TOWARDS A STATISTICAL PHYSICS OF COLLECTIVE MOBILITY AND DEMAND-DRIVEN TRANSPORT

Dissertation
zur Erlangung des mathematisch–naturwissenschaftlichen Doktorgrades
“Doctor rerum naturalium”
der Georg-August-Universität Göttingen

im Promotionsprogramm ProPhys
der Georg-August University School of Science (GAUSS)

vorgelegt von
ANDREAS SORGE
aus Uelzen

Göttingen, 2017

BETREUUNGSAUSSCHUSS

Prof. Dr. Marc Timme
Netzwerkdynamik, MPI für Dynamik & Selbstorganisation
Institut für Nichtlineare Dynamik, Universität Göttingen

Prof. Dr. Thomas Pruschke †

Prof. Dr. Ulrich Parlitz
Biomedial Physics, MPI für Dynamik & Selbstorganisation
Institut für Nichtlineare Dynamik, Universität Göttingen

Prof. Dr. Florentin Wörgötter
Department of Computational Neuroscience
Drittes Physikalisches Institut – Biophysik, Universität Göttingen

MITGLIEDER DER PRÜFUNGSKOMMISSION

Referent: Prof. Dr. Marc Timme
Network Dynamics, MPI für Dynamik & Selbstorganisation

Korreferent: Prof. Dr. Florentin Wörgötter
Drittes Physikalisches Institut – Biophysik, Universität Göttingen

WEITERE MITGLIEDER DER PRÜFUNGSKOMMISSION

Prof. Dr. Theo Geisel
Nichtlineare Dynamik, MPI für Dynamik & Selbstorganisation

Prof. Dr. Stephan Herminghaus
Abteilung Dynamik komplexer Fluide
MPI für Dynamik & Selbstorganisation

Prof. Dr. Reiner Kree
Institut für Theoretische Physik, Universität Göttingen

Prof. Dr. Ulrich Parlitz
Biomedial Physics, MPI für Dynamik & Selbstorganisation

Tag der mündlichen Prüfung: 19. Juni 2017

ACKNOWLEDGMENTS

Bootstrapping an endeavour to model, simulate and study collective mobility and demand-driven transport systems in terms of statistical physics and complex systems, devising an analytical and computational framework to do so, acquiring the necessary cross-disciplinary expertise in complex systems, critical transitions, stochastic processes, queueing theory, discrete-event simulation, high-performance computing, modern scientifc software development, while interfacing with people outside the ivory tower who will be actually funding and operating these systems, it takes time, and time has taken its toll. Here is one of the numerous opportunities I thank my family and friends for their enduring support—first and foremost Caro ♡, as well as Niklas who joined us recently.

I thank Marc Timme and Stephan Herminghaus for fruitful scientific advice and non-scientific discussions, and for entrusting me with opportunity, time and the resources to work out conceptual and computational foundations in this field, as well as for founding the EcoBus project and team together. As the Lower Saxonian Minister for Science also noted when she recently handed over the EcoBus grant notification to us, I am grateful that Stephan and Marc strive for EcoBus not only to become a scientific success, but also to lay the ground for societal innovation and to help master the great transformation of mobility and transport outside Fassberg and Bunsenstraße, which drives all of us. Crucial to this endeavour is Carolin Hoffrogge who has been facilitating the public and political leverage necessary to kickstart our EcoBus grant application and implement collective mobility cooperatively in the region to actually strengthen public transport. In the meantime, others have been reinforcing our initiating efforts, such as Jan Schlüter, heading the Next Generation Mobility Group, Dieter Treichel and Bernd Ctortecka at Max Planck Innovation supporting us filing a patent application and running a scientific project outside the realm of science; Levke Johanna Deutsch who relentlessly masters grant management and funding agencies, and Barbara Kutz who admirably succeeds in keeping the bunch of scientists that we are sufficiently organized on a daily basis.

The late Thomas Pruschke was not only a member of my thesis committee, but was one of my teachers and mentors in Göttingen who inspired me from day one. It is a pleasure and privilege to have studied and worked with him. Only the good die young.

Ulrich Parlitz and his NLD course ignited my passion for complex systems and emergent phenomena, which Theo Geisel and Marc Timme later could tap into and nurture when they gave me the opportunity to study collective phenomena of neural networks and power grids. They have been cultivating an excellent research environment at MPI for Dynamics and Self-Organization, the Bernstein Center for Computational Neuroscience, and in particular, Marc's Network Dynamics Group. It has been a good time with a lot of good people from all over the world, sharing and co-shaping the drive to understand emergence in networked systems to address societal needs and to work for a sustainable future.

Furthermore and foremost, thanks go to my main collaborators in temporal percolation and demand-driven transport systems, Jan Nagler, Nora Molkenthin and Debsankha Manik. Debsankha in particular has been following suit to pursue the subject of this Thesis and help me shape and focus our analytical and computational approach, together with Philip Marszal and Niels Beyer. After I internally "published" my prototype along with its formal specification, Philip and Debsankha have been substantially extending the Python framework for modelling and simulating demand-driven transport systems. Thanks go to my former students Johannes Kassel, Benjamin Walter, Matthias Wendland, for continual stimulating discussions and scientific support in shaping the approach of this Thesis. Among many others, I further acknowledge Bernhard Altaner, Hinrich Arnoldt, Vitaly Belik, Christian Bick, Stephan Eule, Frederik Fix, Harold Gutch, Sarah Hallerberg, Sven Jahnke, Jani-Pekka Jokinen, Christoph Kirst, Felix Krahmer, Moritz Matthiae, Dimitra Maoutsa, Christiane Menzfeld, Michael Patscheke, Kari Rissanen, Henrik Ronellenfitsch, Marie Schmidt, Anita Schöbel, Malte Schröder, Stephan Westphal, Annette Witt, Xiaozhu Zhang and Christian Zigenhorn in doing so. Further invaluable support has been provided by Yorck-Fabian Beensen, Ayse Bolik, Ulrich Degenhardt, Hecke Degering, Irina Fischer, Denny Fliegner, Tanja Gindele, Michael Günther, Achim Hennies, Thomas Köhler, Tobias Niemann, Vika Nowak, Verena Rosenthal, Katharina Schneider, Monika Teuteberg, Regina Wunderlich, to name but a few.

Modern computational research depends on an increasingly larger ecosystem of scientific computing tools, mainly initiated by domain scientists to be used by their peers. I thank my fellow pycnic compies Florencia Noriega, Jana Lasser, Frank Stollmeier and Debsankha Manik in supporting each other in our joint commitment to professional scientific software development and computational science education. I extend these thanks to the developers of the free scientific software tools, whose shoulders this Thesis stands upon.

I acknowledge enduring partial funding by the taxpayers of Germany through the Max Planck Society, in particular the Departments of Dynamics of Complex Fluids and Nonlinear Dynamics, the Network Dynamics Group, and through a scholarship of the IMPRS Physics of Biological and Complex Systems. I acknowledge my current employer, the German Council of Science and Humanities (Wissenschaftsrat), for giving me the opportunity to continue working in this field.

ABSTRACT

Collective mobility and demand-driven transport systems are vital to proper, efficient and sustainable functioning of biological, technical and social systems. They are relevant to mastering several major transitions human society is facing today on a global scale, and they have been attracting considerable interest as on-demand ride-sharing systems are projected to disrupt the individual mobility and public transport sector. In collective mobility systems and demand-driven transport systems alike, vehicles or other discrete mobile units carry individual passengers, goods or other discrete immobile loads. These systems do so upon individual request for transport from individual origins to individual destinations, within individual time windows. Coordination functions in these systems include assigning requests to transporters and routing the transporters within the underlying geometry. When transporters carry multiple loads at the same time, another function of the system is to bundle spatiotemporally overlapping requests. Given both the need and the recent interest and implementation of collective mobility and demand-driven transport systems, it is imperative to understand their core structural and dynamical properties and how they relate to their satisfactory and efficient functioning.

Modelling and simulating such discrete-event systems involves untypical technicalities that presumably have hindered progress in studying these systems from the network dynamics and statistical physics perspective so far. In order to unlock collective mobility and demand-driven transport systems for studies in these fields, I devise a modular framework to model and simulate such systems.

Furthermore, a fundamental steady-state performance measure is the transport capacity of the system. If overall demand exceeds capacity, the system congests and ceases to function. Determining the capacity is henceforth crucial to inform system design for optimized system efficiency and individual service quality. Intriguingly, the brink to congestion constitutes a critical transition reminiscent of percolation in time. I develop a dynamic notion of criticality of such stochastic processes, mapping the transition from stability to instabilty to a hybrid percolation phase transition.

Overall, I anticipate this Thesis and the tools developed to be a starting point for modelling and studying the dynamics of collective mobility and demand-driven transport systems, and for understanding how the intricate interplay of their structure and their dynamics governs their functioning.

SCIENTIFIC SOFTWARE

In the course of this Thesis, I developed the following scientific software:

[1] A. Sorge, *pyd3t – A Scientific Python package implementing the Demand-Driven Directed Transport (D3T) Specification*, version 0.2, 2015.

[2] A. Sorge, *pydevs 0.1.8 – a Python wrapper of adevs*, version 0.1.8, 2015.

[3] A. Sorge, *pytemper – Scientific Python package for finite-time analysis of the recurrence-transience transition in the temporal percolation paradigm*, version 0.3.2, 2017.

[4] A. Sorge, "Pyfssa 0.7.6 – Scientific Python Package for finite-size scaling analysis," Zenodo (2015) `10.5281/zenodo.35293`.

[5] A. Sorge, "Pypercolate 0.4.6 – Scientific Python package for Monte-Carlo simulation of percolation on graphs," Zenodo (2015) `10.5281/zenodo.35305`.

PUBLICATIONS

In the course of this Thesis, I co-authored the following peer-reviewed publications, of which publication [1] has been incorporated into this Thesis (cf. Chapter 6):

[1] A. Sorge, D. Manik, S. Herminghaus, and M. Timme, "Towards a unifying framework for demand-driven directed transport (D3T)," in Proceedings of the 2015 Winter Simulation Conference (2015), pp. 2800–2811.

[2] D. Manik, D. Witthaut, B. Schäfer, M. Matthiae, A. Sorge, M. Rohden, E. Katifori, and M. Timme, "Supply networks: Instabilities without overload," The European Physical Journal Special Topics **223**, 2527 (2014).

[3] M. Rohden, A. Sorge, D. Witthaut, and M. Timme, "Impact of network topology on synchrony of oscillatory power grids," Chaos: An Interdisciplinary Journal of Nonlinear Science **24**, 013123 (2014).

[4] M. Rohden, A. Sorge, M. Timme, and D. Witthaut, "Self-Organized Synchronization in Decentralized Power Grids," Physical Review Letters **109**, 064101 (2012).

[5] J. D. Thompson, P. A. McClarty, H. M. Rønnow, L. P. Regnault, A. Sorge, and M. J. P. Gingras, "Rods of Neutron Scattering Intensity in $Yb_2Ti_2O_7$: Compelling Evidence for Significant Anisotropic Exchange in a Magnetic Pyrochlore Oxide," Physical Review Letters **106**, 187202 (2011).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Part I

Introduction

1 INTRODUCTION

Bypasses are devices that allow some people to dash from point A to point B very fast while other people dash from point B to point A very fast. People living at point C, being a point directly in between, (...) often wish that people would just once and for all work out where the hell they wanted to be. (Douglas Adams)

1.1 THE CASE FOR COLLECTIVE MOBILITY & TRANSPORT

Collective mobility and demand-driven transport systems are vital to proper, efficient and sustainable functioning of biological, technical and social systems. Functional differentiation and specialization, the separation of tasks within a system and among systems, is a hallmark of biological, technical and social systems. In living organisms, vital tasks are separated among several organs, among differentiated cells, and within cells among several organelles, requiring transport mechanisms and distribution networks. [1, 2] Technical systems providing utilities such as water, electricity, information and communication, spatially separate production and consumption of the respective good, enabling virtualization at the consumer's end. Think of ancient aquaeducts and qanats, the steam engine, and as a recent example cloud computing and storage services. [3–7] Modern human society heavily relies on a global divison of labor to sustain population level and ubiquitous access to a plethora of goods, as well as to fulfill higher-order functions. [8–11] Functions humans demand in daily life are highly differentiated and henceforth spatially separated in modern societies and include education, work, leisure, shopping, sports, health. [12, 13]

Self-evidently, spatial separation of tasks entails the need for a transport mechanism to move and exchange objects between the locations of the tasks. For example, the vertebrate brain serves as a central sensory processing and motor control unit. It relies on the transport of signals (action potentials) through neurons extending into the rest of the body, to receive sensory input and exert motor control. The vertebrate cardiovascular system transports carbon dioxide and oxygen between the lungs and the rest of the body, and exchanges nutrients between the cells. Within cells, motor proteins actively transport vesicles loaded with biomolecules towards their destination. [14–16] Transcontinental power grids transport electrical energy provided by power plants to customers all across the continent. A global logistics network ensures the timely transport of raw materials and goods to their customers. [17] Last but not least, aviation networks, railway networks, road networks, and local public transport such as provided by omnibusses or taxis, satisfy human travel and mobility demand. [18–23]

Collective mobility and demand-driven transport systems are relevant to mastering not one but several major transitions human society is facing today on a global scale. In the outgoing fossil and analog age, private cars and central planning have been dominating mobility and transport modes. Several drivers of change underly the recent interest and trend towards collective mobility and demand-driven transport systems. *Globalization* drives complexity of supply chains, *urbanization* drives more efficient use of scarce urban space, *decarbonization* drives more efficient use of scarce ressources, *planetary boundary conditions* and the overall great transformation call for a sustainable way of human civilization on Earth. [24–31] Furthermore, *Industry 4.0* and the *Digital Revolution* enable and require self-organized mobility and logistics systems, and *demographic change* requires new solutions to mobility and transport in increasingly ageing and depopulating rural areas. [32]

Collective mobility and demand-driven transport systems for human travel and the movement of goods and other objects are on the rise. Of particular interest are on-demand ride-sharing services. [33–35] Their promise is to provide reliable and affordable door-to-door mobility service and to reduce emissions, congestion and space consumption especially in urban centers, and to offer flexible mobility to rural areas. While taxi services, prescheduled dial-a-ride and minibus services have been around for a long time, the prevalence of mobile devices and cellular communication now render spontaneous coordination and large-scale on-demand ride-sharing services possible. Mobility on demand services developed by companies like Volkswagen Moia or ride-sharing services piloted by Helsinki Region Transport (Kutsuplus) have the potential to provide satisfactory individual transport and to efficiently use the available ressources at the same time. [36]

1.2 STUDYING MOBILITY AND TRANSPORT SYSTEMS

Collective mobility and demand-driven transport systems feature vehicles or other discrete transporting units that carry individual passengers, goods or other discrete immobile loads from individual origins to individual destinations, at individual times. As a physicist, I model transportation in such systems as having the following properties:

- Transportation is *demand-driven*: there are no external fields such as gravity or an electric field driving transport as in other systems studied in physics (e. g. charge transport in solids). (Also, there are no fixed schedules.)
- Transportation is *on-demand* or "urgent" in the sense that typical requests are placed and require to be served within a time window which is of the same order of magnitude as the travel time.
- Transportation is *directed*, as opposed to diffusive transport, or e. g. the unspecific distribution of nutrients by the cardiovascular system. (Also, there are no changes of transporters or modes.)
- Transported objects are *discrete* loads: there is no flow, there is no continuous quantity such as water.

- The transported loads are *immotile*: they do not move on their own, as opposed to conduction electrons in a metal.
- Loads are transported by *discrete transporter units*: there is no conveyer belt, or pipes, such as the Internet distributes data packets on a continuous basis.
- Transporters are costly: their number is of the order of the average number of requests to be served within the average time it takes to serve a request. In particular, not every load has its own transporter.

Mobility on demand and ride-sharing services motivate this focus on *demand-driven directed transport* (D3T) models. Nevertheless, D3T models might be relevant to a wider range of biological, technical and social transport systems, for example in intracellular transport.

Now, while there is an extensive and growing body of literature on various aspects and various models and instances of such systems, what is lacking so far is an integrated treatment that addresses how the core structural and dynamical properties of such systems relate to their satisfactory and efficient functioning. Of particular interest is whether and how decentralization and self-organization lead to more performant, resilient and sustainable mobility and transport. From a physicist's perspective, D3T systems are dynamical systems embedded in a certain geometry and driven by the requests for transport. These requests arrive according to a spatiotemporal stochastic process. Further parameters are the number of transporters, their capacity, and the algorithm to assign requests to transporters. We are interested in how all these components of the system shape the transport dynamics (the processing of the requests), and how the dynamics influences the performance (output) of these systems. Performance measures include measures of individual service quality, such as the waiting time distribution, and measures of system efficiency, such as the number of transporters needed to serve a given number of requests per time.

We aim at identifying universal principles and behavior that provide insight and offer a unified explanation for a range of mobility and transport systems and their parameters. Understanding these principles and relationships informs system design, improves algorithms to assign and route transporters to individual requests, and helps to prevent system breakdowns and to mitigate service disruptions. A plethora of disciplines is concerned with studying and designing mobility and transport systems, including regional and urban planning, traffic and logistics engineering, operations research, computer science, mathematics and physics. They aim to plan and improve mobility and transport in facilities, cities, municipalities, and, potentially, countries and across the globe.

Mathematicians and computer scientists excel at addressing the problems of assigning requests to transporters and of routing the transporters on a graph within the fields of combinatorics, optimization and graph theory. [37] These search and optimization problems are typically NP-hard. That is, finding an exact and optimal solution for a typical instance requires a number of steps that grows exponentially with system size. Yet, frequently there are search heuristics for finding some suboptimal but good enough solutions in polynomial time and also exact bounds on their optimality. The task is to find these efficient algorithms and adapt them to the computational problem at hand. These computational problems are either *static* (*offline problems*), with all requests known beforehand, or *dynamic* (*online problems*), with the requests becoming known one at a time. These dynamic problems are integral to online decision-making in D3T systems. In D3T systems, an online algorithm assigns requests to transporters, and it is crucial to determine the performance of the assignment and routing algorithms. However, this is not the end of the story, as the performance of an algorithm in isolation does not relate the decisions by that algorithm to the transport dynamics and system performance—for example, we still need to understand how certain algorithms are more prone to congestion than others. Furthermore, we need to compare the dynamic properties and performance of different systems across different parameters.

Pure and applied probability theoreticians describe and study the stochastic spatiotemporal demand process driving transport in D3T systems. They also relate this input pattern to the individual service quality and the system efficiency within the framework of queueing theory. [38] Notions of system performance such as the average waiting time from queueing theory are also appropriate for D3T studies. One caveat is that basic queueing theory is without a notion of moving in a physical space. That is, basic queueing theory only deals with temporal stochastic processes and their fluctuations, and how they affect performance of systems in which the servers do not need to move as transporters in a D3T system. While queueing theory is the science of congestion, excelling at thoroughly describing queueing systems under heavy load, it lacks a description of the transition from a stable queue to an unstable queue in terms of statistical physics.

Engineers in Transportation Planning study specific instances of transport systems and model them to great detail. In the absence of any prospect of analytically tracting or even solving such models, they resort to full-scale simulations of human traffic and daily activity patterns. [13, 39] Their methodology yields immediate results for pressing real-world transportation problems and heuristics to apply to other such problems. What is generally missing is an abstraction and an insight into the underlying dynamical properties and their dependence on structural parameters. This resembles the situation in the field of modeling electrical distribution networks also known as power grids. [40] Electrical engineering employs detailed models with literally thousands of paramaters to model and monitor power grid operations and to mitigate failures. However, these models are intractable and as such, do not facilitate an understanding on how networks parameters relate to grid performance.

Complexity scientists aim at explaining emergent properties of complex adaptive systems in terms of simple microscopic rules, hence unifying a seemingly disparate range of systems in a universal law, typically a power-law scaling with system size or another scale-free distribution. Prominent examples include the metabolic theory of ecology, preferential attachment, and scaling of cities and most recently, scaling of urban ride sharing. [41–44] In the past decade, comprehensive spatiotemporal records of individual human trips (or proxies thereof) or transport movements have become available (e. g. [45]). This has been fueling an on-going effort to uncover and explain universal spatiotemporal patterns of human movements. [22, 23, 46–48] Yet, while large-scale spatiotemporal data are useful for spotting universality across various systems and for testing predictions of models, such analysis still comes short of providing insight into dynamical properties and their fluctuations in D3T systems other than providing useful heuristics.

