2000. doi: 10.1038/35035023.

[120] L. F. Henderson. On the fluid mechanics of human crowd motion. *Transportation Research*, 8(6):509–515, 1974. ISSN 0041-1647. doi: https://doi.org/10.1016/0041-1647(74)90027-6.

[121] M. Herrmann. A domain decomposition parallelization of the fast marching method. Technical report, Stanford Univ. Center for Turbulence Research; Stanford, CA, United States, 2003.

[122] K. Hirai and K. Tarui. A simulation of the behavior of a crowd in panic. In *Proc. of the 1975 International Conference on Cybernetics and Society*, page 409, 1975.

[123] Mario Höcker, Volker Berkhahn, Angelika Kneidl, André Borrmann, and Wolfram Klein. Graph-based approaches for simulating pedestrian dynamics in building models. In *Proceedings of the European Conference on Product and Process Modelling 2010*, pages 389–394, Cork, Republic of Ireland, 2010. CRC Press. doi: 10.1201/b10527-65.

[124] S. Hong and W. K. Jeong. A group-ordered fast iterative method for eikonal equations. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2567397.

[125] Sumin Hong and Won-Ki Jeong. A multi-GPU fast iterative method for eikonal equations using on-the-fly adaptive domain decomposition. *Procedia Computer Science*, 80:190 – 200, 2016. doi: 10.1016/j.procs.2016.05.309. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[126] Serge P. Hoogendoorn and Piet H. L. Bovy. Pedestrian route-choice and activity scheduling theory and models. *Transportation Research Part B: Methodological*, 38(2):169–190, 2004. doi: 10.1016/S0191-2615(03)00007-9.

[127] Serge P. Hoogendoorn and Winnie Daamen. Microscopic calibration and validation of pedestrian models: cross-comparison of models using experimental data. In *Traffic and Granular Flow '05*, pages 329–340, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-47641-2.

[128] Serge P. Hoogendoorn, Femke L.M. van Wageningen-Kessels, Winnie Daamen, and Dorine C. Duives. Continuum modelling of pedestrian flows: from microscopic principles to self-organised macroscopic phenomena. *Physica A: Statistical Mechanics and its Applications*, 416:684 – 694, 2014. ISSN 0378-4371. doi: 0.1016/j.physa.2014.07.050.

[129] R.L. Hughes. The flow of large crowds of pedestrians. *Mathematics and Computers in Simulation*, 53(4):367–370, 2000. doi: 10.1016/S0378-4754(00)00228-7.

[130] Roger L. Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36(6):507–535, 2001. ISSN 0191-2615. doi: 10.1016/S0191-2615(01)00015-7.

[131] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[132] Yasushi Ito, Alan M. Shih, and Bharat K. Soni. Reliable isotropic tetrahedral mesh generation based on an advancing front method. In *IMR*, 2004.

[133] Yasushi Ito, Alan M. Shih, and Bharat K. Soni. Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates. *International Journal for Numerical Methods in Engineering*, 77(13):1809–1833, 2009. doi: 10.1002/nme.2470.

[134] John James. The distribution of free-forming small group size. *American Sociological Review*, 18(5):569–570, 1953.

[135] Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. *SIAM Journal of Scientific Computing*, 30(5):2512–2534, 2008. doi: 10.1137/060670298.

[136] B. Joe. Three-dimensional triangulations from local transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(4):718–741, 1989. doi: 10.1137/0910044.

[137] Anders Johansson, Dirk Helbing, and Pradyumn Shukla. Specification of the Social Force Pedestrian Model by evolutionary adjustment to video tracking data. *Advances in Complex Systems*, 10:271–288, 2007. doi: 10.1142/S0219525907001355.

[138] Fredrik Johansson, Dorine Duives, Winnie Daamen, and Serge Hoogendoorn. The many roles of the relaxation time parameter in force based models of pedestrian dynamics. *Transportation Research Procedia*, 2:300–308, 2014. ISSN 2352-1465. doi: 10.1016/j.trpro.2014.09.057. The Conference on Pedestrian and Evacuation Dynamics 2014 (PED 2014), 22-24 October 2014, Delft, The Netherlands.

[139] Fredrik Johansson, Anders Peterson, and Andreas Tapani. Waiting pedestrians in the Social Force Model. *Physica A: Statistical Mechanics and its Applications*, 419:95–107, 2015. doi: 10.1016/j.physa.2014.10.003.

[140] M. W. Jones, J. A. Baerentzen, and M. Sramek. 3d distance fields: a survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, July 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.56.

[141] Eunjung Ju, Myung Geol Choi, Minji Park, Jehee Lee, Kang Hoon Lee, and Shigeo Takahashi. Morphable crowds. *ACM Trans. Graph.*, 29 (6), December 2010. ISSN 0730-0301. doi: 10.1145/1882261.1866162.

[142] Ioannis Karamouzas and Mark Overmars. A velocity-based approach for simulating human collision avoidance. In *Intelligent Virtual Agents*, pages 180–186, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15892-6.

[143] Twin Karmakharm, Paul Richmond, and Daniela M. Romano. Agent-based large scale simulation of pedestrians with adaptive realistic navigation vector fields. In *Theory and Practice of Computer Graphics*, pages 67–74. The Eurographics Association, 2010. doi: 10.2312/LocalChapterEvents/TPCG/TPCG10/067-074.

[144] Ulrich Kemloh. *Route Choice Modelling and Runtime Optimisation for Simulation of Building Evacuation.* PhD thesis, Bergische Universiät Wuppertal, 2013.

[145] Vitalina Kharchenko. Optimization with evolution strategies. Bachelor's thesis, Hochschule München, June 2014.

[146] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1): 90–98, 1986. doi: 10.1177/027836498600500106.

[147] Oussama Khatib. *The Potential Field Approach And Operational Space Formulation In Robot Control*, pages 367–377. Springer US, Boston, MA, 1986. ISBN 978-1-4757-1895-9. doi: 10.1007/978-1-4757-1895-9_26.