1.3 PHYSICS OF COLLECTIVE MOBILITY & TRANSPORT

Physics of collective mobility and demand-driven transport systems is about general principles that unify the description of the dynamics of a large class of such systems (cf. [49]). These principles relate structure and dynamics and quantify how dynamical system properties scale over a range of magnitudes. They uncover when and how *per se* local interactions of the constituents of the system or small perturbations lead to global effects, the signatures of emergence and criticality. These include dynamical instabilities as well as structural bifurcations and phase transitions – and qualifying and quantifying their precursors. Physics is less about explicitly explaining a particular system in detail. Instead, it aims at models as simple as possible that feature the phenomenon under study and allow to deduce general properties. For critical transitions, universality is the prevalent notion in this endeavour, as a universality class unifies a range of disparate systems with different microscopic interactions but nevertheless same qualitative behavior at critical transitions involving the system as a whole. [50]

Statistical physics of D3T systems includes studying how the parameters and system components such as the underlying geometry and the assignment and routing of transporters affects the dynamics and system performance. For example, it is of immediate interest how dynamical performance quantities scale with parameters such as system size, transporter number, request rate. Regarding the dynamics itself, the general question is how much small parameter changes or spatiotemporally local fluctuations or perturbations affect the system as a whole. Of particular relevance to the reliable functioning of collective mobility and demand-driven transport systems are such small changes when they inflict non-linear responses, as seen in bifurcations, deterministic chaos and phase transitions.

Modelling and studying D3T systems is not straightforward to physicists. They are of high dimensionality, have a rich and rather technical state space and they are not smooth. In fact, with pick-ups and deliveries, their time evolution is governed by discrete events in continuous time, rather than continuous differential equations or low-dimensional mappings in discrete time. What is more, albeit not unseen, the irreducible notion of passive immotile loads transported by discrete transporter units is also rather unusal for models in the physics literature. Additionally, the combination of optimization and routing algorithms as well as time-discrete events in continuous time requires boilerplate abstraction in modelling and simulation, resulting in a steep learning and implementation curve with unclear scientific and personal reward. However, it is in line with the recent trend of physicists increasingly moving into domains of biological, socioeconomic and technical systems. They do so mainly within the methodology of network science and network dynamics as a paradigm for systems of many elements with pairwise interaction other than through physical fields.

1.4 ABOUT THIS THESIS

This Thesis along with the scientific software developed in its course is conceptual work. This work is to facilitate modelling and simulation of collective mobility and demand-driven transport systems, and to relate their instabilities to phase transitions. With the goal of conducting domain science, in studying the statistical physics and network effects of collective mobility and demand-driven transport systems, this Thesis is a fundament to conduct actual studies by providing a framework for rigorous modelling as well as defining the relevant quantities to the statistical physicists, both formally and computationally. In this approach, it complements existing approaches such as data-driven science [34, 44] or mean-field theory [51]. In fact, in providing a modelling and simulation toolkit to the statistical physicist and network scientist, my framework facilitates computational studies that bridge the gap between these approaches. That several of these studies are actually being conducted at the time of writing is a reassuring confirmation of this at times somewhat technical but nonetheless fundamental scientific approach. It would not have materialized without domain science expertise and without designing domain science studies.

The structure of this Thesis is as follows. Part II gives fundamental theoretical background on critical transitions, stochastic processes and discrete-event systems from standard works in these fields. Part III introduces the formal and computational framework I developed to model, simulate and analyze demand-driven directed transport (D3T) systems. This Part has been partially published in a peer-reviewed conference proceeding published by IEEE and partially presented at the 2017 DPG spring meeting. [52, 53] Part IV shines light on the critical transition of simple models of such systems—simple queues or random walks. The temporal percolation paradigm and computational framework I develop links statistical physics and instabilities of stochastic processes such as those modelling queues and collective mobility and demand-driven transport systems. This is so far unpublished work (other than an invited talk in the Physics of Collective Mobility symposium at the DPG spring meeting 2017, and a contributed talk at the DPG spring meeting 2015). [54, 55] Finally, I conclude in Part V.

This Thesis is scripted and typeset with PythonTeX, ArsClassica and ClassicThesis LaTeX packages (among others). [56–60]. This is a selection of the scientific Python packages this Thesis imports: NumPy [61], SciPy [62–64], Matplotlib [65], IPython [66], Jupyter [67, 68], h5py [69], seaborn [70], dask [71, 72] For a detailed account of the computational environment this document and its figures have been prepared with, see Listing C.1.

Part II

Theoretical Background

2 CRITICAL TRANSITIONS

2.1 INTRODUCTION

Systems with many degrees of freedom, or many constituents, exhibit collective behavior that is qualitatively different from the behavior of a single or only a few of its constituents. It is this notion of emergence that simple local interactions among the constituents generate new states of the system as a whole ("more is different"): At each scale, a new effective but no less fundamental theory is needed. [73] Statistical physics aims at a statistical description of such systems, as a full microscopic description of the system beyond its basic equations is out of reach and would anyway not capture the emergent properties at the system scale. [74, 75]

As such large systems feature emergent states which are more than a linear juxtaposition of its parts, they also feature non-linear collective responses such as abrupt changes in their qualitative behavior when subjected to a small perturbation or small change in some parameter. Phase transitions transform systems from one macroscopic state into another, involving all scales of the system; somewhat analogously, bifurcations and similar critical transitions abruptly change the long-term behavior of dynamical systems. [76–79] Such transitions exist only in infinite systems as systems involving only a finite number of finite quantities in analytical expressions such as the partition sum do not generate singularities: Phase transitions exist only in the *thermodynamic limit* of infinitely large systems, critical transitions in dynamical systems exist only in the *asymptotic regime* of infinite duration. [50, 79]

Typically, phase transitions are discontinuous (first order) or continuous (second order). In first-order transitions, macroscopic regions of different microscopic properties coexist at the critical point (such as water ice and liquid water at the melting point). In particular, detail of local interaction is relevant to the critical behaviour of the system. In contrast, continuous transitions let the two phases coincide at the critical point. Instead, the correlation length of fluctuations in the quantity of interest (the "order parameter") distinguishing the two phases diverges. This divergence in turn means that detail of local (short-range) interaction becomes irrelevant at the critical point, giving rise to universal behavior. On the contrary, for first-order transitions, no such universality exist as different microscopic configurations coexist at the critical point. [75, 79]

While the notion of a "critical transition" of dynamical systems remains elusive in the wider literature [78], it is susceptible to mathematical rigor from dynamical systems and bifurcation theory [80, 81]. Both continuous phase transitions as well as critical transitions in the sense of Scheffer [78] and Kuehn [81] — henceforth all referred to as critical transitions — feature critical slowing down and increasing fluctuations in the vicinity of the transition. [79, 82]. These are precursors, also called early-warning signs, of imminent critical transitions that help to predict and eventually mitigate or prevent such a transition. [82–84].

Continuous phase transitions are characterized by an infinite susceptibility and correlation length, as the phases on both sides of the transition coincide at the critical point. Criticality refers to exactly this state of the system at the transition, when the system is gripped by usually localized fluctuations now collectively scaling up to become as large as the system itself. Instead of coexisting microscopic phases in first-order transitions, at a critical transition there is no microscopic any more: as a critical transition involves all scales of the system, it transcends any meaningful separation of scales. [79]

Criticality is in an intricate sense a collective effect of collective effects at all scales of a system. Usually, for each scale of collective behaviour, an effective theory describes the emergent behaviour on that scale and otherwise averages the behavior on smaller and larger scales as there is a clear hierarchical separation in space and time. As it is dominated by fluctuations, criticality defies the very notion of averaging. As perturbations on different scales strongly couple up to system size, there is no separation of scales any more; the collective behavior and its correlations become *self-similar*, the system becomes *scale-free* or *scale invariant*. [79]

The signature of quantified scale invariance of the system is its relevant properties obeying power laws. Indeed, in the vicinity of a critical transition, correlations decay slowly according to a power law. Such scaling laws allow to infer the behavior of a system or the value of a property of a system at any given scale from a known system at a particular scale. The exponents of the relevant power laws describe the divergence of these properties near the critical point. These *critical exponents* quantify the scale invariance and the long-range collective fluctuations in the critial region. In fact, the *scaling hypothesis* postulates that any property of the system that depends on the parameter controlling the transition, only depends on the parameter indirectly, through the scale of coherence. [76] As critical exponents of different critical systems coincide, they signify the same critical behavior in otherwise microscopically different systems, also referred to as universality. Hence, a level of "understanding" a continuous transition is reached when finding or characterizing its universality class by means of its critical exponents. [85]

2.2 PERCOLATION THEORY

2.2.1 Introduction

Percolation theory characterizes how global connectivity emerges in a system of a large number of objects. These objects connect according to some local rule constrained by an underlying topology such as a network or a regular geometric lattice. Given the topology and the local rule, percolation theory is about yielding the global, emergent behavior. [86, 87] Percolation abounds in nature, technical systems and social networks (see Stauffer and Aharony [86], Sahimi [88], Lee, Cho, and Kahng [89], and Saberi [90] and references therein). Early occurrences of percolation theory in the literature include the classic works by Flory [91] and Stockmayer [92] on polymerization and the sol-gel transition. However, it is only later that a theory of percolation starts to emerge. [93]

We say that a system *is at percolation*, or that a system *percolates*, if sufficiently many objects are locally connected such that global connectivity emerges. [87] This global connection is a continuous chain or cluster of locally connected objects, which is unbound in size in infinite systems, or of the order of the system size in finite systems. Typically, percolation also refers to a stochastic process of increasing connectivity and eventual emergence of the giant cluster. In an infinite system, this emergence in an ensemble of system configurations constitutes a geometrical phase transition. In fact, percolation is a phase transition paradigm. [86, 89] The central quantity in percolation settings is the distribution n_s of cluster sizes on a graph. A classical setting is that of a regular lattice of sites connected to their nearest neighbors. In *site percolation*, sites are subsequently picked or occupied, forming larger and larger clusters. In *bond percolation*, it is the links that are subsequently added to eventually form a giant cluster of connected sites.

In the following, we introduce the concepts and notation mainly according to Stauffer's and Aharony's classic textbook [86], before sketching recent developments in percolation theory.

2.2.2 The cluster size distribution

The cluster size distribution n_s is the fundamental quantity in percolation theory. In the regular lattice setting, a *cluster* is a maximum set of occupied sites which are pairwise joined by paths on the lattice only traversing occupied sites. In general, a cluster is a component of (occupied or connected) nodes of the underlying graph. The size s of a cluster is the number of nodes in the component. Infinite graphs allow for infinite cluster sizes. The occupation of sites, or the cluster sizes, typically depend on a (global) control parameter. For example, the paradigmatic percolation model of *Bernoulli percolation* is that each site is independently occupied with some probability p.

All the following statistics only require the general percolation setting of a graph. Let p denote the general control parameter in a percolation setting. In a finite system of N sites, the *cluster number* $n_s(p, N)$ is the number $N_s(p, N)$ of clusters of size s normalized by the total number N of sites:

$$n_s(p, N) = \frac{1}{N} N_s(p, N). \tag{2.1}$$

This definition also applies to systems of infinite size as

$$n_s(p) = \lim_{N \to \infty} \frac{1}{N} N_s(p, N). \tag{2.2}$$

2.2.3 Percolation threshold and characteristic cluster size

Typically, in an infinite system clusters grow with increasing parameter p, and at some critical value p_c, an infinite cluster appears. This value p_c is the *percolation threshold*. At and above p_c, there is an infinite cluster, and the system is said to *percolate*.

The probability that a system of finite size N percolates at parameter p is the probability $\Pi(p, N)$ that it contains a cluster of order of the system size. In an infinite system, we have the phase transition as

$$\Pi(p) = \lim_{N\to\infty} \Pi(p, N) = \begin{cases} 0 & p < p_c, \\ 1 & p \geqslant p_c. \end{cases} \tag{2.3}$$

The *percolation strength* is the fraction of sites belonging to the largest (or infinite) cluster. In the infinite system, the limit fraction is the typical order parameter of the percolation transition.

A typical form of the asymptotic tail of the cluster size distribution is

$$n_s(p) \sim s^{-\tau} \exp(-s/s_\xi), \quad (s \to \infty), \tag{2.4}$$

for large cluster sizes s and with some characteristic cluster size s_ξ. At the percolation transition, the characteristic cluster size s_ξ diverges as a power law

$$s_\xi \sim |p_c - p|^{-1/\sigma}, \quad (p \to p_c) \tag{2.5}$$

with the critical exponent σ.

In general, clusters of size $s < s_\xi$ dominate the moments of the cluster size distribution. These clusters effectively follow a power-law distribution $n_s(p) \sim s^{-\tau}$, as clusters of all sizes do at the critical point with $n_s(p_c) \sim s^{-\tau}$. Meanwhile, in the vicinity of the critical point, for $s \gg s_\xi$, the distribution is cut off exponentially. Thus, clusters larger than the charateristic cluster size do not exhibit critical behavior.

2.2.4 Average cluster size and correlation length

For any given site of any given finite cluster, the average size $S(p, N)$ of the cluster is defined as

$$S(p, N) = \frac{\sum_{s=1}^{\infty} s^2 n_s(p, N)}{\sum_{s=1}^{\infty} s n_s(p, N)} = \frac{M_2(p, N)}{M_1(p, N)}, \tag{2.6}$$

which is the second moment divided by the first moment of the cluster size distribution. Note that this average is different from the average of the (finite) cluster sizes in the system. The *average cluster size* $S(p, N)$ is defined with respect to a site, and thus, it is an intensive quantity.

Further note that for infinite systems ($N \to \infty$), these statistics exclude the infinite cluster. At the critical point, the average cluster size $S(p_c)$ nevertheless diverges as

$$S(p) \sim |p - p_c|^{-\gamma}, \quad (p \to p_c) \tag{2.7}$$

with the critical exponent γ. As S is the second moment of the cluster size distribution (up to the normalization factor), it is a measure of fluctuations in the system. Thus, divergence of S actually defines the percolation phase transition.

The divergence of quantities at the critical point involves sums over all cluster sizes s. The cutoff of the cluster number n_s at the characteristic cluster size $s_\xi \sim |p - p_c|^{1/\sigma}$ marks the cluster sizes $s \approx s_\xi$ that contribute the most to the sums and hence, to the divergence. This also holds for the correlation length ξ, which is the radius of those clusters of sizes $s \approx s_\xi$. As such, this is the one and only length scale which characterizes the behavior of an infinite system in the critical region.

The correlation length ξ defines the relevant length scale. As it diverges at $p \to p_c$, a length scale is absent at the percolation transition $p = p_c$. This lack of a relevant length scale is a typical example of scale invariance at a continuous phase transition. This implies that the system appears self-similar on length scales smaller than ξ. As ξ grows infinite at p_c, the whole system becomes self-similar. The lack of a relevant length scale also implies that functions of powers (power laws) describe the relevant quantities in the critical region. In particular, the correlation length itself diverges according to a power law as

$$\xi \sim |p - p_c|^{-\nu}, \quad (p \to p_c). \tag{2.8}$$

The functional form of this divergence is the same in all systems. The critical exponent ν depends only on general features of the topology and the local rule, giving rise to universality classes of systems with the same critical exponents.

2.2.5 Scaling relations

The scaling theory of percolation clusters relates the critical exponents of the percolation transition to the cluster size distribution. [94] In the absence of any length scale at the critical point, the cluster sizes also follow a power law

$$n_s(p_c) \sim s^{-\tau}, \quad (s \to \infty), \tag{2.9}$$

with the *Fisher exponent* τ. [95] The scaling assumption is that the ratio $n_s(p)/n_s(p_c)$ is a function of the ratio $s/s_\xi(p)$ alone [94], such that

$$\frac{n_s(p)}{n_s(p_c)} = f\left(\frac{s}{s_\xi(p)}\right), \quad (p \to p_c, s \to \infty). \tag{2.10}$$

As in the critical region, the characteristic cluster size diverges as $s_\xi \sim |p - p_c|^{-1/\sigma}$, we have $s/s_\xi(p) \sim |(p - p_c)s^\sigma|^{1/\sigma}$, and hence

$$n_s(p) \sim s^{-\tau} f((p - p_c)s^\sigma), \quad (p \to p_c, s \to \infty), \tag{2.11}$$

with some scaling function f which rapidly decays to zero, $f(x) \to 0$ for $|x| > 1 (s > s_\xi)$. [86]

The correlation length $\xi \sim s_\xi^{\sigma\nu}$ is the crossover length separating the critical and non-critical regimes. [86] The following scaling law relates the system dimensionality d and the fractal dimensionality $D = \frac{1}{\sigma\nu}$ of the infinite cluster to the exponents of the cluster size distribution: [87]

$$\frac{\tau - 1}{\sigma\nu} = d, \quad \tau = 1 + \frac{d}{D}. \tag{2.12}$$

Finally, consider the k-th raw moment of the cluster size distribution,

$$M_k(p) = \sum_s s^k n_s(p) \tag{2.13}$$

which scales in the critical region as

$$M_k(p) \sim \sum_s s^{k-\tau} \exp(-s/s_\xi(p)) \sim |p - p_c|^{(\tau-1-k)/\sigma} \quad (p \to p_c). \tag{2.14}$$

Similarly, in the critical region, the order parameter scales as

$$P(p) \sim \sum_s s(n_s(p_c) - n_s(p)) \sim \sum_s s^{1-\tau} \left(1 - \exp\left(-\frac{s}{s_\xi(p)}\right)\right) \tag{2.15}$$

$$\sim (p - p_c)^{(\tau-2)\sigma} = (p - p_c)^\beta \tag{2.16}$$

with critical exponent

$$\beta = \frac{\tau - 2}{\sigma}. \tag{2.17}$$

As the second raw moment $M_2(p) \sim |p - p_c|^{(\tau-3)/\sigma}$, we have the critical exponent

$$\gamma = \frac{3 - \tau}{\sigma}, \tag{2.18}$$

and the following relationships

$$\sigma = \frac{1}{\beta + \gamma}, \quad \tau = 2 + \frac{\beta}{\beta + \gamma}. \tag{2.19}$$

These are the scaling relations between the critical exponents, which all derive from the exponents τ and σ of the cluster size distribution.

2.2.6 Bond percolation on a regular lattice

In Bernoulli percolation settings each site or bond has an identical probability to be occupied or unoccupied, independent of the others. Here, we consider the classic bond percolation problems on a linear chain with two neighbors and a square two-dimensional lattice with four neighbors. In one dimension, when each bond is present with probability p, the probability to have a cluster of size s is

$$n_s = p^{s-1}(1 - p)^2. \tag{2.20}$$

For p near the critical value $p_c = 1$ we have

$$n_s = s^{-2}(s(p_c - p))^2 p^s = s^{-2}(s(p_c - p))^2 \exp(-(p_c - p)s) \tag{2.21}$$

which is of the postulated scaling form with $f(x) = x^2 \exp(-x)$. Hence, the exponents of the cluster size distributions are $\tau = 2$ and $\sigma = 1$, leading to critical exponents $\beta = 0, \gamma = 1, \nu = 1$ for the percolation transition. [96] Actually, in one dimension, the transition is discontinuous, as the infinite cluster emerges at $p_c = 1$ and contains all sites, leading to a discontinuous jump in the order parameter from 0 to 1 at p_c.

For the two-dimensional regular square lattice, the Fisher exponent is $\tau = \frac{187}{91} \approx 2.05$ and $\sigma = \frac{36}{91} \approx 0.396$, such that $\beta = \frac{5}{36} \approx 0.139, \gamma = \frac{43}{18} \approx 2.39, \nu = \frac{4}{3} \approx 1.33$. [79]

2.3 DIVERSE TYPES OF PERCOLATION TRANSITIONS

Exactly how a percolating cluster emerges and how this transition manifests itself in the cluster statistics is the main subject of percolation theory. The conventional order parameter to characterize the transition is the fraction of nodes that belong to the largest component. Whether the transition is discontinuous or continuous determines the tunability of the system at and towards the transition, as well as predictability due to precursors in the lead-up to the transition. [97–99]

Classic percolation models exhibit a continuous transition. The order parameter grows continuously from zero to finite size at the transition in infinite systems. Furthermore, correlation lengths and cluster sizes scale as a power law, the signature of criticality and a continuous transition. An example is the Erdös-Rényi model, a random network with controlled bond density p, the fraction of occupied bonds per node. [100] Another example is random percolation on a regular two-dimensional lattice. The cluster sizes scale as $n_s \sim s^{-\tau}$ with Fisher exponent $\tau = \frac{187}{91} > 2$.

A notable exception is the discontinuous transition (first-order phase transition) in one dimension. [101–103] In the infinite chain, the order parameter jumps from 0 to 1 at full connectivity. There is no signature of criticality as there is no power-law scaling: the cluster sizes distribute exponentially in the lead-up to the transition.

In contrast to these classical percolation models, competitive and non-reversible percolation models show intermediate behavior transcending the continuous vs. discontinuous dichotomy. [89, 90, 97, 104] While Bernoulli (random) percolation models add links independently, competitive processes select the next bond to add that fits best according to some model-specific rule. [105, 106] Typically, these rules delay the emergence of a giant component, which leads to an "explosive" growth of the order parameter at the transition. [89, 97, 107, 108] While this seemingly discontinuous transition has been proven to be actually continuous at the critical point [109], discontinuities can prevail [110] and shape the lead up to the transition. [98, 99]

Moreover, Sheinman, Sharma, Alvarado, Koenderink, and MacKintosh [111] recently reported a transition with critical signature but nevertheless discontinuous growth of the order parameter. Theoretically, and experimentally, cluster sizes distribute according to a power law with Fisher exponent $\tau \approx 1.8$ or $\tau \approx 1.9$. While the power law signifies a continuous transition, the Fisher exponent smaller than 2 entails a discontinuous order parameter. Indeed, the largest cluster size jumps from 0 to 1 at the critical point p_c in the thermodynamic limit. These findings are consistent with the notion of hybrid percolation transitions (sometimes referred to as mixed-order transitions) featuring characteristics of both discontinuous (first-order) and continuous transitions at the critical point and have been reported for various systems. [89, 112–115]

2.4 FINITE-SIZE SCALING

Scale invariance of critical infinite system manifests itself in scaling laws in the relevant quantities, and, moreover, according to the scaling hypothesis that these scaling laws originate from the diverging coherence length ξ. [79] How quantities that diverge in the infinite system scale with the size of a truncated, finite version of that system, is subject of the following finite-size scaling ansatz. According to the scaling hypothesis, a quantity $A(p)$ that diverges as $|p - p_c|^{-\zeta}$ in the infinite system with some critical exponent ζ should scale as

$$A(p, L) = |p - p_c|^{-\zeta} f\left(\frac{L}{\xi}\right) \sim \xi^{\zeta/\nu} f\left(\frac{L}{\xi}\right) = A(\xi, L) \tag{2.22}$$

with system size L and coherence length $\xi = \xi(p)$ of the infinite system ($L \to \infty$). [79, 116–118] For a system of size much larger than the coherence length, the system is effectively infinite, and as such we have

$$A(\xi, L) \sim \xi^{\zeta/\nu}, \quad (L \gg \xi, p \to p_c). \tag{2.23}$$

For a system of size much smaller than the coherence length of the infinite system ($L \ll \xi$), the coherence is cut off already at L rather than ξ, and we expect at or near the critical point a scaling with system size as

$$A(p, L) \sim L^{\zeta/\nu}, \quad (L \ll \xi, p \to p_c). \tag{2.24}$$

These considerations constitute the finite-size scaling ansatz [116–118]

$$A(\xi, L) = \xi^{\zeta/\nu} f\left(\frac{L}{\xi}\right), \quad (L \to \infty, p \to p_c). \tag{2.25}$$

with the scaling function

$$f(x) \begin{cases} = \text{const.} & \text{for } |x| \gg 1, \\ \sim x^{\zeta/\nu} & \text{for } x \to 0. \end{cases} \tag{2.26}$$

The scaling function $f(x)$ is a dimensionless function of the dimensionless ratio L/ξ of the finite system size and the infinite-system coherence length in the critical region. This ratio controls the finite-size effects. The conventional scaling function is $\tilde{f}(x) = x^{-\zeta} f(x^{\nu})$ [116, 117] such that

$$A(p, L) = L^{\zeta/\nu} \tilde{f}\left(L^{1/\nu}(p - p_c)\right), \quad (L \to \infty, p \to p_c), \tag{2.27}$$

with

$$\tilde{f}(x) \begin{cases} = \text{const.} & \text{for } x \to 0 \quad (L \ll \xi), \\ \sim L^{-\zeta/\nu}(p - p_c)^{-\zeta} & \text{for } |x| \gg 1 \quad (L \gg \xi). \end{cases} \tag{2.28}$$

3 STOCHASTIC PROCESSES

3.1 DEFINITIONS

3.1.1 Overview

The time evolution of a typical deterministic dynamical system is the solution of some differential equation (in continuous time) or given by a map (in discrete time). [119] This applies to both classical deterministic systems (e. g. planetary trajectories in the solar system) and quantum-mechanical systems (e. g. evolution of atomic states).[1] [120]

In contrast, a *stochastic process* describes the non-deterministic time evolution of a dynamical system. This does not only apply to inherently probabilistic systems (e. g. radioactive decay). It also applies to thermodynamic or chaotic systems (e. g. Brownian motion, or the weather) that evolve deterministically in a huge number of degrees of freedom. Nevertheless, the probabilistic description as a stochastic process makes the time evolution of these systems tractable.[2]

3.1.2 Probability spaces and random elements

The core notion of probability theory is a *probability space* and *random variables* defined on it. In the following, I assume familiarity with the distinct concepts of metric spaces and measure spaces.

Definition 3.1 ([122, Definition 1.1]). *A* probability space *is a triple* $(\Omega, \mathcal{F}, P)$ *where* Ω *is any set, referred to as the set of* outcomes*, and* $\mathcal{F}$ *is a* σ*-algebra of subsets of* Ω*, referred to as* events*, and* $P : \mathcal{F} \mapsto [0, 1]$ *a* probability measure *that assigns a probability to each event, with* $P(\Omega) = 1$*.*

The concept of a *random element* unifies the notion of random variables, stochastic processes and other quantities and mappings associated with a probability space $(\Omega, \mathcal{F}, P)$. Similarly, the concept of a *Polish space* unifies the natural and real numbers, arbitrary countable sets, Euclidian spaces, as well as function spaces equipped with a metric:

Definition 3.2 ([123, p. 409]). *Let* S *be a metric space that is complete (each Cauchy sequence is convergent) and separable (there is a countable dense set in* S*). Let* $\mathcal{S}$ *be the (canonical) Borel* σ*-algebra generated by the open sets induced by the metric. Then call the space* $(S, \mathcal{S})$ *a* Polish space*.*

Definition 3.3 ([123, p. 194]). *Let* S, S' *be two Polish spaces. Let* p *be a function* $S \times \mathcal{S}' \to [0, 1]$ *such that* $p(x, \cdot) : \mathcal{S}' \to [0, 1]$ *is a probability measure on* S'

1 While quantum mechanics is inherently probabilistic, the time evolution of a quantum-mechanical state according to the Schrödinger equation is deterministic.

2 See Werndl [121] for a discussion of observational equivalence of stochastic processes and deterministic systems.

and $p(\cdot, B) : S \to [0,1]$ *is measurable for all* $x \in S, B \in \mathcal{S}'$. *Then* p *is called a* probability kernel from S to S'.

Definition 3.4 ([123, p. 409]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space, and let* $(S, \mathcal{S})$ *be a Polish space. A* random element *in* S *is a measurable mapping* $X : \Omega \mapsto S$. *Its* probability distribution *is the probability measure*

$$F_X(B) = P\{\, X \in B \,\} = P \circ X^{-1}(B), \qquad B \in \mathcal{S}.$$

The probability distribution already contains all of the probability information about a random element X without considering other random elements, even without explicity constructing a rather abstract probability space $(\Omega, \mathcal{F}, P)$. [123, p. 406]

Definition 3.5. *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space. A* random variable $X : \Omega \to \mathbb{R}$ *is a random element in* $\mathbb{R}$. *The function* $F : \mathbb{R} \to [0,1]$ *with*

$$F(x) = P\{\, X \leqslant x \,\} \equiv P\{\, \omega : X(\omega) \leqslant x \,\}, \quad x \in \mathbb{R}$$

is monotonously increasing and is called the distribution function *of* X.

Definition 3.6 ([123, p. 406]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space. A statement about events or random elements is said to hold* almost surely *if the statement holds with probability one.*

3.1.3 Random functions and stochastic processes

Following Khintchine [124], and subsequently Serfozo [123] and Capasso and Bakstein [122], a stochastic process is a collection of random variables on the same probability space, which take values in the same (Polish) state space. Ultimately, one is interested in the probability law for the set of trajectories of the stochastic process.

Definition 3.7 ([125, Definition 1.2]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space,* S *be a Polish space, and let* $\mathbb{T}$ *be any set. A* random function X *with state space* S *and parameter set* $\mathbb{T}$ *is a family* $\{\, X(t); t \in \mathbb{T} \,\}$ *of random elements* $X(t) : \Omega \to S, \omega \mapsto X(t, \omega)$ *in* S, *indexed by elements* $t \in \mathbb{T}$.

A discrete-time stochastic process is a random function with parameter set $\mathbb{T} = \mathbb{N}$:

Definition 3.8 ([123, p. 409]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space,* S *be a Polish space. A* discrete-time stochastic process *with state space* S *is a collection of random elements* $X = \{\, X_n : n \in \mathbb{N} \,\}$ *in* S *on* $(\Omega, \mathcal{F}, P)$. *The value* $X_n(\omega) \in S$ *is the* state *of the process at time* n *associated with the outcome* ω.

Serfozo [123, p. 409] points out that the discrete-time stochastic process X is a random element in S^∞.

A continuous-time stochastic process is a random function with parameter set $\mathbb{T} = \mathbb{R}_0^+$:

Definition 3.9 ([123, p. 410]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space,* S *be a Polish space. A* continuous-time stochastic process *with state space* S *is a collection of random elements* $X = \left\{\, X(t) : t \in \mathbb{R}_0^+ \,\right\}$ *in* S *on* $(\Omega, \mathcal{F}, P)$. *The value* $X(t, \omega) \in S$ *is the* state *of the process at time* t *associated with the outcome* ω.

Definition 3.10 ([125, p. 1]). *Let* X *be a random function (stochastic process) with state space* S *and parameter set* $\mathbb{T}$ *on some probability space* $(\Omega, \mathcal{F}, P)$. *Given an elementary outcome* $\omega \in \Omega$, *the function* $X(\cdot, \omega) : \mathbb{T} \to S, t \mapsto X(t, \omega)$ *is called the* trajectory, *the* realization *or the* sample path *of the random function (stochastic process)* X *associated with the outcome* ω.

So far, the rather abstract probability space $(\Omega, \mathcal{F}, P)$ holds all the probability information about a stochastic process. [123, p. 410] To make this probability information accessible, one defines push-forward probability measures on a finite set of points in time. For a fixed number $m \in \mathbb{N}$ of points in time $t_1, \dots, t_m \in \mathbb{T}$, consider the product state space $(S^m, \mathcal{S}^m)$ with the appropriately defined product σ-algebra $\mathcal{S}^m$. Further consider the vector-valued function $X_{t_1,\dots,t_m} : \Omega \to \mathcal{S}^m, \omega \mapsto (X(t_1, \omega), \dots, X(t_m, \omega))$. This function is measurable, i. e. a random element with values in S^m.

Definition 3.11 ([125, Definition 1.3]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space, and let* X *be a random function (stochastic process) with state space* S *and parameter set* $\mathbb{T}$. *Let* $m \in \mathbb{N}$ *and* $t_1, \dots, t_m \in \mathbb{T}$. *The push-forward probability measure* $P^X_{t_1,\dots,t_m} : \mathcal{S}^m \to [0, 1]$ *given as*

$$\begin{aligned} P^X_{t_1,\dots,t_m} &= P \circ X^{-1}_{t_1,\dots,t_m}, \\ P^X_{t_1,\dots,t_m}(B) &= P\{\, \omega \in \Omega : (X(t_1, \omega), \dots, X(t_m, \omega)) \in B \,\}, \end{aligned}$$

is called a finite-dimensional distribution *of the random function (stochastic process)* X.

Furthermore, the set $\left\{ P^X_{t_1,\dots,t_m} : m \in \mathbb{N}, t_1, \dots, t_m \in \mathbb{T} \right\}$ *is called the* family of finite-dimensional distributions *of the random function (stochastic process)* X.

According to the Kolmogorov extension theorem, a family of finite-dimensional distributions that fulfill mild consistency conditions probabilistically defines a stochastic process (and guarantees its existence). [125, Theorem 1.1] The probability space $(\Omega, \mathcal{F}, P)$ does not even need to be unique. This is why stochastic processes are typically sufficiently defined via their finite-dimensional distributions, or by specifying the time evolution of the process, rather than through the explicit construction of a probability space.

3.2 POINT PROCESSES

3.2.1 General point processes

Let us capture the notion of a random set of points in some (Polish) space $(S, \mathcal{S})$. But first, observe that each finite or countable set of points $\{\, x_n \,\}_n$ in S induces a counting measure

$$N = \sum_n \delta_{x_n} : \mathcal{S} \to \mathbb{N}, B \mapsto \sum_n \delta_{x_n}(B)$$

with the Dirac measure $\delta_x : \mathcal{S} \to \{\, 0, 1 \,\}, \delta_x(B) = 1 \Leftrightarrow x \in B$. Furthermore, the set of points should have no accumulation points, i. e. the counting measure shall be finite for each $B \in \hat{\mathcal{S}}$, where $\hat{\mathcal{S}} \subset \mathcal{S}$ is the family of bounded Borel subsets of S.

Definition 3.12 ([123, pp. 181–182]). *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space and* $(S, \mathcal{S})$ *be a Polish space. Let* $\hat{\mathcal{S}} \subset \mathcal{S}$ *denote the family of bounded Borel sets in* S. *A* random set of points $\{X_n\}_n$ *in* S *is a random element in the set of at most countable subsets* $\left\{ \{x_n\}_{n=0}^{N} \mid N \in \mathbb{N} \cup \{\infty\}, x_n \in S \, \forall n \right\}$ *such that in each bounded Borel set* $B \in \hat{\mathcal{S}}$ *the number of points is finite:*

$$N(B)(\omega) = \sum_n \delta_{X_n(\omega)}(B) < \infty, \quad B \in \hat{\mathcal{S}}, \omega \in \Omega$$

where N *now is a random element in the set of all counting measures* ν *on* S *for which* $\nu(B) < \infty$ *for* $B \in \hat{\mathcal{S}}$. N *is called a* point process *on* S *and* X_n *are the points of* N.

Note that a point process is not a stochastic process, as the points are not ordered but merely labelled in interchangeable order. In fact, the counting measure N is invariant under permutation of the labels n of the set of points $\{X_n\}_n$ that induces N.

A point process N is characterized (and guaranteed to exist) by specifying the finite-dimensional distributions on bounded Borel sets that generate $\mathcal{S}$,

$$\left\{ P_m(N(B_i) = n_i, i = 1, \dots, m) : m \in \mathbb{N}, n_i \in \mathbb{N}, B_i \in \hat{\mathcal{S}}, i = 1, \dots, m \right\},$$

a family of probability measures on $\left(S^m, \hat{\mathcal{S}}^m\right)$. [123, 126]

Definition 3.13 ([126, Definition 9.1.II]). *Let* S *be a Polish space and let* N *be a point process on* S. N *is called a* simple point process *on* S *if* $N(\{x\}) \in \{0, 1\}$ *for all* $x \in S$.

In particular, all points of a simple point process are distinct.

3.2.2 Arrival processes

We now consider simple point processes with points $\{T_i\}_{i=1}^{\infty}$ in time, i. e. on the positive real half-line $\mathbb{R}^+$. The most general notion of such point processes is that of an arrival process:

Definition 3.14 ([127]).

1. *An* arrival process *is a sequence of increasing finite random variables* $\{T_i\}_{i=1}^{\infty}$ *with* $0 < T_1 < T_2 < \cdots$ *and* $T_n \to \infty$ *as* $n \to \infty$ *almost surely. The random variable* T_i *is the* i*-th* arrival epoch *or* occurrence time.

2. *Equivalently, an* arrival process *is a sequence of positive finite random variables* $\{X_i\}_{i \in \mathbb{N}}$ *with* $\sum_{i=0}^{n} X_i \to \infty$ *as* $n \to \infty$, *where the random variable* $X_i > 0$ *is called the* i*-th* interarrival time *or* inter-occurrence time.

3. *Equivalently, an* arrival process *is a simple point process* N *on* $\mathbb{R}^+$ *where the counting random variable* $N(t)$ *represents the cumulative number of arrivals in the time interval* $(0, t]$ *with* $N(0) = 0$ *and* $N(t) \to \infty$ *as* $t \to \infty$.

3.2.3 Marked point processes

Consider an arrival process $N(t)$ where the occurrences are of different types, i. e. each occurrence event T_i features a random mark Y_i in a (Polish) mark space $(S, \mathcal{S})$. The mark Y_i only depends on T_i according to a probability kernel $p : \mathbb{R}^+ \times \mathcal{S} \to [0, 1]$, where $p(t, \cdot)$ is the probability measure on S for assigning a mark in a Borel set $B \in \mathcal{S}$ to an event occurring at time t. [123, p. 194]

Definition 3.15 ([123, p. 194])**.** *Let* $N(t)$ *be an arrival process with arrival times* T_i*, i. e. a simple point process* $N = \sum_n \delta_{T_n}$ *on* $\mathbb{R}^+$*. Let* $p : \mathbb{R}^+ \times \mathcal{S} \to [0, 1]$ *be a probability kernel. Let* $M = \sum_n \delta_{(T_n, Y_n)}$ *be a simple point process on* $\mathbb{R}^+ \times S$ *such that*

$$P\{ Y_i \in B_i (i = 1, \dots, n) \} = p(T_1, B_1) \cdots p(T_n, B_n) \quad B_i \in \mathcal{S}, n \in \mathbb{N}.$$

The point process M *is called a* marked point process *associated with* N*, and* Y_i *is the* mark *of* T_i*.*

3.3 renewal processes & poisson processes

The concept of a renewal process embodies the notion of a stochastic process starting over and over again:

Definition 3.16 ([123, p. 100])**.** *Let* N *be an arrival process with occurrence times* $0 < T_1 < T_2 < \dots$ *and inter-occurrence times* $\{ X_i = T_{i+1} - T_i \}_{i \in \mathbb{N}}$ *with* $T_0 = 0$*. The arrival process* N *is called a* renewal process *if the inter-occurrence times are independent and identically distributed. Its occurrence times* $\{ T_i \}$ *are called* renewal times *or* renewal epochs*, its inter-occurrence times* $\{ X_i \}$ *are called* inter-renewal times*, and* $N(t)$ *is the* number of renewals *in the time interval* $(0, t]$*.*

To define a renewal process, it suffices to specify the distribution function F with $F(0) = 0$ for the iid (independent and identically distributed) inter-renewal times. [123] As the inter-renewal times are iid, for a renewal time $T_n = \tau$, the process $\{ N(\tau + t) - N(\tau), t \geqslant 0 \}$ is again a renewal process with iid inter-renewal times with the same distribution as the original process N. [127] This is meant when it is said that a renewal process probabilistically starts over again at each occurrence time. Furthermore, as Gallager [127] notes, when studying stochastic processes with renewal occurrences (regenerative processes), this characteristic property of renewal processes allows to separately investigate the long-term behavior (depending on the distribution of the inter-renewal times), and the short-term behavior within each renewal period.

To quantify the average long-term behavior of a renewal process, renewal theory considers limiting time-averages for individual outcomes ω, such as the limiting time-average renewal rate $\lim_{t \to \infty} \frac{N(t)}{t}$. Furthermore, it considers limiting ensemble averages such as the limiting ensemble average renewal rate $\lim_{t \to \infty} \mathbb{E}\left[\frac{N(t)}{t}\right]$.