[148] Peter M. Kielar and André Borrmann. Spice: a cognitive agent framework for computational crowd simulations in complex environments. *Autonomous Agents and Multi-Agent Systems*, 32(3):387–416, 2018. ISSN 1387-2532. doi: 10.1007/s10458-018-9383-2.

[149] Peter M. Kielar, Oliver Handel, Daniel H. Biedermann, and André Borrmann. Concurrent hierarchical finite state machines for modeling pedestrian behavioral tendencies. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 576–584, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.098.

[150] Seongjai Kim. An O(n) level set method for eikonal equations. *SIAM Journal on Scientific Computing*, 22(6):2178–2193, 2001. doi: 10.1137/S1064827500367130.

[151] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences of the United States of America*, 95(15):8431–8435, 1998. doi: 10.1073/pnas.95.15.8431.

[152] Ansgar Kirchner and Andreas Schadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. *Physica A: Statistical Mechanics and its Applications*, 312(1):260–276, 2002. ISSN 0378-4371. doi: 10.1016/S0378-4371(02)00857-9.

[153] Ansgar Kirchner, Katsuhiro Nishinari, and Andreas Schadschneider. Friction effects and clogging in a cellular automaton model for pedestrian dynamics. *Physical Review E*, 67:056122, 2003. doi: 10.1103/PhysRevE.67.056122.

[154] Ansgar Kirchner, Hubert Klüpfel, Katsuhiro Nishinari, Andreas Schadschneider, and Michael Schreckenberg. Discretization effects and the influence of walking speed in cellular automata models for pedestrian dynamics. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10011, 2004. doi: 10.1088/1742-5468/2004/10/P10011.

[155] Ekaterina Kirik, Tat'yana Yurgel'yan, and Dmitriy Krouglov. An intelligent floor field cellular automation model for pedestrian dynamics. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, pages 21:1–21:6, 2007. ISBN 1-56555-316-0.

[156] Ekaterina S. Kirik, Tatyana B. Yurgelyan, and Dmitriy V. Krouglov. The shortest time and/or the shortest path strategies in a CA FF pedestrian dynamics model. *Mathematics and Physics*, 2(3):271–278, 2009.

[157] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, and Gerta Köster. Vadere: an open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4, 2019. doi: 10.17815/CD.2019.21.

[158] Benedikt Kleinmeier, Gerta Köster, and John Drury. Agent-based simulation of collective cooperation: from experiment to model. *Journal of The Royal Society Interface*, 17(171), 2020. doi: 10.1098/rsif.2020.0396.

[159] Adrian Kłusek, Paweł Topa, Jarosław Wąs, and Robert Lubaundefined. An implementation of the Social Distances Model using multi-GPU systems. *Int. J. High Perform. Comput. Appl.*, 32(4):482–495, 2018. ISSN 1094-3420. doi: 10.1177/1094342016679492.

[160] Angelika Kneidl. *Methoden zur Abbildung menschlichen Navigationsverhaltens bei der Modellierung von Fußgängerströmen.* PhD thesis, Technische Universität München, 2013.

[161] Angelika Kneidl, André Borrmann, and Dirk Hartmann. Generation and use of sparse navigation graphs for microscopic pedestrian simulation models. *Advanced Engineering Informatics*, 26(4):669–680, 2012. doi: 10.1016/j.aei.2012.03.006.

[162] Angelika Kneidl, Dirk Hartmann Hartmann, and André Borrmann. A hybrid multi-scale approach for simulation of pedestrian dynamics. *Transportation Research Part C: Emerging Technologies*, 37:223–237, 2013. doi: 10.1016/j.trc.2013.03.005.

[163] Moonsoo Ko, Taewan Kim, and Keemin Sohn. Calibrating a social-force-based pedestrian walking model based on maximum likelihood estimation. *Transportation*, 40(1):91–107, Jan 2013. ISSN 1572-9435. doi: 10.1007/s11116-012-9411-z.

[164] Jonas Koko. A MATLAB mesh generator for the two-dimensional finite element method. *Applied Mathematics and Computation*, 250:650 – 664, 2015. ISSN 0096-3003. doi: 10.1016/j.amc.2014.11.009.

[165] Gerta Köster and Benedikt Zönnchen. Queuing at bottlenecks using a dynamic floor field for navigation. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 344–352, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.029.

[166] Gerta Köster and Benedikt Zönnchen. A queuing model based on social attitudes. In *Traffic and Granular Flow '15*, pages 193–200, Nootdorp, the Netherlands, 2016. Springer International Publishing. doi: 10.1007/978-3-319-33482-0. 27–30 October 2015.

[167] Gerta Köster, Franz Treml, and Marion Gödel. Avoiding numerical pitfalls in social force models. *Physical Review E*, 87(6):063305, 2013. doi: 10.1103/PhysRevE.87.063305.

[168] Gerta Köster, Daniel Lehmberg, and Angelika Kneidl. Walking on stairs: experiment and model. *Phys. Rev. E*, 100:022310, Aug 2019. doi: 10.1103/PhysRevE.100.022310.

[169] Tobias Kretz. *Pedestrian Traffic - Simulation and Experiments.* PhD thesis, Universität Duisburg Essen, 2007.

[170] Tobias Kretz. Computation speed of the F.A.S.T. model. *CoRR*, abs/0911.2900, 2009.

[171] Tobias Kretz and Michael Schreckenberg. F.A.S.T. – floor field- and agent-based simulation tool. *arXiv*, 2006.

[172] Tobias Kretz, Marko Wölki, and Michael Schreckenberg. Characterizing correlations of flow oscillations at bottlenecks. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(02):P02005, 2006. doi: 10.1088/1742-5468/2006/02/P02005.

[173] Tobias Kretz, Cornelia Bönisch, and Peter Vortisch. Comparison of various methods for the calculation of the distance potential field. In *Pedestrian and Evacuation Dynamics 2008*, pages 335–346. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-04504-2_29.

[174] Tobias Kretz, Andree Große, Stefan Hengst, Lukas Kautzsch, Andrej Pohlmann, and Peter Vortisch. Quickest paths in simulations of pedestrians. *Advances in Complex Systems*, 14(05):733–759, 2011. doi: 10.1142/S0219525911003281.

[175] Florian Kuenzner, Tobias Neckel, and Isabella von Sivers. Practical use of Chaospy for pedestrian traffic simulations. In *SIAM UQ 2016*, 2016.

[176] Florian Künzner, Tobias Neckel, and Hans-Joachim Bungartz. Prediction and reduction of runtime in non-intrusive forward uq simulations. *SN Applied Sciences*, 2019. doi: 10.1007/s42452-019-1066-3.

[177] François Labelle and Jonathan Richard Shewchuk. Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. *ACM Transactions on Graphics*, 26(3):57.1–57.10, 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276448. Special issue on Proceedings of SIGGRAPH 2007.

[178] Taras I. Lakoba, D. J. Kaup, and Neal M. Finkelstein. Modifications of the Helbing-Molnár-Farkas-Vicsek Social Force Model for pedestrian evolution. *Simulation*, 81(5):339–352, 2005. doi: 10.1177/0037549705052772.

[179] Gregor Lämmel, Bernhard Steffen, and Armin Seyfried. Large scale and microscopic: a fast simulation approach for urban areas. In *2014 TRB Annual Meeting Proceedings*, pages 14–3890, Washington, Jan 2014. Transportation Research Board 93th Annual Meeting, Washington D.C. (USA), 12 Jan 2014 - 16 Jan 2014, Transportation Research Board Annual Meeting Online.

[180] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic, 7th edition, 1991. ISBN 978-0792391296.

[181] Charles L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3(4):365 – 372, 1972. ISSN 0012-365X. doi: 10.1016/0012-365X(72)90093-3.

[182] C.L. Lawson. Software for C^1 surface interpolation. In John R. Rice, editor, *Mathematical Software*, pages 161 – 194. Academic Press, 1977. ISBN 978-0-12-587260-7. doi: 10.1016/B978-0-12-587260-7.50011-X.

[183] D. T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, Jun 1980. ISSN 1573-7640. doi: 10.1007/BF00977785.

[184] Alon Lerner, Yiorgos Chrysanthou, and Dani Lischinski. Crowds by example. *Computer Graphics Forum*, 26(3):655–664, 2007. ISSN 0167-7055. doi: 10.1111/j.1467-8659.2007.01089.x.

[185] Kurt Lewin. *Field theory in social science: selected theoretical papers*. Harper, New York, 1951.

[186] Chi Wan Lim, Xiaofeng Yin, Tianyou Zhang, Senthil Kumar Selvaraj, Yi Su, Chi-Keong Goh, Alejandro Moreno, and Shahrokh Shahpar. Towards automatic blocking of shapes using evolutionary algorithm. *Computer-Aided Design*, 120:102798, 2020. ISSN 0010-4485. doi: 10.1016/j.cad.2019.102798.

[187] Baoxi Liu, Hong Liu, Hao Zhang, and Xin Qin. A social force evacuation model driven by video data. *Simulation Modelling Practice and Theory*, 84:190–203, 2018. doi: 10.1016/j.simpat.2018.02.007.

[188] R. Löhner. *Data Structures and Algorithms*, chapter 2, pages 7–33. Wiley-Blackwell, 2008. ISBN 9780470989746. doi: 10.1002/9780470989746.ch2.

[189] R. Löhner. *Grid Generation*, chapter 3, pages 35–107. Wiley-Blackwell, 2008. ISBN 9780470989746. doi: 10.1002/9780470989746.ch3.

[190] Rainald Löhner. On the modeling of pedestrian motion. *Applied Mathematical Modelling*, 34(2):366–382, 2010. doi: 10.1016/j.apm.2009.04.017.

[191] Rainald Löhner and Paresh Parikh. Generation of three-dimensional unstructured grids by the Advancing-front Method. *International Journal for Numerical Methods in Fluids*, 8(10):1135–1149, 1988. ISSN 1097-0363. doi: 10.1002/fld.1650081003.

[192] Rainald Löhner, Muhammad Baqui, Eberhard Haug, and Britto Muhamad. Real-time micro-modelling of a million pedestrians. *Engineering Computations*, 33(1):217–237, 2016. doi: 10.1108/EC-02-2015-0036.

[193] Rainald Löhner, E. Haug, Claudio Zinggerling, and Eugenio Oñate. Real-time micro-modelling of city evacuations. *Computational Particle Mechanics*, 5:71–86, 2018.

[194] Hong Luo, Seth Spiegel, and Rainald Löhner. Hybrid grid generation method for complex geometries. *AIAA Journal*, 48(11):2639–2647, 2010. doi: 10.2514/1.J050491.

[195] LWJGL. Lightweight java game library 3. URL https://www.lwjgl.org/.

[196] S. Maniccam. Traffic jamming on hexagonal lattice. *Physica A: Statistical Mechanics and its Applications*, 321(3–4):653–664, 2003. doi: 10.1016/S0378-4371(02)01549-2.

[197] Christina Maria Mayr. The Heat Method for geodesic distance computation on 2d domains. Master's thesis, Munich University of Applied Sciences, 2019.

[198] Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, SCG '92, pages 212–221, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915178. doi: 10.1145/142675.142720.

[199] Edvard I. Moser, Emilio Kropff, and May-Britt Moser. Place cells, grid cells, and the brain's spatial representation system. *Annual Review of Neuroscience*, 31(1):69–89, 2008. doi: 10.1146/annurev.neuro.31.061307.090723.

[200] Mehdi Moussaïd and Jonathan D. Nelson. Simple heuristics and the modelling of crowd behaviours. In *Pedestrian and Evacuation Dynamics 2012*, pages 75–90. Springer International Publishing, Cham, Switzerland, 2014. doi: 10.1007/978-3-319-02447-9_5.