Definition 3.17. *Let* $N(t)$ *be a renewal process. Denote by* $\mathbb{E}[X]$ *the mean inter-renewal time.*

Theorem 3.18 (Strong Law of Large Numbers for Renewal Processes [127])**.** *Let* $N(t)$ *be a renewal process with finite mean inter-renewal time* $\mathbb{E}[X] < \infty$*. Then almost surely* $\lim_{t\to\infty} \frac{N(t)}{t} = \frac{1}{\mathbb{E}[X]}$.

Theorem 3.19 (Elementary Renewal Theorem [127])**.** *Let* $N(t)$ *be a renewal process with mean inter-renewal time* $\mathbb{E}[X]$ *either finite or infinite. Then*

$$\lim_{t\to\infty} \frac{\mathbb{E}[N(t)]}{t} = \frac{1}{\mathbb{E}[X]}.$$

Hence, time average and limiting ensemble average inter-renewal time coincide (if they exist).

Definition 3.20. *Let* $N(t)$ *be a renewal process with finite mean inter-renewal time* $\mathbb{E}[X] < \infty$*. Then call* $\frac{1}{\mathbb{E}[X]}$ *the* renewal rate *of* N.

We have seen that renewal processes start over again at their renewal times. Are there arrival processes that indeed probabilistically start over at *any* time t, irrespective of an event occurring at t, and irrespective of when the last event occurred before t?

Definition 3.21. *Let* $N(t)$ *be a renewal process with inter-renewal times that are exponentially distributed with parameter* $\lambda > 0$*. The renewal process* $N(t)$ *is called a* Poisson process with rate λ.

The exponential distribution and an exponentially distributed random variable X are *memoryless*, as X satisfies [127]

$$P\{X > t + x \mid X > t\} = P\{X > x\} \quad \forall x, t \geqslant 0.$$

It is in this sense that the Poisson process "looks the same" from whichever time one inspects it.

Let us define a Poisson process in terms of its finite-dimensional distributions:

Theorem 3.22 ([123, 127])**.** *Let* $N(t)$ *be a Poisson process with rate* λ*. Then* N *has* independent increments, *i. e.* $N(I_1), \dots, N(I_n)$ *are independent for disjoint finite intervals* $I_1, \dots, I_n$*. Furthermore, for each finite interval* $I \subset \mathbb{R}_0^+$*, the random variable* $N(I)$ *is Poisson distributed with parameter* $\lambda|I|$*, i. e.*

$$P\{N(I) = n\} = \frac{1}{n!}(\lambda|I|)^n \exp(\lambda|I|).$$

3.4 MARKOV CHAINS

Stochastic processes capture the probabilistic notion of a dynamic system evolving in time. Classical time-discrete dynamical systems are represented by maps f that evolve their next state $x_{n+1} = f(x_n)$ deterministically depending solely on their respectively current state x_n (the very definition of state). Arguably, a Markov chain is what comes closest to a map for time-discrete stochastic processes, as its state X_{n+1} depends solely on the last state X_n (rather than the whole past of the process).

3.4.1 Markov chains on countable spaces

Definition 3.23 ([123, 127, 128]). *Let* S *be a countable set, and let* $X = \{ X_n : n \in \mathbb{N} \}$ *be a discrete-time stochastic process with values in* S. *Such a process* X *is a* Markov chain *if its random state* X_n *for each* $n > 0$ *depends on the past only through the previous state* X_{n-1} *(*Markov property*), that is*

$$P\{ X_n = j \mid X_0 = i_0, \ldots, X_{n-1} = i_{n-1} \} = P\{ X_n = j \mid X_{n-1} = i_{n-1} \}$$

for all $n > 0, j, i_0, \ldots, i_{n-1} \in S$. *For all* $x, y \in S$, *let* $P(x,y)$ *denote the transition probability from state* x *to state* y,

$$P(x,y) = P\{ X_n = y \mid X_{n-1} = x \},$$

such that $P(x,y) \geqslant 0$ *and* $\sum_{z \in S} P(x,z) = 1$ *for* $x, y \in S$. *Recursively define the* n-step transition matrix *as*

$$P^n(x,z) = \sum_{y \in S} P(x,y) P^{n-1}(y,z)$$

with P^0 *defined as the identity* $P^0(y,z) = \delta_{yz}$, *so that for all* $x, y \in S$, $P^n(x,y) = P\{ X_n = y \mid X_0 = x \}$. *The probability measure* $\mu(x) = P\{ X_0 = x \}$ *is the* initial distribution *of the chain.*

Theorem 3.24 ([128]). *Let* S *be a countable space, and let* $\mu : S \to [0,1]$ *be an initial probability measure on* S, *and further let* $P(x,y)$ *be transition probabilities such that* $P(x,y) \geqslant 0$ *and* $\sum_{z \in S} P(x,z) = 1$ *for* $x, y \in S$. *Then there is a Markov chain* X_n *such that*

$$P\{ X_n = y \mid X_{n-1} = x, \ldots, X_0 = x_0 \} = P(x,y) \quad n > 0, x, y, x_0 \in S.$$

and $P\{ X_0 = x_0 \} = \mu(x_0)$ *for* $x_0 \in S$.

Theorem 3.25 ([123, p. 8]). *Let* S *be a countable space, let* S' *be a Polish space, let* $f : S \times S' \to S$ *be measurable and let* X_n *be a time-discrete stochastic process on* S *such that*

$$X_n = f(X_{n-1}, Y_n), \quad n > 0$$

with $Y_1, Y_2, \ldots$ *iid random variables with values in* S' *independent of* X_0. *Then,* X_n *is a Markov chain with transition probabilities* $P(x,y) = P\{ f(x, Y_1) = y \}$.

3.4.2 Markov chains on general spaces

Theorem 3.26 ([128]). *Let* $(S, \mathcal{S})$ *be a Polish space (in fact,* S *could be any space endowed with a countably generated σ-algebra* $\mathcal{S}$*). Let* μ *be a probability measure on* $\mathcal{S}$, *and let* $P(x, B)$ *be a probability kernel for all* $x \in S, B \in \mathcal{S}$. *Then there exists a discrete-time stochastic process* X_n *such that for* $n > 0, B_0, \ldots, B_n \in \mathcal{S}$

$$\begin{aligned} &P\{ X_0 \in B_0, \ldots, X_n \in B_n \} \\ &= \int_{x_0 \in B_0} \cdots \int_{x_{n-1} \in B_{n-1}} \mu(dx_0) P(x_0, dx_1) \cdots P(x_{n-1}, B_n) \end{aligned}$$

and $P\{ X_0 \in B_0 \} = \mu(B_0)$ *for* $B_0 \in \mathcal{S}$.

Definition 3.27 ([128]). *Such a discrete-time stochastic process* X_n *is called a* Markov chain *on* $(S, \mathcal{S})$ *with transition probability kernel* $P(x, B)$ *and initial distribution* μ. *Recursively define the* n-step transition probability kernel *as*

$$P^n(x, B) = \int_S P(x, dy) P^{n-1}(y, B) \quad x \in S, B \in \mathcal{S}$$

with $P^0(x, B) = \delta_x(B)$, *so that for all* $x, y \in S$ *we have* $P\{X_n = y \mid X_0 = x\} = P^n(x, y)$.

3.4.3 First passages and returns

The following definitions are formulated for Markov chains on general Polish spaces. The definitions easily transfer to Markov chains on a countable space by regarding single elements y instead of Borel sets B, where applicable.

Definition 3.28 ([127, 128]). *Let* X_n *be a discrete-time stochastic process on a Polish space* S, *and let* $B \in \mathcal{S}$. *The* occupation number η_B *is the random number of (possibly infinite) visits of* X *to* B:

$$\eta_B = \sum_{n=1}^{\infty} \delta_{X_n}(B).$$

The event that the process visits the set $B \in \mathcal{S}$ *infinitely often after starting at* $x \in S$ *has the probability*

$$Q(x, B) = P\{\eta_B = \infty \mid X_0 = x\}.$$

For $n > 0$ *define the* first-passage-time probability (kernel) $f_n : S \times \mathcal{S} \to [0, 1]$ *from state* $x \in S$ *to set* $B \in \mathcal{S}$ *as the probability that* n *is the smallest* i *for which* $X_i \in B$ *given that* $X_0 = x$:

$$f_n(x, B) = P\{X_n \in B, X_{n-1} \notin B, \dots, X_1 \notin B \mid X_0 = x\}$$

with $f_1(x, B) = P(x, B)$. *Furthermore, for* $n > 0$, *let* $F_n : S \times \mathcal{S} \to [0, 1]$ *be the probability (kernel) that the process starting at* $x \in S$ *visits a set* $B \in \mathcal{S}$ *between times* 1 *and* n, *inclusive:*

$$F_n(x, B) = \sum_{i=1}^{n} f_i(x, B).$$

The first return time τ_B *is the random time after* 0 *when the process first enters* B *(or when it first returns to* B, *if* $X_0 \in B$*):*

$$\tau_B = \min\{n > 0 : X_n \in B\}.$$

Given that the process starts in x, *the probability distribution of* τ_B *is*

$$P\{\tau_B = n \mid X_0 = x\} = f_n(x, B).$$

For $x \in S$ *and* $B \in \mathcal{S}$, *define the* return probabilities *as the probability to return to* B *(in finite time) when starting in* x:

$$L(x, B) = P\{\tau_B < \infty \mid X_0 = x\} = \sum_{n=1}^{\infty} P\{\tau_B = n \mid X_0 = x\} = F_\infty(x, B)$$

Definition 3.29 ([128]). *Let X_n be a Markov chain on a Polish space S with n-step transition probability kernel P^n. Define the auxiliary probability kernel $U : S \times \mathcal{S}$ as*

$$U(x, B) = \sum_{n=1}^{\infty} P^n(x, B) \quad (x \in S).$$

We have for all $x \in S, B \in \mathcal{S}$ the expected number of returns to B after starting at x as $\mathbb{E}[\eta_B \mid X_0 = x] = U(x, B)$.

3.4.4 Irreducibility

Irreducibility of a Markov chain guarantees that the chain eventually visits all regions of its state space:

Definition 3.30 ([128]). *Let S be a Polish space. A Markov chain X_n on S is φ-irreducible if there is a measure φ on S such that for all $x \in S, B \in \mathcal{S}$:*

$$\varphi(B) > 0 \Rightarrow L(x, B) > 0.$$

Theorem 3.31 ([128]). *Let X_n be a Markov chain on a Polish space S. The following statements are equivalent: X is φ-irreducible. $\varphi(B) > 0 \Rightarrow U(x, B) > 0$ for all $x \in S, B \in \mathcal{S}$.*

Theorem 3.32 ([128]). *Let X be a φ-irreducible Markov chain on a Polish space S for some measure φ. Then there exists an "essentially unique maximal" irreducibility measure ψ on S such that*

1. *X is ψ-irreducible.*
2. *$\psi(B) = 0 \Rightarrow \psi\{x \in S : L(x, B) > 0\} = 0$ for all $B \in \mathcal{S}$.*
3. *$\psi(S \setminus B) = 0 \Rightarrow B = B_0 \cup N : \psi(N) = 0, P(x, B_0) = 1$ for all $x \in B_0$ (B_0 is absorbing).*

Definition 3.33 ([128]). *A Markov chain X is ψ-irreducible if it is φ-irreducible for some measure φ and if the measure ψ is a maximal measure according to the preceding theorem. Define the family of sets of positive ψ measure as*

$$\mathcal{S}^+ = \{B \in \mathcal{S} : \psi(B) > 0\}.$$

The set $\mathcal{S}^+$ is the same for different maximal irreducibility measures, and hence, $\mathcal{S}^+$ is well-defined. [128] For a countable state space S, the maximal irreducibility measure is the counting measure.

3.4.5 Transience and recurrence

Recurrence is a weak notion of stability of a Markov chain X. A recurrent chain X visits every set of positive measure infinitely often. Contrarily, a transient chain visits bounded sets only a finite number of times, and eventually leaves any such set. Specifically, we consider recurrence and transience in terms of the occupation number random variable η_B.

Definition 3.34 ([128]). *Let X be a Markov chain on a Polish space S. A set $B \in \mathcal{S}$ is* uniformly transient *if there exists an upper bound $M < \infty$ such that $U(x, B) \leqslant M$ for all $x \in B$. A set $B \in \mathcal{S}$ is* recurrent *if $U(x, B) = \infty$ for all $x \in B$. A set $B \in \mathcal{S}$ is* transient *if there is a countable cover of B by uniformly transient sets.*

Definition 3.35 ([128]). *Let X be a ψ-irreducible Markov chain on a Polish space S. The chain X is* recurrent *if every set $B \in \mathcal{S}^+$ is recurrent. The chain X is transient if S is transient.*

Theorem 3.36 ([128]). *Let X be a ψ-irreducible Markov chain on a Polish space S. Then X is either recurrent or transient.*

Definition 3.37 ([128]). *Let X be a Markov chain on a Polish space S. A set $B \in \mathcal{S}$ is* Harris recurrent *if $Q(x, B) = 1$ for all $x \in B$. The chain X is* Harris recurrent *if it is ψ-irreducible and every set $B \in \mathcal{S}^+$ is Harris recurrent (or equivalently, it holds that $L(x, B) = 1$ for all $x \in S$).*

Hence, Harris recurrence is stronger than recurrence: The *expected* number of visits to a *recurrent* set is infinite, while a *Harris recurrent* set is visited infinitely often *almost surely*.

Theorem 3.38 ([128]). *Let X be a recurrent Markov chain on a Polish space S. Then*

$$X = H \cup N$$

with an absorbing and nonempty set H and a transient set N with $\psi(N) = 0$. Every subset of H in $\mathcal{S}^+$ is Harris recurrent.

The theorem implies that the restriction of a recurrent chain X to H differs to the original chain only by a ψ-null set. At the same time, the restriction to H yields stronger stability results in terms of Harris recurrence. For a countable state space S, the set N is empty: a recurrent chain on a countable state space is also Harris recurrent.

3.4.6 Stochastic recursive sequences

Stochastic recursive sequences generalize the notion of Markov chains to discrete-time stochastic processes. Rather than by a sequence of iid random variables, they are driven by an arbitrary random sequence:

Definition 3.39 ([129, p. 507]). *Let S and S' be two Polish spaces. Let ξ_n be a sequence of random elements on S'. Let f be a deterministic measurable function $S \times S' \to S$. A time-discrete stochastic process X_n on S is a* stochastic recursive sequence driven by the sequence ξ_n *if X_n satisfies the relation*

$$X_n = f(X_{n-1}, \xi_n), \quad n > 0$$

with X_0 independent of ξ_n.

As Borovkov [129, p. 17] points out, each Markov chain is a stochastic recursive sequence driven by iid ξ_n. Furthermore, there is a notion of *renovating* events of the process X_n from which on only the driving sequence ξ_n determines the evolution of the process rather than the states X_n before the event. The notion of renovating events is weaker than renewals, but nevertheless allows to infer long-term behavior and ergodic properties. [129]

4 DISCRETE-EVENT SYSTEMS

4.1 SYSTEM THEORETIC MODELING AND SIMULATION

Studying entities in the real world means running experiments on them. The *experimental frame* describes the conditions for real-world experimentation. In particular, the experimental frame of choice restricts the data that the experimentator observes and how she views and records them. [130] Thus, the experimental frame defines the questions one is able to ask, and possibly, answer by experiment. [131]

A mathematical *model* of a real-world entity is an abstraction that facilitates analysis and understanding of the entity. Modeling aims at generating model behavior which matches the behavior of the real entity under study. Accordingly, the *experimental frame of the model* describes the context of the model, the simplifying assumptions, and possible questions and results. [131] A mathematical *model* is a set of equations. Solving the model means solving these equations. Consequently, the solution of the model predicts the results of all possible experiments. Still, these predictions need to match the experimental measurements on the real entity. Experimentally checking the model predictions is the *validation* of the model.

There are models that lack a solution. For instance, a model might not have a closed analytical solution, or the solution has not been found yet. However, each model permits to numerically *compute* approximate results. Here it is the *computational experimental frame* that describes the context of the computation. For example, this is the computer code that contains all the parameters, provides an interface to output the results, and actually calls the computation. The numerical results need to match the model predictions, and ultimately, the real-world behavior. Checking the numerical computation accordingly is the *verification* of the computation. Computer-based numerical computation is also called *simulation*. Simulation is particulary well-suited to tackle complex problems inaccessible to detailed general analytical solutions. That is, simulation simply computes the behavior for a particular set of parameters.

Systems theory regards a *system* as a set of *components* and their interactions. In this reductionist ansatz, the behavior of all individual components and their interactions constitute the system behavior. [131] Commonly, nonlinear interactions among the components result in emergent system behavior: an effective behavior on a different level of description that is more than the simple sum of the component behaviors. [73] The system does not need to be a natural, real entity; also artifical or abstract entities are systems, as long as the components and interactions are mathematically describable.

In general, a model of a system specifies how the components interact and evolve in time. A typical example is an ordinary differential equation (ODE) that models how a certain physical quantity evolves continuously in time. A system provides observable input/output behavior. Hence, Zeigler, Kim, and Praehofer [130] refer to the system as the *source system*. Each component has its own input and output. How exactly a component reacts to input and subsequently produces output depends on the internal state of the component. The time evolution of these observables are the *trajectories* of the input/output variables. Accordingly, the model of the source system needs to produce trajectories that match the trajectories of the source system.

Specifically, a *simulation model* is a structure and a set of rules to represent the input/output behavior of a system at the relevant level of description. [130] This set of rules describes the step-wise dynamics of the model, also called the *local dynamics*.

Simulation means some agent executes these local rules. Such a *simulator* is an abstract concept, encompassing any computational entity that processes the model set of rules to generate the model output trajectories. This includes algorithms, brains, and CPUs. The model output trajectories subject to certain input trajectories is also called the *global dynamics*.

This Chapter is structured as follows. Section 4.2 introduces and discusses models with trajectories that only change at discrete instants of time (*discrete-event models*). Section 4.3 presents the Discrete-Event System Specification (*DEVS*) and its formalism to formally define such discrete-event models and their local rules. Finally, the global dynamics of a DEVS model is the subject of Section 4.4.

4.2 DISCRETE-EVENT MODELS

Models of dynamical systems in nature typically comprise a set of differential equations. These equations describe how the model state continously changes over a continuous time interval. Conversely, human-engineered systems typically involve abrupt state changes. For instance, consider traffic lights turning green, switching on the lights at home, an ICE high-speed train arriving at Hamburg Hbf, etc. These abrupt state changes are also called *events*. Events occur in a discrete instant of time, as opposed to a continuous state change progressing over a time interval. Accordingly, piecewise constant trajectories model the time evolution of such entities.

Wainer [131] refers to systems with a model representation in which discrete events occur at arbitrary points in time (*continuous time*) as *discrete-event dynamic systems (DEDS)*. In modeling and simulation of DEDS, only a finite number of events is allowed to occur in any given (finite) time interval. There are several mathematical *formalisms* which offer a unifying abstract language to exactly specify models of DEDS. Such formalisms allow to abstract from the concrete implementation in a programming language and algorithms. In this Work, I choose to introduce and employ the *discrete-event system specification (DEVS)* formalism developed by Chow and Zeigler [132] (see also Zeigler, Kim, and Praehofer [130]). There are other formalisms, such as Timed Petri Nets, Timed Finite State Machines, and Event Graphs. [131]

Discrete-event modeling entails the subtle issue of simultaneous events. There are two distinct mechanism which generate simultaneous events in simulations (cf. Ref. [133]):

1. The model intentionally schedules events at the same time, and
2. the finite precision of digital computers.

The latter implies that if a model schedules two events for sufficiently close points in time, the computer represents the distinct times by the same floating-point value. This is a fundamental restriction of simulators implemented on digital computers. The model needs to address the former, intentional, scheduling of simultaneous events. In simulation, finite precision causes simultaneous events which for the computer are indistinguishable from the first kind. Hence, the computer treats both kinds of simultaneous events according to the single mechanism the model defines.

In discrete-event models, changes of the model state occur in instants of time. The duration of such *events* is zero. The separation into discrete and continuous models is conceptually and technically convenient. However, such a distinction is absent in the real world. Moreover, a digital computer only has a finite but large number of states, and hence, finite precision. Thus, each simulation of a continuous model on a digital computer is inherently discrete. Of course, the large number of states lets the simulation still appear continuous in the appropriate frame. An approach to a discrete model is to conceive it as a limit of a series of models in which the event duration approaches zero. However, in general the discrete model with zero event duration exhibits qualitatively different behaviour. [134] To this end, *all discrete-event models are wrong*, but they are still useful as long as these severe limitations are kept in mind.

4.3 DISCRETE-EVENT MODELING WITH DEVS

4.3.1 The DEVS formalism

The DEVS formalism allows to model all discrete-event dynamic systems. [131] Wainer [131] further emphasizes the generality of the formalism, as the reaction to input depends on the time the system spent in the previous state. Moreover, the DEVS formalism features a hierarchical organization of models. Furthermore, the internal states of the components are inaccessible. Both features facilitate encapsulation, leading to more comprehensible models of complex systems. In addition, the DEVS framework features a clear distinction and interface between its generic components and the model-dependent implementation (see Figure 4.1).

In practice, using the DEVS formalism means employing the DEVS specification to mathematically model a DEDS. Indeed, the DEVS specification is the *(Parallel) Discrete Event System Specification (DEVS)*, which allows to model an arbitrary DEDS. [130, 132] A *DEVS model* is a *specific* model that satisfies the generic DEVS specification. A *DEVS simulator* is a generic simulation algorithm that calculates the global dynamics of a DEVS model by interpreting the DEVS specification dynamically (cf. "Concept" row in Figure 4.1). The *DEVS formalism* comprises both the DEVS specification and a DEVS simulator, offering a generic formalism to define and simulate specific DEVS models. A possible software implementation of the DEVS formalism is

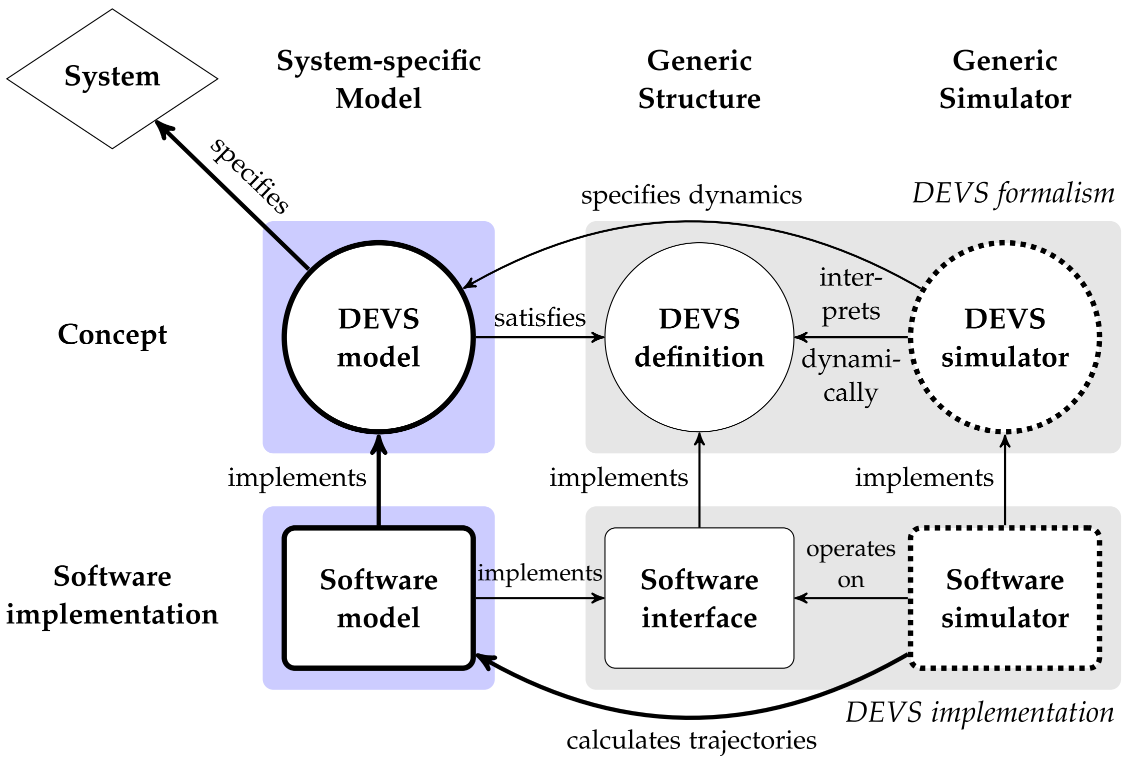

Figure 4.1: The Discrete Event System Specification (DEVS) framework for modeling and simulation after Zeigler, Kim, and Praehofer [130] and Nutaro [133].

to provide a generic interface that implements the DEVS specification. The software implementation of the DEVS model then in turn needs to implement this interface. A generic software implementation of the DEVS simulator simply operates on this interface to compute the trajectories of the model (cf. "Software implementation" row in Figure 4.1).

4.3.2 DEVS specification

The DEVS formalism comprises a state transition mechanism and an output generation mechanism. Each DEVS model has an internal state, according to which it reacts to input trajectories (state transition mechanism). Such a model generates output trajectories, also depending on its internal state (output generation mechanism).

Definition 4.1 (*Parallel discrete event system specification*, or simply *DEVS specification*, [132]). *A* DEVS model *is a tuple* $M = (X, S, Y, ta, \lambda, \delta_{int}, \delta_{ext}, \delta_{con})$ *with*

1. X*: a set of input events* (input set)*,*
2. S*: a set of sequential states* (state set)*,*
3. Y*: a set of output events* (output set)*,*
4. $ta : S \to \overline{\mathbb{R}}_0$*: the* time advancement function*,*
5. $\lambda : S \to Y^b$*: the* output function*, where* Y^b *is the set of all* bags (multisets *[135]) on* Y*,*

6. $\delta_{int} : S \to S$*: the* internal transition function*,*

7. $Q = \{ (s, e) : s \in S, 0 \leqslant e < ta(s) \} \subset S \times \overline{\mathbb{R}}_0$*: the set of* total states*,* e *is the* time elapsed *since the last transition,*

8. $\delta_{ext} : Q \times X^b \to S$*: the* external transition function*, where* X^b *is the set of all* bags (multisets *[135]) on* X*,*

9. $\delta_{con} : S \times X^b \to S$*: the* confluent transition function*.*

For consistency, a DEVS model satisfies (cf. [130, p. 144])

$$\begin{aligned} ta\left(\delta_{ext}(s, e, \emptyset)\right) &= ta(s) - e, \\ \delta_{con}(s, \emptyset) &= \delta_{int}(s). \end{aligned}$$

A model that satisfies this definition works as follows: At any time t, the model is in exactly one state $s \in S$. The model remains in this state for a certain time, the *lifespan* $ta(s)$ given by the time advancement function. At the end of the lifespan, the model generates output. Output may comprise several and even multiple simultaneous output events $y \in Y$. Hence, the output given by the output function is a *bag* (or *multiset*) $\lambda(s) = y^b \in Y^b$. Immediately after the model has generated the output, it changes into the state $\delta_{int}(s)$ given by the internal transition function. Output is only generated at the end of the state lifespan, immediately preceding the internal transition function. The output function does not modify the state. The external transition function δ_{ext} handles (external) input events. As with output, input may comprise several and multiple simultaneous input events $x \in X$ collected in the *input bag* $x^b \in X^b$. Formally, even empty input/output bags $\{\ \} = \emptyset$ are considered valid inputs/outputs. They are distinct from the "non-event" symbolized as $\hat{\varnothing}$. How a DEVS model reacts to input does not only depend on the (partial) state s, but also on the time e elapsed since the last transition. An input x^b to the *total state* (s, e) triggers a transition to the state $\delta_{ext}(s, e, x^b) \in S$.

A DEVS model needs to handle simultaneity at two points: either when it simply receives multiple external events (*simultaneous events*), or when input triggers an external transition at the same time an internal transition is scheduled (*collisions*). After Chow and Zeigler [132], the Parallel DEVS formalism allows the modeler to control the *collision handling*. It transparently implements *parallelism* of collisions and simultaneous events. The DEVS specification provides for not only one input event, but multiple input events as an *input bag* x^b. Hence, the external transition function already equips the modeler to handle simultaneous events in parallel. In addition, the output function λ allows to generate simultaneous output events as an *output bag* y^b. Furthermore, the *confluent transition function* δ_{con} allows to specify how to handle collisions of input events received at times $e = ta(s)$ of internal transitions. For example, a confluent transition function $\delta_{con}(s, x^b) = \delta_{ext}(\delta_{int}(s), 0, x^b)$ implies that internal transitions are handled before external input. Vice versa, a confluent transition function $\delta_{con}(s, x^b) = \delta_{int}(\delta_{ext}(s, e = ta(s), x^b))$ handles external input first. Of course, instead of just serializing the treatment of collisions, one is free to specify an interdependent confluent transition function.

Simultaneity also accounts for a subtle definition of the *total states* that co-determine the external transition function. On the one hand, δ_{con} handles all input events received at times of internal transitions. Hence, the external transition function remains undefined for $e = \text{ta}(s)$. One is also tempted to expect the external transition function never to act on a total state with zero elapsed time $e = 0$. On the other hand, this is possible in coupled models. Immediately after a transition, $e = 0$, another model receiving the output might transit into a transitory state ($\text{ta}(s) = 0$). Hence, it is possible that a model receives input at $e = 0$ although it has just undergone a transition.

The *generic transition function* $\delta : \overline{Q} \times X^b$ of a DEVS model unifies the handling of transitions: [130, 132, 133]

$$\delta\left(s, e, x^b\right) = \begin{cases} \delta_{\text{int}}(s) & \text{if } e = \text{ta}(s), x^b = \emptyset, \\ \delta_{\text{con}}\left(s, x^b\right) & \text{if } e = \text{ta}(s), x^b \neq \emptyset, \\ \delta_{\text{ext}}\left(s, e, x^b\right) & \text{if } 0 \leqslant \text{ta}(s) < e, x^b \neq \emptyset. \end{cases}$$

Here, the *closed set of total states* $\overline{Q} = Q \cup \left\{ (s, e) \in S \times \mathbb{R}_0^+ : e = \text{ta}(s) \right\}$.

A state s with lifespan $\text{ta}(s) = 0$ is called *transitory*, while a state of infinite lifespan $\text{ta}(s) = \infty$ is called *passive*. Transitory states are not interrupted by external events and produce output immediately. Passive states never produce output and state transitions are only triggered by external events. [130]

For some instructive examples of DEVS models from the literature, see Section A.1.

4.3.3 DEVS with ports

The *DEVS with Ports* specification extends the DEVS specification by assigning a specific *input port* or *output port* to each I/O event. In the DEVS formalism, the definition of ports is optional. However, ports increase clarity and conciseness of a model.

Definition 4.2 ([130, p. 84])**.** *Let $M = (X, S, Y, ta, \lambda, \delta_{int}, \delta_{ext}, \delta_{con})$ be a DEVS model. The DEVS model M is a* DEVS model with ports *if the input set X is of form*

$$X = \{ (p, v) : p \in \mathbf{X}, v \in X_p \},$$

and the output set Y is of analogous form

$$Y = \{ (p, v) \mid p \in \mathbf{Y}, v \in Y_p \}.$$

Here, X_p denotes the set of possible events at the input port p, and Y_p denotes the set of possible events at the output port p.

4.3.4 Legitimacy

What happens to a DEVS model if it was stuck in a cycle of transitory states? Or if it traverses a state sequence $(s_n)_n$ with a convergent series $\sum_n \text{ta}(s_n)$ of the state lifespans?

In simulation, the DEVS model would generate an infinite number of transitions in finite time, and get stuck (cf. Thomson's lamp, Achilles and the tortoise [136–138]). This is referred to as *illegitimate* or *Zeno* system behavior, reminiscent of Zeno's paradoxes. [137, 138] On the contrary, a DEVS model is well-defined and called *legitimate* if in any finite time interval, only a finite amount of transitions occur. Equivalently, a DEVS model is legitimate if for any internal state $s_0 \in S$, it takes infinite time to traverse the (infinite) state sequence $(s_n)_n, s_{n+1} = \delta_{\text{int}}(s_n)$:

$$\forall s_0 \in S : \sum_n \text{ta}(s_n) = \infty, s_{n+1} = \delta_{\text{int}}(s_n).$$

Sufficient conditions for a DEVS model to be legitimate are: [130]

1. If the set S of states is finite, every state cycle (periodic orbit of internal states) contains a non-transitory state (necessary and sufficient for finite-state DEVS models):

$$|S| < \infty \Rightarrow$$
$$\forall s_1 \in S : \exists s_k \text{ in } s_1 \to s_2 \to \cdots \to s_n \to s_1 : \text{ta}(s_k) > 0.$$

2. If there is a positive lower bound on the lifespan of each state:

$$\exists b > 0 : \forall s \in S : \text{ta}(s) > b.$$

4.3.5 Network models

One of the appealing features of the DEVS formalism is the possibility to couple existing DEVS models to build new models. Such a *DEVS coupled model* is also called a *network model* of *component* DEVS models. In fact, it is the components of a network model that are solely responsible for processing input to the network and producing its output. The network model itself only specifies the input/output coupling of the components. The network and its components only communicate via input and output. Indeed, the internal state of a component is shielded from direct access or influence by the other components and the network. A component of a network model is either a DEVS model (also referred to as *atomic model*), or another network model.

Like an atomic model, a network model has an internal state, a state transition function, and generates an output trajectory from an input trajectory. In fact, a network model is indistinguishable from an equivalent atomic model that produces the same output trajectory as the network model, for all input trajectories. This atomic model is also referred to as the *resultant* of the network model.

A network model specifies how its components handle input to the network, and how its components generate output of the network. Input to each component is either input to the network model, or output from other components. Output of the network model stems from output of its components.

This presentation of network models also follows the presentation of Zeigler, Kim, and Praehofer [130] and Nutaro [133].

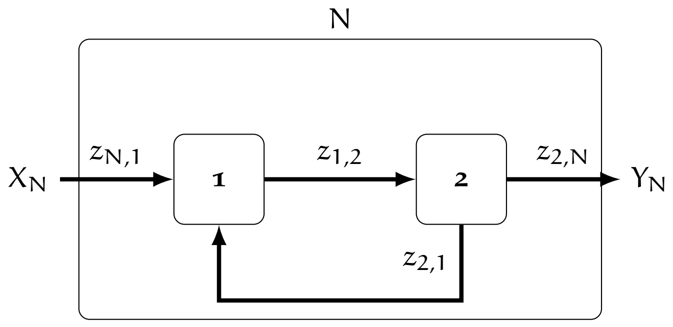

Figure 4.2: An example DEVS network model with two components [133, Figure 3.4].

Definition 4.3 (*Network model* [133])**.** *A* network model N *is a tuple*

$$N = (X_N, Y_N, D, \mathcal{I}, Z),$$

with

1. *the set of input events* X_N *to the network model,*
2. *the set of output events* Y_N *of the network model,*
3. *the* set of components D *of the network model, i.e. each element* $d \in D$ *is an atomic model or a network model,*
4. *the family* $\mathcal{I} = \{ I_d \subset D \cup \{ N \} \setminus \{ d \} : d \in D \cup \{ N \} \}$ *of sets* I_d *of* influencers *of a component* d *(or the network itself), where a component (or the network) is not to directly couple to itself,*
5. *the set of coupling functions* $Z = \{ z_{d',d} : d' \in I_d, d \in D \cup \{ N \} \}$*, where a coupling function couples output of a component* $d' \in I_d$ *to input of a component* $d \in D$ *output to component input,* $z_{d',d} : Y^b_{d'} \to X^b_d$*, or network input to component input,* $z_{N,d} : X^b_N \to X^b_d$*, or component output to network output,* $z_{d',N} : Y^b_{d'} \to Y^b_N$*.*

The output of the components $d \in I_N$ determines the network output via the respective coupling function $z_{d,N}$ (cf. Figure 4.2). See Chapter 2.3 of Ref. [131] for an instructive "Hello world" coupled DEVS model, and Section A.2 of this Thesis for another example.

4.4 DEVS SIMULATION

4.4.1 Time base, trajectories, and segments

Simulating a DEVS model requires a notion of time. Usually, time instants are a real number, and the passage of time means moving along the real time axis $\mathbb{R}$. However, transitory states in discrete-event systems require a notion of events happening at the same time, but still one after the other (not simultaneously). Thus, following Nutaro [133], the *time base* $\mathbb{T}$ for discrete-event simulation includes a discrete event counter $c \in \mathbb{N}_0$:

$$\mathbb{T} = \mathbb{R}^+_0 \times \mathbb{N}_0.$$

The total order of $\mathbb{R}$ and $\mathbb{N}_0$ induces a total order on the Cartesian product $\mathbb{T}$, the lexicographical order:

$$\tau_1 = (t_1, c_1) < \tau_2 = (t_2, c_2) \Leftrightarrow t_1 < t_2 \vee (t_1 = t_2 \wedge c_1 < c_2).$$

The total order on $\mathbb{T}$ allows to keep the notion of (open, connected) time intervals $(\tau_1 = (t_1, c_1), \tau_2 = (t_2, c_2))$, with closed and half-open intervals defined accordingly. The definition of a convenient *time advance operator* $\rhd$ is

$$\tau \rhd \Delta\tau = (t, c) \rhd (\Delta t, \Delta c) \equiv \begin{cases} (t + \Delta t, 0) & \text{if } \Delta t > 0, \\ (t, c + \Delta c) & \text{if } \Delta t = 0, \Delta c \geqslant 0. \end{cases}$$

A *trajectory* is any function $z : \mathbb{T} \to A$ that assigns a value $a(t) \in A$ for each point in time $t \in \mathbb{T}$. The *input trajectory* of a DEVS model with input set X is $x : \mathbb{T} \to X^b \cup \{\hat{\varnothing}\}$, where $\hat{\varnothing}$ is the *non-event*: $x(t) = \hat{\varnothing}$ represents the absence of input at the given point in time t. Similarly, the *output trajectory* $y : \mathbb{T} \to Y^b \cup \{\hat{\varnothing}\}$, the *state trajectory* $s : \mathbb{T} \to S$ and the *total state trajectory* $q : \mathbb{T} \to \overline{Q}$.

A *segment* $z_{[\tau_1,\tau_2)}$ of a trajectory $z : \mathbb{T} \to A$ is the restriction $z_{[\tau_1,\tau_2)} \equiv z\big|_{[\tau_1,\tau_2)}$ to the interval $[\tau_1, \tau_2)$. The *length* $l(z_{[\tau_1,\tau_2)})$ of a segment $[\tau_1, \tau_2)$ is the duration of the continuous time:

$$l(z_{[\tau_1,\tau_2)}) = t_2 - t_1.$$

Two segments $z_{[\tau_1,\tau_2)}, z_{[\tau_3,\tau_4)}$ of a trajectory $z : \mathbb{T} \to A$ are called *contiguous* if and only if $\tau_2 = \tau_3$. The *concatenation operator* $\cdot$ concatenates two contiguous segments $z_{[\tau_0,\tau_1)}, z_{[\tau_1,\tau_2)}$ in the obvious way:

$$z_{[\tau_0,\tau_2)} \equiv z_{[\tau_0,\tau_1)} \cdot z_{[\tau_1,\tau_2)}.$$

A segment $z_{[\tau_0,\tau_n)} : [\tau_0, \tau_n) \to A (n \in \mathbb{N}_0)$ is called *piecewise constant* if and only if it is a concatenation of a finite number of segments, each of which is a constant function:

$$\exists \tau_1 < \ldots < \tau_{n-1} \in (\tau_0, \tau_n); a_1, \ldots, a_n \in A :$$
$$z_{[\tau_0,\tau_n)} = z_{[\tau_0,\tau_1)} \cdot z_{[\tau_1,\tau_2)} \cdots z_{[\tau_{n-1},\tau_n)}, z_{[\tau_{i-1},\tau_i)}(\tau) = a_i \forall \tau \in [\tau_{i-1}, \tau_i).$$

A *piecewise constant trajectory* is a trajectory which has only piecewise constant segments (cf. state trajectories). A segment $z_{[\tau_0,\tau_1)} : [\tau_0, \tau_1) \to A \cup \{\hat{\varnothing}\}$ is called a *primitive event segment* iff

$$z_{[\tau_0,\tau_1)}(\tau) = \begin{cases} a \in A \cup \{\hat{\varnothing}\} & \text{if } \tau = \tau_0, \\ \hat{\varnothing} & \text{otherwise.} \end{cases}$$

A primitive event segment either has exactly one event at the beginning of the trajectory, or no event at all. A segment $z_{[\tau_0,\tau_n)} : [\tau_0, \tau_n) \to A \cup \{\hat{\varnothing}\} (n \in \mathbb{N}_0)$ is called an *event segment* if and only if it is a concatenation of a finite number of primitive event segments:

$$\exists \tau_1 < \ldots < \tau_{n-1} \in [\tau_0, \tau_n); a_1, \ldots, a_n \in A :$$
$$z_{[\tau_0,\tau_n)}(\tau) = \begin{cases} a_i & \text{if } \tau = \tau_i, i \in \{1, \ldots, n-1\}, \\ \hat{\varnothing} & \text{otherwise.} \end{cases}$$

Accordingly, any event segment $z_{[\tau_0,\tau_n)}$ can be decomposed into a chain of contiguous primitive event segments z_i:

$$\begin{aligned}z_{[\tau_0,\tau_n)} &= z_0 \cdot \cdots \cdot z_n,\\ &\quad z_i : [\tau_i, \tau_{i+1}) \to A \cup \{\hat{\varnothing}\}, z_0 = [\tau_0, \tau_1) \to \{\hat{\varnothing}\},\end{aligned}$$

where the initial segment z_0 does not contain events. All subsequent primitive segments do have one event, which is at the beginning. If $\tau_1 = \tau_0$, the initial segment has an empty domain. An *event trajectory* is a trajectory which has only event segments (cf. input/output trajectories). In particular, we demand that any input trajectory to a DEVS model is an event trajectory. The following ensures piecewise constant state trajectories and event output trajectories.