[201] Mehdi Moussaïd, Dirk Helbing, Simon Garnier, Anders Johansson, Maud Combe, and Guy Theraulaz. Experimental study of the behavioural mechanisms underlying self-organization in human crowds. *Proceedings of the Royal Society B: Biological Sciences*, 276:2755–2762, 2009. doi: 10.1098/rspb.2009.0405.

[202] Mehdi Moussaïd, Niriaska Perozo, Simon Garnier, Dirk Helbing, and Guy Theraulaz. The walking behaviour of pedestrian social groups and its impact on crowd dynamics. *PLoS ONE*, 5(4):e10047, 2010. doi: 10.1371/journal.pone.0010047.

[203] Mehdi Moussaïd, Dirk Helbing, and Guy Theraulaz. How simple rules determine pedestrian behavior and crowd disasters. *Proceedings of the National Academy of Sciences*, 108(17):6884–6888, 2011. doi: 10.1073/pnas.1016507108.

[204] Hubert Mroz and Jaroslaw Was. Discrete vs. continuous approach in crowd dynamics modeling using GPU computing. *Cybernetics and Systems*, 45(1):25–38, 2014. doi: 10.1080/01969722.2014.862104.

[205] Hubert Mróz, Jarosław Wąs, and Paweł Topa. The use of GPGPU in continuous and discrete models of crowd dynamics. In *Parallel Processing and Applied Mathematics*, pages 679–688. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-55195-6. doi: 10.1007/978-3-642-55195-6_64.

[206] S. R. Musse and D. Thalmann. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001.

[207] Ken Nakanishi. Multibunch solutions of the differential-difference equation for traffic flow. *Physical Review E*, 62:3349–3355, 2000. doi: 10.1103/PhysRevE.62.3349.

[208] C.A. Navarro, N. Hitschfeld-Kahler, and E. Scheihing. A quasi-parallel GPU-based algorithm for Delaunay edge-flips. 03 2011.

[209] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[210] G. F. Newell. Nonlinear effects in the dynamics of car following. *Operations Research*, 9(2):209–229, 1961. doi: 10.1287/opre.9.2.209.

[211] Katsuhiro Nishinari, Ansgar Kirchner, Alireza Namazi, and Andreas Schadschneider. Extended floor field CA model for evacuation dynamics. *IEICE TRANSACTIONS on Information and Systems*, E87-D(3):726–732, March 2004.

[212] Shigeyuki Okazaki. A study of pedestrian movement in architectural space: part 1 pedestrian movement by the application of magnetic models. *Transactions of the Architectural Institute of Japan*, 283:111–119, 1979. doi: 10.3130/aijsaxx.283.0_111.

[213] John O'Keefe and Lynn Nadel. *The Hippocampus as a Cognitive Map*. Oxford: Clarendon Press, 1978. ISBN 0-19-857206-9.

[214] Jan Ondřej, Julien Pettré, Anne-Hélène Olivier, and Stéphane Donikian. A synthetic-vision based steering approach for crowd simulation. *ACM Transactions on Graphics*, 29(4):123:1–123:9, 2010. doi: 10.1145/1778765.1778860.

[215] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12 – 49, 1988. ISSN 0021-9991. doi: 10.1016/0021-9991(88)90002-2.

[216] Sebastien Paris, Julien Pettré, and Stéphane Donikian. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Computer Graphics Forum*, 26(3):665–674, 2007. ISSN 0167-7055. doi: 10.1111/j.1467-8659.2007.01090.x.

[217] Daniel R. Parisi, Marcelo Gilman, and Herman Moldovan. A modification of the Social Force Model can reproduce experimental data of pedestrian flows in normal conditions. *Physica A: Statistical Mechanics and its Applications*, 388(17):3600–3608, 2009. doi: 10.1016/j.physa.2009.05.027.

[218] Steven E. Pav and Noel Walkington. Robust three dimensional Delaunay refinement. In *Proceedings of the 13th International Meshing Roundtable, IMR 2004, Williamsburg, Virginia, USA, September 19-22, 2004*, pages 145–156, 2004.

[219] N. Pelechano, J. M. Allbeck, and N. I. Badler. Controlling individual agents in high-density crowd simulation. In *ACM SIGGRAPH/Eurographics Symposium on Computer animation*, 2007.

[220] S. Pellegrini, A. Ess, K. Schindler, and L. Van Gool. You'll never walk alone: modeling social behavior for multi-target tracking. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 261–268. IEEE, 2009. doi: 10.1109/ICCV.2009.5459260.

[221] Per-Olof Persson and Gilbert Strang. A simple mesh generator in MATLAB. *SIAM Review*, 46(2):329–345, 2004. doi: 10.1137/S0036144503429121.

[222] Julien Pettré, Jan Ondřej, Anne-Hélène Olivier, Armel Cretual, and Stéphane Donikian. Experiment-based modeling, simulation and validation of interactions between virtual walkers. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2009.

[223] Enrico Puppo and Daniele Panozzo. RGB Subdivision. *IEEE transactions on visualization and computer graphics*, 15(2):295–310, 2009. ISSN 1077-2626. doi: 10.1109/tvcg.2008.87.

[224] Jianliang Qian, Yong-Tao Zhang, and Hong-Kai Zhao. Fast sweeping methods for eikonal equations on triangular meshes. *SIAM Journal on Numerical Analysis*, 45:83–107, 2007. doi: 10.1137/050627083.

[225] Jianliang Qian, Yong-Tao Zhang, and Hong-Kai Zhao. A fast sweeping method for static convex Hamilton-Jacobi equations. *J. Sci. Comput.*, 31(1-2):237–271, may 2007. doi: 10.1007/s10915-006-9124-6.

[226] Wang Qiu-Dong. The global solution of the N-body problem. *Celestial Mechanics and Dynamical Astronomy*, 50(1):73–88, March 1990. ISSN 0923-2958. doi: 10.1007/BF00048987.