Theorem 4.4 ([133]). *Let M be a legitimate DEVS model. If the input trajectory of M is an event trajectory, the state trajectory is piecewise constanct and the output trajectory is an event trajectory.*

4.4.2 System state transition function

This Subsection describes the global dynamics of a DEVS model by the *state transition function* Δ. Specifically, let M be a legitimate DEVS model in the total state $(s, e) \in \overline{Q}$ at time τ_0. Let M be subject to the input trajectory segment $x : [\tau_0, \tau_f) \to X^b \cup \{\hat{\varnothing}\}$. Then it transits to a new (total) state $(s', e') = \Delta((s, e), x)$ at time τ_f. In other words, Δ is the time evolution operator of the DEVS model under external input.

In order to be consistent, the state transition function Δ needs to have the following properties: [133] composition and concatenation need to commute, and Δ needs to be invariant under empty input:

$$\begin{aligned}\Delta((s, e), x_1 \cdot x_2) &= \Delta(\Delta((s, e), x_1), x_2),\\ \Delta\left((s, e), x|_{[\tau,\tau)}\right) &= (s, e)\end{aligned}$$

Furthermore, DEVS models are *time-invariant*. [133] That means, the response of a DEVS model to an input does not depend explicitly on the time of input. Time-dependent models can incorporate time as an internal state variable to comply with this assumption.

The state trajectory s and total state trajectory q are determined by the input trajectory x and the initial total state (s_0, e_0) at time $\tau_0 = (0, 0)$:

$$q(\tau) = (s(\tau), e(\tau)) = \Delta\left((s_0, e_0), x|_{[(0,0),\tau)}\right).$$

The decomposition of the input trajectory segment x into contiguous primitive segments is

$$x = x_0 \cdot \cdots \cdot x_n,$$

with input events at times $\tau_1, \ldots, \tau_n$. Therefore, the total state (s', e') at time τ_f is

$$\begin{aligned}(s', e') &= \Delta((s, e), x)\\ &= \Delta((s, e), x_0 \cdot \cdots \cdot x_n)\\ &= \boldsymbol{\delta}(\boldsymbol{\delta}(\cdots \boldsymbol{\delta}(\boldsymbol{\delta}((s, e), x_0), x_1) \cdots), x_n),\end{aligned}$$

where $\boldsymbol{\delta}$ is the state transition function for a primitive input segment. Hence, the response of a DEVS model to an input trajectory segment is given by recursively applying the state transition function $\boldsymbol{\delta}$ to successive primitive input segments. [133]

Let the DEVS model be in an initial total state $(s, e) \in \overline{Q}$ at time τ_0. Let the DEVS model be subject to the primitive input event segment $x : [\tau_0, \tau_1) \to X^b \cup \{\hat{\varnothing}\}$ at that time. Then, the state transition function $\boldsymbol{\delta}$ maps the initial total state to the total state at time τ_1 at the end of that input segment.

Now, the state transition functions connects all the possible transitions and input events of a DEVS model to produce dynamics:

(I) $\boldsymbol{\delta}((s,e),x) = (s, e + l(x))$ for $\tau_1 \leqslant \tau_0 \vee x(\tau_0) = \hat{\varnothing}, \mathrm{ta}(s) - e > l(x)$,

(II) $\boldsymbol{\delta}((s,e),x) = \boldsymbol{\delta}\left((s, \mathrm{ta}(s)), x|_{[\tau_0 \rhd (\mathrm{ta}(s)-e,0), \tau_1)}\right)$ for $\tau_1 > \tau_0, x(\tau_0) = \hat{\varnothing}$, $e < \mathrm{ta}(s) \leqslant e + l(x)$,

(III) $\boldsymbol{\delta}((s,e),x) = \boldsymbol{\delta}\left((\delta(s, e, x(\tau_0)), 0), x|_{[\tau_0 \rhd (0,1), \tau_1)}\right)$ for $\tau_0 < \tau_1, e = \mathrm{ta}(s) \vee x(\tau_0) \neq \hat{\varnothing}$

Accordingly, the state transition function differentiates three cases:

1. (I) The primitive input event segment x neither contains input nor reaches an internal transition.
2. (II) The primitive input event segment x does not contain input, but spans across an internal transition.
3. (III) The primitive input event segment x contains input, or an internal transition is imminent.

In the absence of input and transitions (I), the state transition function $\boldsymbol{\delta}$ simply increases the elapsed time of the total state by the length of the event segment. The length of the event segment can also be zero. The other cases treat $\mathrm{ta}(s) = e + l(x)$. For an internal transition within the event segment (II), the state transition function $\boldsymbol{\delta}$ fast-forwards recursively to the time of the internal transition. (Primitive event segments provide input only at the beginning—if any.) For an imminent transition (III), the state transition function $\boldsymbol{\delta}$ applies the generic transition function δ and advances the discrete time count by 1.

Inputs may arrive at discrete times $c > 0$. For example, if an input arrives at time $(0, 0)$, and another input arrives at the same real time, but at a later discrete time, $(0, 2)$, there are two non-simultaneous inputs at different points in time! In the formalism, this leads to two primitive event segments x_0 on $[(0, 0), (0, 2))$ and x_1 starting at $(0, 2)$. Case (I) perfectly deals with the situation after the external transition: at time $(0, 1)$, case (I) applies (if $\mathrm{ta}(s) > 0$), and $\boldsymbol{\delta}$ simply advances the elapsed time by $l(x_0) = 0$, fast-forwarding to the next input time $(0, 2)$.

4.4.3 System output function

The *total output function* is defined on all total states. It yields the output $\lambda(s)$ immediately preceding an internal transition,

$$\Lambda : \overline{Q} \to Y^b \cup \{\hat{\varnothing}\}, (s,e) \mapsto \begin{cases} \lambda(s) & \text{if } e = \mathrm{ta}(s), \\ \hat{\varnothing} & \text{otherwise.} \end{cases}$$

Hence, the output trajectory of a DEVS model is

$$\begin{aligned} y &= \Lambda \circ s, \\ y(\tau) &= \Lambda(s(\tau)). \end{aligned}$$

Due to the definition of the state transition function, there is a minimal time delay between input and output, which is $(0,1)$. For example, a DEVS model that receives input at time τ_0, will have changed its state only at time $\tau_0 \rhd (0,1)$ (cf. case III in the state transition function δ). The same argument holds for internal transitions which are due but not carried out yet at time τ_0. In essence, state transitions actually happen in-between the discrete times $\tau_0, \tau_0 \rhd (0,1)$. It is precisely this notion that allows to loosely speak of the DEVS model being in two states at the same time t: the DEVS model is still in the initial state at time τ_0, but changed to the next state by time $\tau_0 \rhd (0,1)$. Both discrete points in time are at the same point in real time $t \in \mathbb{R}_0^+$.

See Section A.3 for an example of a table-based time evolution (simulation) of a DEVS model.

4.4.4 Reducing a network model to its resultant atomic model

Still missing for a dynamical interpretation of DEVS models is a dynamical interpretation of a network model. It is a hallmark of DEVS that for any given network model, there is a representation as an atomic DEVS model. This is referred to as *closure under coupling*. There is a constructive procedure to reduce a network model N to an atomic model M_N, called the *resultant model*. The procedure is detailed in References [130, 133] as well as in Section A.4. Conveniently, the construction of the resultant yields a simulation algorithm for a network model.

The coupled DEVS specification allows to successively nest network models. Following Nutaro [133], each instance of a component is to belong to at most one instance of a network model.

Indeed, every network model is the root of a tree-shaped hierarchy of DEVS models, with atomic models as leaves and network models as internal vertices. Since a network encapsulate the inner working of its components, every model communicates only with its parent network model, and its child nodes. Closure under coupling means that to collapse any subtree within this hierarchy is to replace the subtree by the resultant of its root network model.

Part III

Demand-Driven Directed Transport Systems

ABSTRACT

Understanding the nonlinear dynamics, scaling behavior, critical transitions and other emergent properties of collective mobility and demand-driven transport systems is crucial to optimize system efficiency and individual service quality. As discrete events such as pick-ups and deliveries govern their time evolution, it is not straightforward to apply standard analytical and computational methods from statistical physics, nonlinear dynamics and network science, as discrete-event systems typically feature a high-dimensional, non-smooth and technical state space. Here, I outline a unifying theoretical and computational framework to efficiently model, simulate, and assess the performance of demand-driven directed transport (D3T) systems. By proposing a formal language (D3TS) based on the Discrete-Event System Specification (DEVS), I facilitate modelling and performance analysis of mobility and transport systems across disciplinary domains. Building on this formal language, the D3T Python package (pyd3t) allows researchers to easily assemble modules in a toolbox-like fashion. By providing a number of topologies, with e. g. $\mathbb{R}^2$, and any kind of network among them, and a variety of basic dispatching rules, the user may combine them to effortlessly build a deterministic or stochastic spatio-temporal mobility or transport simulation experiment. Furthermore, the user is free to specify her own modules. An analysis module provides ample statistics to aggregate the dynamics and assess the performance of the simulated instance, while a visualization module enables visual inspection of the simulated transport dynamics. I envision both the formal language of D3TS and the computational implementation of pyd3t to facilitate studying collective mobility and demand-driven transport and developing the necessary tools to do so in a reproducible manner.

5 INTRODUCTION

Studying and designing mobility transport systems is a multidisciplinary endeavor. The quest to optimize transportation of goods and people in facilities, cities, regions, or whole nations, unites engineers, operation researchers, mathematicians, computer scientists, urban and regional planners, policymakers and physicists. They all bring their expertise and disciplinary language to the table in the on-going extensive efforts to understand distributed, collective systems. These complex systems under study are themselves as diverse as the disciplines involved, including autonomous production and distribution logistics, urban courier services, and demand-responsive transport in human transit. [34, 38, 139–142] In general, models of these systems are analytically intractable, and the optimization problems involved are NP-hard and dynamical, [143] such that the method of choice to analyze these systems is discrete-event simulation. Seemingly, what the field has been lacking so far, is a common high-level language: a language that allows to state the given transportation model and the optimization problem at hand in an accessible but mathematically exact form, and at the same time immediately translates it into an executable simulation model. This way, attention focusses on the dynamics, rather than on a purely structural optimization problem description. On the other hand, such a language would add structure and reproducibility to simulation studies hitherto seemingly conducted in an ad-hoc fashion.

This Part of this Thesis is a contribution towards such a cross-disciplinary language and a computational implementation. In the following, I outline a unifying theoretical and computational framework to efficiently model, simulate, and assess the performance of demand-driven directed transport (D3T) systems. Chapter 6 introduces the Demand-Driven Directed Transport System Specification (D3TS), based on the Discrete-Event System Specification (DEVS). This Chapter has been partially published as a peer-reviewed IEEE conference proceeding (Sorge, Manik, Herminghaus, and Timme [144]) and is based on an extensive unpublished technical report of the full D3T Specification (Sorge [52]). Section 6.5 outlines the computational implementation of the formal D3TS language as the Python package pyd3t. The software has been partially introduced in a contributed talk at the recent DPG Spring Meeting (Sorge, Timme, and Manik [53]).[1] Finally, Chapter 7 concludes this Part and sketches avenues of further work.

1 While the abstract was mainly authored by the first author, the actual talk was prepared and presented by my co-author and pyd3t co-lead developer Debsankha Manik.

6 MODELLING AND SIMULATING D3T

6.1 THE D3T FRAMEWORK

6.1.1 Overview

The Demand-Driven Directed Transport framework is a framework for modelling transportation systems that serve transport requests of discrete immotile loads in a physical transport space. In the system, transporters process requests by travelling along paths of subsequent origins and destinations and transporting the loads. Let me restate the characteristics of a D3T model here (cf. Section 1.2):

- Transportation is *demand-driven*: there are no external fields such as gravity or an electric field driving transport as in other systems studied in physics (e. g. charge transport in solids). (Also, there are no fixed schedules.)
- Transportation is *on-demand* or "urgent" in the sense that typical requests are placed and require to be served within a time window which is of the same order of magnitude as the travel time.
- Transportation is *directed*, as opposed to diffusive transport, or e. g. the unspecific distribution of nutrients by the cardiovascular system. (Also, there are no changes of transporters or modes.)
- Transported objects are *discrete* loads: there is no flow, there is no continuous quantity such as water.
- The transported loads are *immotile*: they do not move on their own, as opposed to conduction electrons in a metal.
- Loads are transported by *discrete transporter units*: there is no conveyer belt, or pipes, such as the Internet distributes data packets on a continuous basis.
- Transporters are costly: their number is of the order of the average number of requests to be served within the average time it takes to serve a request. In particular, not every load has its own transporter.

This framework provides a unifying mathematical language to model D3T systems and assess their performance. Typical performance measures of interest are the throughput capacity (the maximum long-term average rate of requests servable without overloading the system), the utilization of transporters (system efficiency), and individual waiting and travel times (individual service quality). Formally, the D3T framework is a subset of the mighty, general *Parallel Discrete Event-System Specification (P-DEVS)* language. [132]

(As before, and in the following, when we speak of DEVS models, we refer to P-DEVS models.) This is a crucial design decision. In summary, the (parallel) DEVS formalism features the following properties: [132]

1. Collision handling: the modeler decides how to handle collisions of external and internal events (by the implementation of a confluent transition function).
2. Parallelism: simultaneity (both collisions and simultaneous events) is integral in the formalism (confluent transition function, multisets of input and output events).
3. Closure under coupling: the behavior of each network model comprising coupled DEVS models is described by its resultant, which is an atomic DEVS model itself.
4. Hierarchical consistency: the behavior of a nested network model is independent of the specific implementation of a hierarchy (the modeler is free to organize the structure of the nested network models as she sees appropriate, as long as the functional coupling of the components is preserved).

A crucial feature of the D3T framework is its strict *modularity*: it specifies the abstract component classes and the protocol for interactions of component modules. A D3T modeller may either implement her own component modules, or choose from a set of pre-defined (and programmed) modules to specify a D3T simulation.

In fact, I aim to provide a solid technical foundation for D3T modelling and simulation such that no further modelling and programming is needed to build a whole D3T model and run a full-scale D3T simulation.

6.1.2 Phenomenological D3T dynamics

In D3T systems, vehicles transport discrete loads (or passengers) in the transport space from their respective origin to their respective destination. These loads arrive to the system according to a stochastic birth process, with their origin and destination determined and fixed at arrival time (see Figure 6.1 for a simple example). A D3T model is a *dynamic* model in the sense of a dynamic optimization problem: not all information is available, but only reveils in the course of the evolution of the system: a load arrival is only known at the arrival time, and not before.

Load arrivals are independent of load transport. While load arrivals are stochastic, load transport is deterministic. Load arrivals are the only external events influencing the transportation dynamics: the load arrival process feeds into the deterministic transportation dynamics.

A transporter picks up a load at its origin, and delivers a load at its destination. Once picked up, loads remain on-board the transporter until it reaches the destination of the load. Loads do not change transporters.

Which of the transporters picks up a given load, is subject to the dispatching policy. The D3T framework allows the dispatching policy to reject load requests. If rejected, a load leaves the transport system immediately.

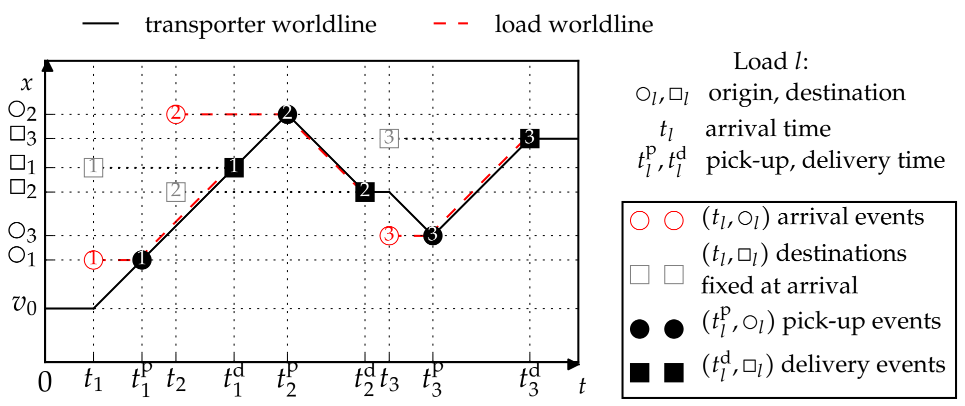
transporter worldline
load worldline
x
$\bigcirc_2$
$\square_3$
$\square_1$
$\square_2$
$\bigcirc_3$
$\bigcirc_1$
v_0
0
t_1
t_1^p
t_2
t_1^d
t_2^p
t_2^d
t_3
t_3^p
t_3^d
t
Load l:
$\bigcirc_l, \square_l$ origin, destination
t_l arrival time
t_l^p, t_l^d pick-up, delivery time
$(t_l, \bigcirc_l)$ arrival events
$(t_l, \square_l)$ destinations fixed at arrival
$(t_l^p, \bigcirc_l)$ pick-up events
$(t_l^d, \square_l)$ delivery events

Figure 6.1: A trajectory of a simple D3T model with only 1 transporter and 3 transport requests $(l, \bigcirc_l, \square_l)$. The transporter is a myopic taxi that processes transport requests on the real line $\mathbb{R}$ on a first-come-first-served basis, one at a time. If all known requests have been served, the transporter remains idle at its current position. The taxi also needs to travel empty to pick-up a new load l at its origin $\bigcirc_l$, and deliver it at its destination $\square_l$. The Figure shows origin and destination at the arrival time t_l. The point $(t_l^p, \bullet)$ in space-time marks the pick-up event at the load origin, and the point $(t_l^d, \blacksquare)$ in space-time marks the delivery event at the load destination.

The travel time between any two points of the transport space is the metric distance. In geodesic geometries, transporters travel at unit speed along geodesic segments, i.e. shortest paths between subsequent waypoints. In networks, transporters jump along the nodes of the shortest path, where the jump duration is the arc weight.

Even though loads arrive according to a continuous-time stochastic process, and transporters move continuously in geodesic geometries, it is discrete-time events governing the transportation dynamics. Transporters pick-up and deliver loads, depart and arrive at positions, at intrinsically discrete instants of time.

6.1.3 The transport space

Demand-driven directed transport (D3T) takes place in a physical space. The mathematical model of such a *transport space* is a hemimetric space $\mathcal{M}$ with a hemimetric d. [145] In particular, a hemimetric is a metric that does not need to be neither discernible nor symmetric. Additionally, the transport space $\mathcal{M}$ shall be either "continuous" (geodesic), or discrete (a network).

A *geodesic hemimetric space* $\mathcal{M}$ is a hemimetric space for which any two distinct points x, y are connected by a *geodesic directed segment* (or shortest path) from x to y. For example, the Euclidian space $\mathbb{R}^n$ with the standard metric $d(x, y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$ is a geodesic metric space for all $n \in \mathbb{N}$.

While geodesic metric spaces allow for continuous movements along geodesic segments, networks are discrete metric spaces which only allow jumps along *paths* of links (*arcs*) between their elements (*nodes* or *vertices*). A *network* $G = (V, E, \omega)$ is a weighted digraph which is simple and strongly connected. The network is *undirected*, if $\forall e = (v, v') \in E : e' = (v', v) \in$

$E, \omega(e') = \omega(e)$. A network $G = (V, E, \omega)$ is endowed with the hemimetric of the shortest-path length, yielding a hemimetric space.

See Section B.1 for a detailed specification of the transport spaces.

6.1.4 Transport requests and loads

Each transport request $(l, \bigcirc_l, \square_l)$ defines a point-like material entity: a *load*. A load l is immotile and demands active transport from its *origin* $\bigcirc_l \in \mathcal{M}$ to its *destination* $\square_l \in \mathcal{M}$ in the transport space $\mathcal{M}$. At time of arrival, a load requests transport immediately, The *load index* $l \in \mathbb{N}$ enumerates the transport requests in order of arrival. Transport requests arrive to a D3T system according to a stochastic process, and precisely, according to a marked point process $\{(T_n, Y_n), n \in \mathbb{N}\}$. The random variable T_n is the arrival epoch of the n-th load. If $T_n = \infty$ for some n, no more loads arrive. The random variable Y_n denotes the *mark* $(\bigcirc_n, \square_n, \sigma_n)$ of the n-th load, from the augmented *mark space* $\mathcal{M} \times \mathcal{M} \times \Sigma \cup \{\nabla\}$. The elements of Σ are additional marks that the dispatching policy and/or performance analysis may take into account. For example, a D3T model may specify a mark for priority customers. Note that $Y_n = \nabla$ if and only if $T_n = \infty$.

6.1.5 Transporters

GENERAL PROPERTIES A transporter is a point-like motile object in the transport space $\mathcal{M}$. Its purpose is to actively transport immotile loads. Each transporter i is endowed with an integer (or infinite) *capacity* C_i, which is the maximum number of loads it can transport at the same time. A D3T model has a fixed number N of transporters which might differ in their capacity but are otherwise identical.

In the D3T framework, transporters are actively transporting the loads, but nevertheless they are myopic passive agents that merely execute the dispatching policy. A transporter typically lacks any but short-term knowledge about its own queue of loads to transport and positions to travel to. Furthermore, transporters do not interact with other objects in the transport space, i.e. loads or other transporters. Transporters are also ideal in the sense that they are always in service, and they do not need to refuel or the like, unless explicitly told to do so. A transporter either moves at unit velocity, or remains at its current position $v \in \mathcal{M}$. In geodesic spaces, transporters move continuously along geodesic segments. In networks, transporters perform discrete jumps along nodes of shortest paths, where each jump takes a time given by the weight of the respective arc.

JOBS This framework describes transporter operation as a sequence of jobs. Each job $j = (\mathbb{P}, \tilde{v}, \mathbb{D})$ is a tuple of

1. a *pick-up set* $\mathbb{P}$ of loads to pick up at the current location v of the transporter,
2. the scheduled destination $\tilde{v}$ to travel to,
3. a *delivery set* $\mathbb{D}$ of loads to deliver at the new location $\tilde{v}$.

At any time t, a transporter either has a current transporter job to process, or it is idle. The set $\tilde{\mathbb{P}}$ denotes the *loads scheduled for pick-up* at the current position v, the position $\tilde{v} \in \mathcal{M}$ the destination to travel to, and the set $\tilde{\mathbb{D}}$ denotes the *loads scheduled for delivery* at the destination $\tilde{v}$.

PICKING UP AND DELIVERING LOADS Picking up and delivering loads takes the transporter a certain amount of time. The pick-up period and delivery period may depend on the position $v \in \mathcal{M}$ and on the set of loads to pick-up ($\mathbb{P}$) or deliver ($\mathbb{D}$). The period does not depend on the particular transporter. The *pick-up period function* (*delivery period function*) $\tau_P(v, \mathbb{P})$ ($\tau_D(v, \mathbb{D})$) determines the time it takes a transporter to pick up (deliver) loads $\mathbb{P}$ ($\mathbb{D}$) at the position v.

QUEUE Each transporter has a queue Q of transporter jobs to process. Formally, at any time t the queue $Q = \left(\tilde{j}_n\right)_n$ is a finite sequence of transporter jobs $\tilde{j}_n = \left(\tilde{\mathbb{P}}, \tilde{v}, \tilde{\mathbb{D}}\right)$ scheduled for the transporter to process sequentially in ascending order. The job that the transporter currently processes is not part of the queue (any more).

DISPATCHER ACTION As stated before, the transporters merely execute the dispatching policy. In practice, they receive commands from the dispatching unit (called *dispatcher*). In the D3T framework, there are the following dispatcher actions a transporter willingly accepts at any time:

1. appending a sequence of jobs J to the transporter queue $Q \mapsto (Q, J)$ (*submit*),
2. replacing the queue $Q \mapsto J$ with a new sequence of jobs J (*submit and replace*)
3. emptying the queue $Q \mapsto \emptyset$ (*submit and replace empty job sequence*),
4. canceling the current job $c \mapsto 1$ (*cancel*). This implies also emptying the queue ($Q \mapsto \emptyset$) and makes the most sense if the dispatcher submits a new sequence of jobs at the same time.

Note that the first 3 actions only modify the queue, but do not affect the current job. The last action is the most intrusive, regarding the ongoing operation of the transporter.

TRANSPORTER STATES AND STATE TRANSITIONS At any time t, the state of a transporter is given by the variables in Table 6.2. A transporter experiences the following state transitions (see Table 6.3 and Figure 6.4): A transporter starts a job ($*$), when it has an non-empty queue ($Q \neq \emptyset$) and either has just ended a job ($\dagger$) or has been idle (I). Starting a job, a transporter transits to the pick-up phase (P) when the set of loads scheduled for pick-up is non-empty ($\tilde{\mathbb{P}} \neq \emptyset$). After pick-up, the transporter starts moving (T), unless it has been ordered to cancel (c). If there are not any loads to pick up ($\tilde{\mathbb{P}} = \emptyset$), the transporter directly transits to the travel phase (T). After travelling, the transporter transits to the delivery phase (D), unless there are not any loads to deliver ($\tilde{\mathbb{D}} = \emptyset$), or it has been ordered to cancel ($c = 1$). A transporter

Table 6.2: Transporer state variables in D3T models.

Symbol	State variable
$v \in \mathcal{M}$	Current position
$w \in \mathbb{R}_0^+$	Waiting time (remaining jump time) to arrive at $\tilde{v}$
$\mathbb{L}$	Current cargo (set of loads on-board the transporter)
Q	Current queue of transporter jobs
$\tilde{\mathbb{P}}$	Current job: Set of loads scheduled for pick-up at current position
$\tilde{v} \in \mathcal{M}$	Current job: Destination
$\tilde{\mathbb{D}}$	Current job: Set of loads scheduled for delivery at destination
M	Current *mode*, see Table 6.3.
$c \in \{0, 1\}$	A Boolean variable that indicates whether the dispatcher ordered the transporter to cancel its current job.

Table 6.3: Transporter modes in D3T models. We introduce the pseudo-modes for the start of processing the current job ($*$) and end of processing the current job (†). The *lifetime* of a mode is the period a transporter spends in that mode before it advances to the next mode. The *idle* mode is only left at external input from the dispatcher, when it adds jobs to the transporter queue. At the end of a mode, the transporter state autonomously changes according to the *effects* column.

Symbol	Mode	Lifetime	Entry condition	Effect
I	idle	∞	$Q = \emptyset \wedge w = 0 \wedge \tilde{v} = v \wedge \tilde{\mathbb{P}} = \tilde{\mathbb{D}} = \emptyset$	
$*$	(job start)	0	$Q \neq \emptyset$	$(\tilde{\mathbb{P}}, \tilde{v}, \tilde{\mathbb{D}}) \mapsto \tilde{j}_1, Q \mapsto (\tilde{j}_n)_{n=2}^{\lvert Q \rvert}$
P	pick-up	$\tau_P(v, \tilde{\mathbb{P}})$	$\tilde{\mathbb{P}} \neq \emptyset \wedge \neg c$	$\mathbb{L} \mapsto \mathbb{L} \cup \tilde{\mathbb{P}}, \tilde{\mathbb{P}} \mapsto \emptyset$
T	travel	$d(v, \tilde{v})$	$\tilde{\mathbb{P}} = \emptyset \wedge \tilde{v} \neq v \wedge \neg c$	$v \mapsto \tilde{v}$
D	delivery	$\tau_D(\tilde{v}, \tilde{\mathbb{D}})$	$\tilde{\mathbb{P}} = \emptyset \wedge \tilde{v} = v \wedge w = 0 \wedge \tilde{\mathbb{D}} \neq \emptyset \wedge \neg c$	$\mathbb{L} \mapsto \mathbb{L} \setminus \tilde{\mathbb{D}}, \tilde{\mathbb{D}} \mapsto \emptyset$
†	(job end)	0	$(\tilde{\mathbb{P}} = \emptyset \wedge \tilde{v} = v \wedge w = 0 \wedge \tilde{\mathbb{D}} = \emptyset) \vee c$	$c \mapsto 0$

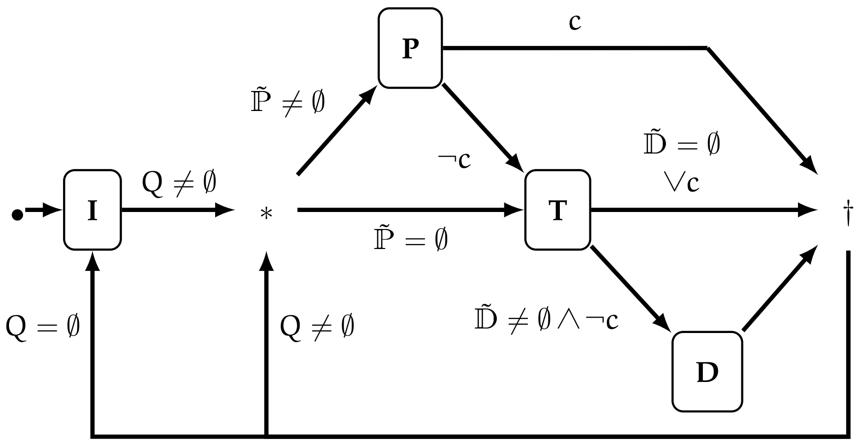
c
P
$\tilde{\mathbb{P}} \neq \emptyset$
$\neg c$
$\tilde{\mathbb{D}} = \emptyset \vee c$
I
$Q \neq \emptyset$
$*$
T
†
$\tilde{\mathbb{P}} = \emptyset$
$Q = \emptyset$
$Q \neq \emptyset$
$\tilde{\mathbb{D}} \neq \emptyset \wedge \neg c$
D

Figure 6.4: Transition diagram of a transporter in a D3T model. See Table 6.2 for the state variables and see Table 6.3 for the modes. Labels at the transition arcs are the entry condition for the successive mode.

ends a job (†) after delivery, unless there have not been any loads scheduled for delivery, or the transporter was ordered to cancel. If the queue is empty ($Q = \emptyset$) after ending a job, the transporter becomes idle.

When the dispatcher orders to cancel a job, the transporter queue is emptied. Furthermore,

1. if the transporter is picking up loads (**P**), it regularly finishes pick-up and, prematurely, ends the current job thereafter;
2. if the transporter is travelling (**T**), it stops as soon as possible (in a geodesic geometry this means immediately, in a network this means when the current jump to the next node is completed);
3. if the transporter is delivering loads (**D**), it regularly finishes delivery and ends the current job thereafter.

6.1.6 Dispatching policy

The *dispatching policy* $\mathfrak{D}$ is a set of rules that determine the instructions for the transporters given the load arrivals up to the current time. The dispatching policy covers each load arrival: it either rejects a load request, or assigns the load to a transporter (at least eventually). It is the dispatching policy that governs the transporter queues. In the light of new load arrivals, the dispatching policy may modify transporter queues, or even cancel the job currently processed by any transporter.

In a simulation model, the *dispatcher* is the component that embodies the dispatching policy. In a D3T model, the dispatcher contains all the *model-specific logic* of the D3T transport. Therefore, its internal state is typically rather intricate and highly model-specific. For sophisticated dispatching policies, the dispatcher will even contain an own view of the world: in this framework, the dispatcher does not "own" the transporters, their states are hidden from the dispatcher. The only way to know about the transporter states is via signalling (input/output events).

6.1.7 Summary: Parameters of a D3T model

To conclude this Section, a D3T model

$$\mathbf{M} = \Big((\mathcal{M}, d), \Sigma, (\mathcal{T}, \mathcal{Y}), N, \{ C_i \}_{i=1}^{N}, \{ \nu_i^0 \}_{i=1}^{N}, \tau_P, \tau_D, \mathbf{D}_{\mathfrak{D}}, \mathcal{O} \Big) \in \hat{\mathcal{M}}$$

is specified by the following parameters:

- the transport space $\mathcal{M}$ (either geodesic or network) and the associated hemimetric d,
- the set Σ of additional load marks and the load arrival process $(\mathcal{T}, \mathcal{Y}) = \{ (T_n, Y_n), n \in \mathbb{N} \}$ on the mark space $\mathcal{M} \times \mathcal{M} \times \Sigma$,
- the number N of transporters,
- the capacities $\{ C_i \}_{i=1}^{N}$ of the transporters with $C_i \in \mathbb{N} \cup \{ \infty \}$,

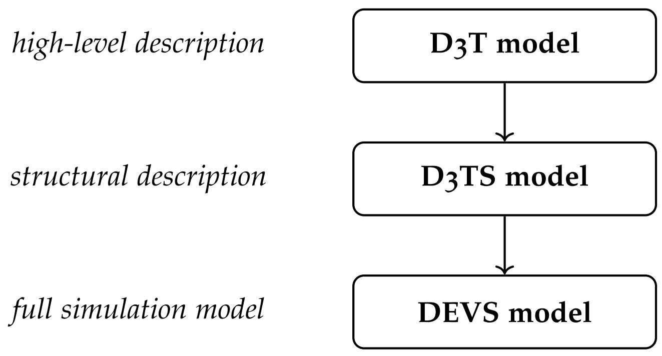

Figure 6.5: Three-tier description of a D3T model, which maps unto a D3TS model as described in this Section, and ultimately, unto a low-level but fully-fledged DEVS simulation model.

- the initial positions $\{ v_i^0 \}_{i=1}^N$ of the transporters with $v_i^0 \in \mathcal{M}$,
- the pick-up period function τ_P,
- the delivery period function τ_D,
- the dispatching policy $\mathfrak{D}$ (the dispatcher model $\mathbf{D}_{\mathfrak{D}}$),
- and additionally, we also include the set of observer models $\mathcal{O} = \{ \mathbf{O}_j \}_j$ that record simulation data for performance analysis.

6.2 THE D3T SPECIFICATION (D3TS)

6.2.1 Overview

A D3TS model is a formal mapping of a D3T model onto a system of interacting components in the system-theoretic sense. [130, 131] The DEVS description of a D3TS model is the complete description for simulation of a D3T model (cf. Figure 6.5). In other words, my contribution here is a common formal language (D3TS model) for specifying demand-driven directed transport models in the general formal P-DEVS language. The existing P-DEVS framework in turn comes with an abstract simulation algorithm and general implementations such as the *adevs* C++ library. [133, 146]

The D3TS specification defines generic *component classes* of specific D3T model components. There are four pairwise disjoint *component classes*:

- Load source class $\hat{L}$, containing the *load sources*, the simulation components which embody the load arrival process;
- Dispatcher class $\hat{D}$,
- Transporter class $\hat{T}$,
- Observer class $\hat{O}$.

One of the characteristic design traits of DEVS and the D3T Specification is that each simulation component is modelled as a black box: Internal states

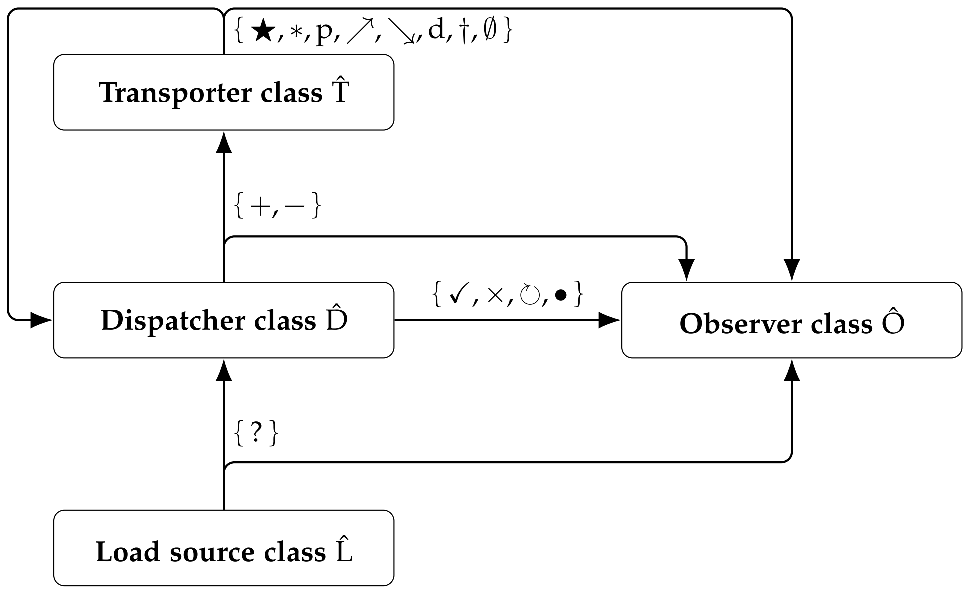
{★, ∗, p, ↗, ↘, d, †, ∅}
Transporter class $\hat{T}$
{+, −}
Dispatcher class $\hat{D}$
{✓, ×, ↻, •}
Observer class $\hat{O}$
{?}
Load source class $\hat{L}$

Figure 6.6: The D3T system framework.

are inaccessible, the only way of knowing about the system and its other components is via input and output events. This specification provides several *event types*. Examples of events are load arrivals (*requests*) or a transporter delivering loads (*delivery*).

Given an event type $\mathbf{e}$, all event instances $e \in \mathbf{e}$ of that type originate from component instances C of only one class $\hat{C}_{\mathbf{e}}$. For example, only load source instances output request events, and only transporter instances output delivery events. The component classes and their event types define the D3TS framework (Figure 6.6).

6.2.2 Event types

This Section introduces all event types $\mathbf{e}$ of this framework (Table 6.7).

REQUEST EVENT A load source instance L outputs an event instance $e_?$ of the *request* event type ? whenever a load arrives. The mark is $(\sigma_l, \bigcirc_l, \Box_l)$. The output event instance is $e_? = (l, \sigma, \bigcirc, \Box)$ with the load index l, the load origin $\bigcirc = \bigcirc_l$ and the load destination $\Box = \Box_l$.

ASSIGN EVENT A dispatcher instance D outputs an event instance $e_\checkmark = (l, i)$ of the *assign* event type ✓ whenever it assigns a load l for transport by transporter i. This event confirms the load request, i.e. assignment implies acceptance of the load request.

REJECT EVENT A dispatcher instance D outputs an event instance $e_\times = (l)$ of the *reject* event type $\times$ whenever it rejects a load request l.

SUBMIT EVENT A dispatcher instance D outputs an event instance $e_+ = (i, \tilde{\mathbf{j}}, r)$ of the *submit* event type $+$ to submit a sequence of jobs $\tilde{\mathbf{j}}$ to a transporter i. If the boolean *replace* variable r is *true*, the submitted jobs replace

Table 6.7: Event types and their instance data tuples, names, and producing component classes.