[227] Michael J. Quinn, Ronald A. Metoyer, and Katharina Hunter-Zaworski. Parallel implementation of the Social Forces Model. In *The Second International Conference in Pedestrian and Evacuation Dynamics*, 2003.

[228] A. Rahman, N. A. W. A. Hamid, A. R. Rahiman, and B. Zafar. Towards accelerated agent-based crowd simulation for Hajj and Umrah. In *2015 International Symposium on Agents, Multi-Agent Systems and Robotics (ISAMSR)*, pages 65–70, August 2015. doi: 10.1109/ISAMSR.2015.7379132.

[229] Alexander Rand and Noel Walkington. 3d Delaunay refinement of sharp domains without a local feature size oracle. In *Proceedings of the 17th International Meshing Roundtable*, pages 37–54, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87921-3.

[230] Christian Rasch and Thomas Satzger. Remarks on the O(n) implementation of the Fast Marching Method. *IMA Journal of Numerical Analysis*, pages 1–8, 2008. doi: 10.1093/imanum/drm028.

[231] S. Rebay. Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm. *Journal of Computational Physics*, 106(1):125–138, 1993. ISSN 0021-9991. doi: 10.1006/jcph.1993.1097.

[232] Paweł Renc, Maciej Bielech, Tomasz Pęcak, Piotr Morawiecki, Mateusz Paciorek, Wojciech Turek, Aleksander Byrski, and Jarosław Wąs. HPC large-scale pedestrian simulation based on proxemics rules. In *Parallel Processing and Applied Mathematics*, pages 489–499, Cham, 2020. Springer International Publishing. ISBN 978-3-030-43222-5.

[233] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4):25–34, 1987. doi: 10.1145/37402.37406.

[234] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, pages 763–782, San Jose, CA, 1999. Miller Freeman Game Group, San Francisco, CA.

[235] P. Richmond and D. Romano. A high performance framework for agent based pedestrian dynamics on GPU hardware. *European Simulation and Modelling*, 2008.

[236] M. Cecilia Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International Journal for Numerical Methods in Engineering*, 20(4):745–756, 1984. doi: 10.1002/nme.1620200412.

[237] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231, June 1985. ISSN 0163-5964. doi: 10.1145/327070.327215.

[238] Elisabeth Rouy and Agnes Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal on Numerical Analysis*, 29(3): 867–884, 1992. doi: 10.1137/0729053.

[239] Sergio Ruiz-Loza and Benjamin Hernandez. A hybrid reinforcement learning and cellular automata model for crowd simulation on the GPU. *High Performance Computing*, 2019. doi: DOI:10.1007/978-3-030-16205-4_5.

[240] Martin Rumpf and Alexandru Telea. A continuous skeletonization method based on level sets. In *Proceedings of the Symposium on Data Visualisation 2002*, VISSYM '02, Goslar, DEU, 2002. Eurographics Association. ISBN 158113536X.

[241] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548 – 585, 1995. ISSN 0196-6774. doi: 10.1006/jagm.1995.1021.

[242] Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 83–92, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0-89871-313-7.

[243] S. Sarmady, F. Haron, and A. Z. Talib. Simulating crowd movements using fine grid cellular automata. In *2020 12th International Conference on Computer Modelling and Simulation*, pages 428–433, 2010. doi: 10.1109/UKSIM.2010.85.

[244] A. Schadschneider and M. Schreckenberg. Cellular automation models and traffic flow. *Journal of Physics A: Mathematical and General*, 26 (15):L679, 1993. doi: 10.1088/0305-4470/26/15/011.

[245] Andreas Schadschneider. Cellular automaton approach to pedestrian dynamics – theory. In *Pedestrian and Evacuation Dynamics*, pages 75–86. Springer, 2001.

[246] Andreas Schadschneider, Ansgar Kirchner, and Katsuhiro Nishinari. CA approach to collective phenomena in pedestrian dynamics. In *Cellular Automata*, pages 239–248, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45830-2.

[247] Andreas Schadschneider, Wolfram Klingsch, Hubert Klüpfel, Tobias Kretz, Christian Rogsch, and Armin Seyfried. Evacuation dynamics: Empirical results, modeling and applications. In *Encyclopedia of Complexity and Systems Science*, pages 3142–3176. Springer, New York, 2009. ISBN 978-0-387-75888-6. doi: 10.1007/978-0-387-30440-3_187.

[248] Joachim Schöberl. NETGEN an advancing front 2d/3d-mesh generator based on abstract rules. *Computing and Visualization in Science*, 1 (1):41–52, Jul 1997. ISSN 1432-9360. doi: 10.1007/s007910050004.

[249] M. Schreckenberg, A. Schadschneider, K. Nagel, and N. Ito. Discrete stochastic models for traffic flow. *Physical Review E*, 51:2939–2949, 1995. doi: 10.1103/PhysRevE.51.2939.

[250] John M. Schreiner, Carlos Eduardo Scheidegger, Shachar Fleishman, and Cláudio T. Silva. Direct (re)meshing for efficient surface processing. *Comput. Graph. Forum*, 25:527–536, 2006.

[251] P. Scovanner and M. F. Tappen. Learning pedestrian dynamics from the real world. In *2009 IEEE 12th International Conference on Computer Vision*, pages 381–388, 2009.

[252] Stefan Seer. *A unified framework for evaluating microscopic pedestrian simulation models*. PhD thesis, Technische Universität Wien, Fakultät für Mathematik und Geoinformation, Institut für Analysis und Scientific Computing, 2015. TU Wien Online Katalog (AC12656133).

[253] R Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 404–413, New York, NY, USA, 1986. ACM. ISBN 0-89791-193-8. doi: 10.1145/12130.12172.

[254] Raimund Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1981.

[255] Michael Seitz, Gerta Köster, and Alexander Pfaffinger. Pedestrian group behavior in a cellular automaton. In *Pedestrian and Evacuation Dynamics 2012*, pages 807–814. Springer International Publishing, 2014. doi: 10.1007/978-3-319-02447-9-67.

[256] Michael J. Seitz. *Simulating pedestrian dynamics: Towards natural locomotion and psychological decision making*. Dissertation, Technische Universität München, Munich, Germany, 2016.

[257] Michael J. Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86(4):046108, 2012. doi: 10.1103/PhysRevE.86.046108.

[258] Michael J. Seitz and Gerta Köster. How update schemes influence crowd simulations. *Journal of Statistical Mechanics: Theory and Experiment*, 2014(7):P07002, 2014. doi: 10.1088/1742-5468/2014/07/P07002.

[259] Michael J. Seitz, Felix Dietrich, and Gerta Köster. A study of pedestrian stepping behaviour for crowd simulation. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 282–290, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.054.

[260] Michael J. Seitz, Felix Dietrich, and Gerta Köster. The effect of stepping on pedestrian trajectories. *Physica A: Statistical Mechanics and its Applications*, 421:594–604, 2015. doi: 10.1016/j.physa.2014.11.064.

[261] Michael J. Seitz, Nikolai W. F. Bode, and Gerta Köster. How cognitive heuristics can explain social interactions in spatial movement. *Journal of the Royal Society Interface*, 13(121):20160439, 2016. doi: 10.1098/rsif.2016.0439.

[262] Michael J. Seitz, Anne Templeton, John Drury, Gerta Köster, and Andrew Philippides. Parsimony versus reductionism: how can crowd psychology be introduced into computer simulation? *Review of General Psychology*, 21(1):95–102, 2016.

[263] J. A. Sethian and A. Vladimirsky. Fast methods for the eikonal and related Hamilton-Jacobi equations on unstructured meshes. *Proceedings of the National Academy of Sciences*, 97(11):5699–5703, 2000. doi: 10.1073/pnas.090060097.

[264] M. I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162, Oct 1975. doi: 10.1109/SFCS.1975.8.

[265] Wei Shao and Demetri Terzopoulos. Autonomous pedestrians. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 19–28, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931988. doi: 10.1145/1073368.1073371.

[266] Jonathan Richard Shewchuk. Triangle: engineering a 2d quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70680-9.

[267] Jonathan Richard Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, SCG '98, pages 86–95, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919734. doi: 10.1145/276884.276894.

[268] Jonathan Richard Shewchuk. Sweep algorithms for constructing higher-dimensional constrained Delaunay triangulations. In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry*, SCG '00, pages 350–359, New York, NY, USA, 2000. ACM. ISBN 1-58113-224-7. doi: 10.1145/336154.336222.

[269] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1):21 – 74, 2002. ISSN 0925-7721. doi: 10.1016/S0925-7721(01)00047-5. 16th ACM Symposium on Computational Geometry.

[270] Meng Shi, Eric Wai Ming Lee, and Yi Ma. A newly developed mesoscopic model on simulating pedestrian flow. *Procedia Engineering*, 211: 614–620, 2018. ISSN 1877-7058. doi: 10.1016/j.proeng.2017.12.055. 2017 8th International Conference on Fire Science and Fire Protection Engineering (ICFSFPE 2017).

[271] Z. Shiller, F. Large, and S. Sekhavat. Motion planning in dynamic environments: obstacles moving along arbitrary trajectories. In *IEEE International Conference on Robotics and Automation, 2001*, volume 4, pages 3716–3721, 2001. ISBN http://id.crossref.org/isbn/0-7803-6576-3. doi: 10.1109/robot.2001.933196.

[272] A. Shrestha and I. Senocak. Multi-level domain-decomposition strategy for solving the eikonal equation with the fast-sweeping method. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2297–2303, Oct 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2018.2829869.

[273] A. Sieben, J. Schumann, and A. Seyfried. Collective phenomena in crowds-where pedestrian dynamics need social psychology. *PLoS ONE*, 12(6), 2017. doi: 10.1371/journal.pone.017732.

[274] Harmeet Singh, Robyn Arter, Louise Dodd, Paul Langston, Edward Lester, and John Drury. Modelling subgroup behaviour in crowd dynamics DEM simulation. *Applied Mathematical Modelling*, 33(12):4408–4423, 2009. doi: 10.1016/j.apm.2009.03.020.

[275] Tamal Krishna Dey Siu-Wing Cheng and Jonathan Richard Shewchuk. *Delaunay Mesh Generation*. Taylor & Francis, 2013.

[276] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693, 9781848000698.

[277] Sven Skyum. A sweepline algorithm for generalized Delaunay triangulations and a simple method for nearest-neighbour search. In *Proceedings of the Workshop on Computational Geometry*. Universiteit Utrecht, 1992.

[278] S. W. Sloan. A fast algorithm for generating constrained Delaunay triangulations. *Computers & Structures*, 47(3):441–450, 1993. ISSN 0045-7949. doi: 10.1016/0045-7949(93)90239-A.

[279] Anthony Steed and Roula Abou-Haidar. Partitioning crowded virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '03, pages 7–14. ACM, 2003. doi: 10.1145/1008653.1008658.

[280] G. Sutmann and V. Stegailov. Optimization of neighbor list techniques in liquid matter simulations. *Journal of Molecular Liquids*, 125(2): 197–203, 2006. ISSN 0167-7322. doi: 10.1016/j.molliq.2005.11.029. Complex Liquids.

[281] Till Tantau. Graph drawing in TikZ. In *Proceedings of the 20th International Conference on Graph Drawing*, GD'12, pages 517–528, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36762-5. doi: 10.1007/978-3-642-36763-2_46.

[282] Franco Tecchia and Yiorgos Chrysanthou. Real-time rendering of densely populated urban environments. In Bernard Péroche and Holly Rushmeier, editors, *Rendering Techniques 2000*, pages 83–88, Vienna, 2000. Springer Vienna. ISBN 978-3-7091-6303-0.

[283] Kardi Teknomo and Gloria Gerilla. Mesoscopic multi-agent pedestrian simulation. *Transportation Research Trends*, pages 323–336, 01 2008.

[284] Anne Templeton and Neville Fergus. Modelling collective behaviour: insights and applications from crowd psychology. In *Crowd Dynamics, Volume 2: Theory, Models, and Applications*. Springer International Publishing, 2020.

[285] Anne Templeton, John Drury, and Andrew Philippides. From mindless masses to small groups: Conceptualizing collective behavior in crowd modeling. *Review of General Psychology*, 19(3):215–229, 2015. doi: 10.1037/gpr0000032.

[286] Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherill. *Handbook of Grid Generation*. CRC-Press, 1 edition, 1998. doi: 10.1201/9781420050349.

[287] Edward C Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948.

[288] Antoine Tordeux and Armin Seyfried. Collision-free nonuniform dynamics within continuous optimal velocity models. *Physical Review E*, 90:042812, 2014. doi: 10.1103/PhysRevE.90.042812.

[289] Antoine Tordeux, Mohcine Chraibi, and Armin Seyfried. Collision-free first order model for pedestrian dynamics. In *Traffic and Granular Flow '15*, Nootdorp, the Netherlands, 2015. 27–30 October 2015.

[290] Antoine Tordeux, Gregor Lämmel, Flurin S. Hänseler, and Bernhard Steffen. A mesoscopic model for large-scale simulation of pedestrian dynamics. *Transportation Research Part C: Emerging Technologies*, 93:128–147, 2018. ISSN 0968-090X. doi: eknomoj.trc.2018.05.021.

[291] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Transactions on Graphics (SIGGRAPH 2006)*, 25(3):1160–1168, 2006. doi: 10.1145/1141911.1142008.

[292] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995.

[293] Monika Twarogowska, Paola Goatin, and Regis Duvigneau. Comparative study of macroscopic pedestrian models. *Transportation Research Procedia*, 2(Supplement C):477 – 485, 2014. doi: 10.1016/j.trpro.2014.09.063. The Conference on Pedestrian and Evacuation Dynamics 2014 (PED 2014), 22-24 October 2014, Delft, The Netherlands.

[294] Vadere team. Vadere: open source framework for pedestrian simulation, 2020. URL http://www.vadere.org/. Accessed 26. May 2020.

[295] J. van den Berg, Ming Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *IEEE International Conference on Robotics and Automation, 2008 (ICRA 2008)*, pages 1928–1935, 2008. doi: 10.1109/ROBOT.2008.4543489.

[296] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[297] Isabella von Sivers. Numerische Methoden zur Optimierung der Schrittrichtung und -weite in einem Modell der Personenstromsimulation. Master's thesis, Fernuniversität in Hagen, 2013.

[298] Isabella von Sivers and Gerta Köster. Realistic stride length adaptation in the optimal steps model. In *Traffic and Granular Flow '13*, Jülich, Germany, 2013.

[299] Isabella von Sivers and Gerta Köster. How stride adaptation in pedestrian models improves navigation. *arXiv*, 1401.7838(v1), 2014.

[300] Isabella von Sivers and Gerta Köster. Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74:104–117, 2015. doi: 10.1016/j.trb.2015.01.009.

[301] Isabella von Sivers and Gerta Köster. Realistic stride length adaptation in the optimal steps model. In *Traffic and Granular Flow '13*, pages 171–178. Springer, 2015. ISBN 978-3-319-10629-8. doi: 10.1007/978-3-319-10629-8_20.

[302] Isabella von Sivers, Anne Templeton, Gerta Köster, John Drury, and Andrew Philippides. Humans do not always act selfishly: social identity and helping in emergency evacuation simulation. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 585–593, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.099.

[303] Isabella von Sivers, Florian Künzner, and Gerta Köster. Pedestrian evacuation simulation with separated families. In *Proceedings of the 8th International Conference on Pedestrian and Evacuation Dynamics (PED2016)*, Hefei, China, October 2016.

[304] Isabella von Sivers, Michael Jakob Seitz, and Gerta Köster. How do people search: a modelling perspective. In *Parallel Processing and Applied Mathematics, 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part II*, volume 9574 of *Lecture Notes in Computer Science*, pages 487–496. Springer, 2016. doi: 10.1007/978-3-319-32152-3_45.

[305] Isabella von Sivers, Anne Templeton, Florian Künzner, Gerta Köster, John Drury, Andrew Philippides, Tobias Neckel, and Hans-Joachim Bungartz. Modelling social identification and helping in evacuation simulation. *Safety Science*, 89:288–300, 2016. ISSN 0925-7535. doi: 10.1016/j.ssci.2016.07.001.

[306] Jiayue Wang, Wenguo Weng, Maik Boltes, Jun Zhang, Antoine Tordeux, and Verena Ziemer. Step styles of pedestrians at different densities. *Journal of Statistical Mechanics: Theory and Experiment*, 2018(2):023406, 2018.

[307] J. Was, B. Gudowski, and P.J. Matuszyk. Social Distances Model of pedestrian dynamics. In *Cellular Automata*, volume 4173 of *Lecture Notes in Computer Science*, pages 492–501. Springer Berlin Heidelberg, 2006. doi: 10.1007/11861201_57.

[308] Jarosław Was and Robert Lubaś. Adapting Social Distances Model for mass evacuation simulation. *Journal of Cellular Automata*, 8:395–405, 2013. Journal of Cellular Automata, Old City Publishing.

[309] Jarosław Was, Hubert Mroz, and Pawel Topa. GPGPU computing for microscopic simulations of crowd dynamics. *COMPUTING AND INFORMATICS*, 2015.

[310] Michael Waskom and the seaborn development team. mwaskom/seaborn, September 2020.

[311] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2): 167–172, 01 1981. ISSN 0010-4620. doi: 10.1093/comjnl/24.2.167.

[312] Ulrich Weidmann. *Transporttechnik der Fussgänger*, volume 90 of *Schriftenreihe des IVT*. Institut für Verkehrsplanung, Transporttechnik, Strassen- und Eisenbahnbau (IVT) ETH, Zürich, 2nd edition, 1992. doi: 10.3929/ethz-a-000687810.

[313] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611 – 615, 2011. doi: 10.1016/j.cpc.2010.11.002.

[314] Wes McKinney. Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[315] G. B. Whitham. Exact solutions for a discrete system arising in traffic flow. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 428(1874):49–69, 1990. doi: 10.1098/rspa.1990.0025.

[316] Nanda Wijermans. *Understanding Crowd Behaviour: Simulating Situated Individuals*. PhD thesis, Rijksuniversiteit Groningen, 2011.

[317] Stephen Wolfram. Statistical mechanics of cellular automata. *Review of Modern Physics*, 55:601–644, 1983. doi: 10.1103/RevModPhys.55.601.

[318] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 311:419–424, 1984. doi: 10.1038/311419a0.

[319] D. Wolinski, S. J. Guy, A.-H. Olivier, M. Lin, D. Manocha, and J. Pettré. Parameter estimation and comparative evaluation of crowd simulations. *Comput. Graph. Forum*, 33(2):303–312, May 2014. ISSN 0167-7055. doi: 10.1111/cgf.12328.

[320] Y. Xiao, M. Chraibi, Y. Qu, A. Tordeux, and Z. Gao. Investigation of Voronoi diagram based direction choices using uni- and bi-directional trajectory data. *Physical Review E*, 97(5), 2018. doi: 10.1103/PhysRevE.97.052127.

[321] K. Yamaguchi, A. C. Berg, L. E. Ortiz, and T. L. Berg. Who are you with and where are you going? In *CVPR 2011*, pages 1345–1352, 2011.

[322] J. Yang and F. Stern. A highly scalable massively parallel fast marching method for the eikonal equation. *ArXiv e-prints*, feb 2015.

[323] L.Z. Yang, D.L. Zhao, J. Li, and T.Y. Fang. Simulation of the kin behavior in building occupant evacuation based on cellular automaton. *Building and Environment*, 40(3):411–415, 2005. doi: 10.1016/j.buildenv.2004.08.005.

[324] Liron Yatziv, Alberto Bartesaghi, and Guillermo Sapiro. O(n) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212:393–399, 03 2006. doi: 10.1016/j.jcp.2005.08.005.

[325] Erdal Yilmaz, Veysi Isler, and Yasemin Yardimci Çetin. The virtual marathon: parallel computing supports crowd simulations. *IEEE Comput. Graph. Appl.*, 29(4):26–33, July 2009. ISSN 0272-1716. doi: 10.1109/MCG.2009.77.

[326] W. J. Yu, R. Chen, L. Y. Dong, and S. Q. Dai. Centrifugal Force Model for pedestrian dynamics. *Physical Review E*, 72:026112, 2005. doi: 10.1103/PhysRevE.72.026112.

[327] G. Zeng, S. Cao, C. Liu, and W. Song. Experimental and modeling study on relation of pedestrian step length and frequency under different headways. *Physica A: Statistical Mechanics and its Applications*, 500:237–248, 2018. doi: 10.1016/j.physa.2018.02.095.

[328] J. Zhang, Wolfram Klingsch, Andreas Schadschneider, and Armin Seyfried. Transitions in pedestrian fundamental diagrams of straight corridors and T-junctions. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(06):P06004, 2011. doi: 10.1088/1742-5468/2011/06/P06004.

[329] J. Zhang, A. Schadschneider, and A. Seyfried. Empirical fundamental diagrams for bidirectional pedestrian streams in a corridor. In *Pedestrian and Evacuation Dynamics 2012*, pages 245–250. Springer International Publishing, 2014. doi: 10.1007/978-3-319-02447-9_19.

[330] H. Zhao. Parallel implementations of the Fast Sweeping Method. *Journal of Computational Mathematics*, 25:421–429, July 2007.

[331] Hongkai Zhao. A fast sweeping method for eikonal equations. *Math. Comput.*, 74(250):603–627, 2005. doi: 10.1090/S0025-5718-04-01678-3.

[332] M. Zhao, S. J. Turner, and W. Cai. A data-driven crowd simulation model based on clustering and classification. In *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 125–134, 2013.

[333] Jinghui Zhong, Nan Hu, Wentong Cai, Michael Lees, and Linbo Luo. Density-based evolutionary framework for crowd model calibration. *Journal of Computational Science*, 6:11–22, 2015. doi: 10.1016/j.jocs.2014.09.002.

[334] Jinghui Zhong, Wentong Cai, Linbo Luo, and Mingbi Zhao. Learning behavior patterns from video for agent-based crowd modeling and simulation. *Autonomous Agents and Multi-Agent Systems*, 30:990–1019, 2016.

[335] B. Zhou and S. Zhou. Parallel simulation of group behaviors. In *Simulation Conference, 2004. Proceedings of the 2004 Winter*, volume 1, page 370, Dec 2004. doi: 10.1109/WSC.2004.1371337.

[336] Benedikt Zönnchen and Gerta Köster. A parallel generator for sparse unstructured meshes to solve the eikonal equation. *Journal of Computational Science*, 32:141–147, 2018. ISSN 1877-7503. doi: 10.1016/j.jocs.2018.09.009.

[337] Benedikt Zönnchen and Gerta Köster. GPGPU computing for microscopic pedestrian simulation. In *Parallel Computing: Technology Trends*, volume 36, pages 93–104, 2020. doi: 10.3233/APC200029.

[338] Benedikt Zönnchen, Matthias Laubinger, and Gerta Köster. Towards faster navigation algorithms on floor fields. In *In Traffic and Granular Flow '17*, pages 307–315, Cham, 2019. Springer International Publishing. ISBN 978-3-030-11440-4. doi: 10.1007/978-3-030-11440-4_34.

[339] Benedikt Zönnchen, Benedikt Kleinmeier, and Gerta Köster. Vadere – a simulation framework to compare locomotion models. In *Traffic and Granular Flow 2019*, pages 331–337, Cham, 2020. Springer International Publishing. doi: 10.1007/978-3-030-55973-1_41.