Name	$\mathbf{e}$	$\hat{C}_{\mathbf{e}}$	Instance $e \in \mathbf{e}$
request	$?$	$\hat{L}$	$(l, \sigma, \bigcirc, \Box)$
assign	$\checkmark$	$\hat{D}$	(l, i)
reject	$\times$	$\hat{D}$	(l)
submit	$+$	$\hat{D}$	$(i, \tilde{\mathbf{j}}, r)$
cancel	$-$	$\hat{D}$	(i)
busy	$\circlearrowright$	$\hat{D}$	(i)
idle	$\bullet$	$\hat{D}$	(i)
init	$\bigstar$	$\hat{T}$	(i, v, C)
job start	$*$	$\hat{T}$	(i, j)
pick-up	p	$\hat{T}$	$(i, j, \mathbb{P})$
departure	$\nearrow$	$\hat{T}$	$(i, j, \tilde{v}, \tilde{\tau})$
arrival	$\searrow$	$\hat{T}$	(i, j, v)
delivery	d	$\hat{T}$	$(i, j, \mathbb{D})$
job end	$\dagger$	$\hat{T}$	(i, j)
empty queue	$\emptyset$	$\hat{T}$	(i)

the transporter queue Q_i. Otherwise, the submitted jobs are appended to the queue.

CANCEL EVENT A dispatcher instance D outputs an event instance $e_{-} = (i)$ of the *cancel* event type $+$ to order a transporter i to cancel processing its current job. Hence, this command is particularly useful in combination with a submit event that replaces the current transporter queue. In this combination, the transporter receives new instructions to follow immediately.

BUSY EVENT A dispatcher instance D outputs an event instance $e_{\circlearrowright} = (i)$ of the *busy* event type $\circlearrowright$ to report that a transporter i has become busy.

IDLE EVENT A dispatcher instance D outputs an event instance $e_{\bullet} = (i)$ of the *idle* event type $\bullet$ to report that a transporter i has become idle.

INIT EVENT At the begin of the simulation, a transporter instance T_i outputs an event instance $e_{\bigstar} = (i, v, C)$ of the *init* event type $\bigstar$ to report initialization of transporter i at position v with capacity C. Together with the init event instance, the transporter instance outputs an empty queue event instance (see below).

JOB START EVENT A transporter instance T_i outputs an event instance $e_{*} = (i, j)$ of the *job start* event type $*$ when transporter i starts processing job no. j.

PICK-UP EVENT A transporter instance T_i outputs an event instance $e_p = (i, j, \mathbb{P})$ of the *pick-up* event type p when transporter i finishes picking up the loads $\tilde{\mathbb{P}}$ of its current job j.

DEPARTURE EVENT A transporter instance T_i outputs an event instance $e_{\nearrow} = (i, j, \tilde{v}, \tilde{\tau})$ of the *departure* event type $\nearrow$ when transporter i departs from its current position v to travel to the destination $\tilde{v}$ of its current job j. The expected travel time is $\tilde{\tau} = d(v, \tilde{v})$.

ARRIVAL EVENT A transporter instance T_i outputs an event instance $e_{\searrow} = (i, j, v)$ of the *arrival* event type $\searrow$ whenever transporter i arrives (and stops) at a position v. The transporter either stops at its scheduled destination $\tilde{v}$, or when the dispatcher canceled its current job j.

DELIVERY EVENT A transporter instance T_i outputs an event instance $e_d = (i, j, \mathbb{D})$ of the *delivery* event type d whenever transporter i finishes delivering the loads $\tilde{\mathbb{D}}$ of its current job j.

JOB END EVENT A transporter instance T_i outputs an event instance $e_\dagger = (i, j)$ of the *job end* event type $\dagger$ when transporter i ends processing the current job j. This also includes jobs that have been prematurely canceled by the dispatcher instance.

EMPTY QUEUE EVENT A transporter instance T_i outputs an event instance $e_\emptyset = (i)$ of the *empty queue* event type $\emptyset$ whenever transporter i has ended processing its current job and finds its queue empty, $Q_i = \emptyset$. It also outputs an empty queue event instance at time of initialization (as transporter instances are initialized with empty queues).

TRANSPORTER EVENTS We refer to the event types $\{\bigstar, *, p, \nearrow, \searrow, d, \dagger, \emptyset\}$ as *transporter events*.

6.2.3 Mapping a D3T model to a D3T system

A D3T system contains a certain number of instances of the underlying D3TS component model classes, as specified by mapping the D3T model. Note that the behavior of a component model $\mathbf{C}(\mathbf{M})$ for a given D3T model $\mathbf{M}$ depends on the other component models and the other parameters of the D3T model.

LOAD SOURCES Given a D3T model $\mathbf{M}$, the load source model $\mathbf{L}(\mathbf{M})$ embodies the marked point process $\{(T_n, Y_n), n \in \mathbb{N}\}$. A D3T system can contain multiple load sources at the discretion of the modeler. Load sources do not have any input events.

DISPATCHER A dispatcher component model $\mathbf{D}_{\mathfrak{D}} \in \hat{D}$ implements a specific dispatching policy $\mathfrak{D}$. In each D3T system, there is only one instance $\mathbf{D}_{\mathfrak{D}}(\mathbf{M})$ of the dispatcher model. The framework requires each dispatcher model to output submit, assign, busy, and idle events. The dispatcher model may further output cancel and/or reject events, depending on the dispatching policy.

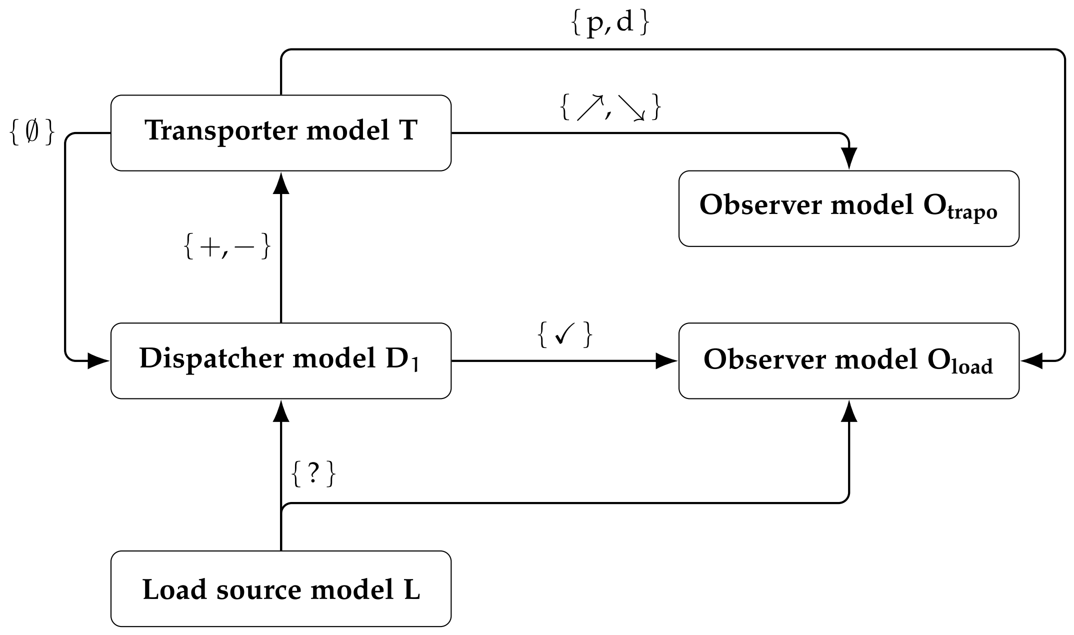

Figure 6.8: D3TS model graph of the example D3T model with the component models as nodes and arcs labelled by the event types.

TRANSPORTER This framework specifies exactly one transporter component model $\mathbf{T} \in \hat{\mathrm{T}}$. For a given D3T model $\mathbf{M}$, there is a transporter instance $T_i \in \mathbf{T}(\mathbf{M})$ for each of the N transporters. An instance T_i is initialized with initial position v_i^0 and empty cargo $\mathbb{L} = \emptyset$ at time $t = 0$.

OBSERVER An observer listens to simulation events but does not interfere with the simulation: it is the simulator's tool to record simulation statistics, instead of accessing simulation components' states directly. This design pattern facilitates and enforces encapsulation and the black-box paradigm. There are various observer component models $\mathbf{O} \in \hat{\mathrm{O}}$. Each D3T model $\mathbf{M}$ may specify multiple observer models $\mathbf{O}_i \in \hat{\mathrm{O}}$, and in the D3T system there will be exactly one instance $O_i \in \mathbf{O}_i(\mathbf{M})$ of each observer model. As observers are passive, they do not affect the dynamics of the D3T system. For a D3T model, the observer models merely specify which observables are accessible for recording.

6.2.4 A D3T model example

Let us illustrate the technicalities of the D3TS component and event framework with a concrete example $\mathbf{M}_1$ of a D3T model. As there is only one load source model $\mathbf{L} \in \hat{\mathrm{L}}$ and one transporter model $\mathbf{T} \in \hat{\mathrm{T}}$, we only need to specify the dispatcher model $\mathbf{D}_1 \in \hat{\mathrm{D}}$ and the observer models $\mathcal{O}$. Specifically, let the number of transporters be $N = 3$. Let the dispatcher model be $\mathbf{D} = \mathbf{D}_1$, and let the set of observer models be $\mathcal{O} = \{ \mathbf{O}_{load}, \mathbf{O}_{trapo} \}$.

Let us consider a dispatching policy which cancels transporter jobs (− event type), and reacts upon transporters reporting empty queues (∅ event type). Let observer model $\mathbf{O}_{load}$ record load statistics of arrivals, assignments, pick-ups and deliveries, and let observer model $\mathbf{O}_{trapo}$ record transporter statistics such as departures and arrivals. These component input/output ports determine the *D3TS model graph* (Figure 6.8) which is a directed graph without loops.

6.2.5 On the mapping of a D3TS model to a DEVS model

To complete the presentation of the framework, I sketch the technical mapping of a D3T model instance M to a parallel DEVS network model $N(M) = (X_N, Y_N, D, \mathcal{I}, Z)$ with ports. Each port p in the DEVS network model is an event type. The DEVS network model coupling functions treat all event instances independently and only couple output ports to input ports of the same event type. The DEVS network model itself neither takes input nor produces output ($X_N(M) = Y_N(M) \equiv \emptyset$.) The set of components $D(M)$ of the network model contains a component d for each component instance C in the D3T system as specified in Section 6.2.3. The family of sets of influencers $\mathcal{I}$ directly derives from the D3T system framework (Figure 6.6). A coupling function $z_{d,d'} \in Z$ transforms output of the DEVS network component d to input of the DEVS network component d'. Within the component and event framework it is again particulary simple to define this transformation. From any output bag y^b that d generates, it selects all events (p, v) whose type p is implemented as input at the receiving end.

The load source component model generates the load arrival events. In fact, every instance of the DEVS network model (i.e. every simulation run) implements one realization of the load arrival process, through the load source. Note that it is in this sense that the deterministic DEVS formalism accounts for random events: the DEVS model instance is formally initialized with a single realization of the stochastic process. However, the software implementation typically generates the respective random events on the fly. Within the constraints of this framework regarding the event types, the D3T modeler is free to specify dispatcher and observer component models as generic DEVS atomic models. Finally, the transporter DEVS model derives from the transporter states (Table 6.2), including modes (Table 6.3), and the transition graph (Figure 6.4).

This completes the outline of the steps necessary to arrive at a formal DEVS description of a D3T system.

6.3 OBSERVABLE DATA OF D3T SYSTEMS

6.3.1 Overview

Observables are the quantitative output of the transport dynamics of a D3T instance.

Given a D3T instance M, each component $C \in M$ has an output trajectory $y_C(\tau)$ (see Section 4.4.1).

Definition 6.1 (*Total output trajectory*). *Let*

$$M = \Big(\{ L_\sigma : \sigma \in \Sigma \}, D_{\mathfrak{D}}, \{ T_i \}_{i=1}^{N}, \{ O \in \mathbf{O}(\mathbf{M}) : \mathbf{O} \in \mathcal{O} \} \Big)$$

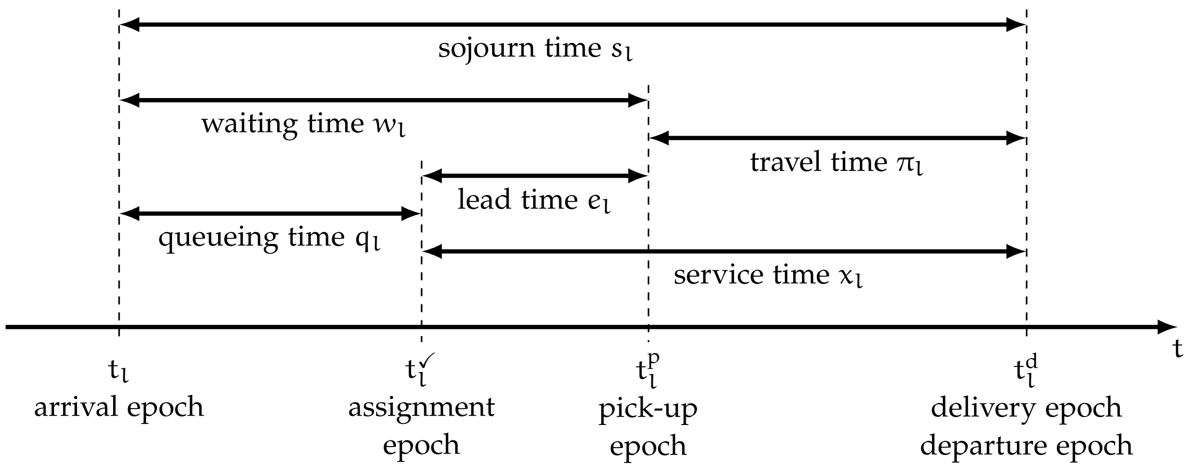

Figure 6.9: Load epochs and times as observables of D3T models.

be a D3T instance. At each time $\tau \in \mathbb{T}$*, the* total output trajectory *of* M *is*

$$\mathbf{y}_M(\tau) = \biguplus_{\sigma \in \Sigma} y_{L_\sigma}(\tau) \uplus y_{D_{\mathcal{D}}}(\tau) \uplus \biguplus_{i=1}^{N} y_{T_i}(\tau),$$

where $\uplus$ *denotes multiset addition.*

Hence, the total output trajectory aggregates the outputs of the dispatcher and all load sources and transporters.

Observer instances preprocess the total output trajectory $\mathbf{y}_M$ of a D3T instance M. They provide recordable simulation data. Specifically, each observer model implements a combination of

1. scalar values,
2. event trajectories, and
3. state trajectories,

as simulation output. While a scalar value aggregates a quantity over the whole simulation run, a trajectory provides online time-discrete output (event trajectories), or time-continuous output (state trajectories).

6.3.2 Load observables

OVERVIEW Load observables relate the transport dynamics to individual service quality. Figure 6.9, Table 6.10 and Table 6.11 summarize the observables regarding the arrival and transport of loads.

Table 6.10: Load observables in D3T models.

Symbol	Domain / Value	Observable
l	$\in \mathcal{L} = \mathbb{N}$	load index
$\bigcirc_l$	$\in \mathcal{M}$	origin
$\square_l$	$\in \mathcal{M}$	destination
d_l	$= d(\bigcirc_l, \square_l)$	direct travel time
t_l	$\in \mathbb{R}_0^+, t_l \geqslant t_{l-1}$	arrival epoch
Δt_l	$= t_l - t_{l-1}, \Delta t_1 = t_1$	interarrival time
$t_l^{\checkmark}$	$\geqslant t_l$	assignment epoch
t_l^p	$\geqslant t_l^{\checkmark}$	pick-up epoch
t_l^d	$\geqslant t_l^p + d_l$	delivery epoch, departure epoch
q_l	$= t_l^{\checkmark} - t_l$	queueing time
w_l	$= t_l^p - t_l$	waiting time
e_l	$= t_l^p - t_l^{\checkmark}$	lead time
x_l	$= t_l^d - t_l^{\checkmark}$	service time
π_l	$= t_l^d - t_l^p$	travel time
s_l	$= t_l^d - t_l$	sojourn time, system time
$\hat{q}_l$	$= q_l / d_l$	relative queueing time
$\hat{w}_l$	$= w_l / d_l$	relative waiting time
$\hat{e}_l$	$= e_l / d_l$	relative lead time
$\hat{x}_l$	$= x_l / d_l$	relative service time
$\hat{\pi}_l$	$= \pi_l / d_l$	relative travel time
$\hat{s}_l$	$= s_l / d_l$	relative sojourn time
n_l	$= l(t_l^-)$	system size at arrival epoch

Table 6.11: Derived load observables of D3T models.

Symbol	Domain / Value	Observable
$\mathcal{L}^0(t)$	$= \{ l : t_l \leqslant t \}$	arrival set
$\mathcal{L}^d(t)$	$= \left\{ l : t_l^d \leqslant t \right\}$	departure set
$\mathcal{L}(t)$	$= \left\{ l : t_l \leqslant t < t_l^d \right\} = \mathcal{L}^0(t) \setminus \mathcal{L}^d(t)$	system set
$n_L^0(t)$	$= \lvert \mathcal{L}^0(t) \rvert$	arrival number
$n_L^d(t)$	$= \lvert \mathcal{L}^d(t) \rvert$	departure number
$l(t)$	$= \lvert \mathcal{L}(t) \rvert$	system size
n_l	$= l(t_l^-)$	system size at arrival epoch
$l_k(t)$	$= \frac{1}{t} \int_0^t I_{\{ \tilde{t} : l(\tilde{t}) = k \}} \, dt'$	system size fraction
$n_k(t)$	$= \lvert \{ l : t_l \leqslant t, n_l = k \} \rvert$	arrival system size frequency

LOAD EPOCHS The arrival time t_l of the l-th load is implicitly defined as[1]

$$(?, (l, \sigma, \bigcirc, \square)) \in \mathbf{y}_M(t_l, c).$$

Similarly, the origin $\bigcirc_l = \bigcirc$ and destination $\square_l = \square$. Define the *direct travel time* as the metric distance from origin to destination:

$$d_l = d(\bigcirc_l, \square_l).$$

The *inter-arrival time* is

$$\Delta t_l = t_l - t_{l-1},$$

with $\Delta t_1 = t_1$. The dispatcher assigns the load for transport to a transporter i at the *assignment epoch* $t_l^{\checkmark}$. Its implicit definition is

$$(\checkmark, (l, i_l)) \in \mathbf{y}_M(t_l^{\checkmark}, c),$$

where i_l is the index of the transporter the load l is assigned to. Similarly, the implicit definitions of the pick-up epoch t_l^p and the delivery epoch t_l^d are

$$(p, (i, \mathbb{P})) \in \mathbf{y}_M(t_l^p, c), l \in \mathbb{P},$$
$$(d, (i, \mathbb{D})) \in \mathbf{y}_M(t_l^d, c), l \in \mathbb{D}.$$

LOAD TIMES In queueing theory, the assignment epoch is the time instant at which transporter i starts servicing load l. Hence, we call the period between arrival and assignment

$$q_l = t_l^{\checkmark} - t_l$$

the *queueing time*. The queueing time is less than the *waiting time* between arrival and pick-up

$$w_l = t_l^p - t_l,$$

which also includes the *lead time* between assignment and pick-up

$$e_l = t_l^p - t_l^{\checkmark}.$$

During the lead time, a transporter travels to the origin of the load. In queueing theory, the transporter would already serve the load. However, in transportation, the transporter has not picked up the load yet. Besides the lead time, the *service time* between assignment and delivery

$$x_l = t_l^d - t_l^{\checkmark}$$

1 A note on potential confusion of queueing and physical notions here: in queueing theory, *arrival* of a job or customer refers to the point in time a job *enters* the system, i.e. is known to and dealt with by the queueing system. In transport, arrival refers to the physical arrival of a load at its destination. In practice, we will not speak of arrival of a load, but rather of the *delivery* of a load – which entails that the load *departs* (or exits, leaves) the system. Hence, if not otherwise stated, arrival of a load means the arrival of a load to the (queueing) system. Similarly, departure of a load refers to the load being delivered and subsequently, leaving the queueing system.

as defined in queueing theory also comprises the *travel time* between pick-up and delivery

$$\pi_l = t_l^d - t_l^p,$$

which must be at least the direct travel time,

$$\pi_l \geqslant d_l.$$

Equality holds if the transporter travels directly from the load origin to the load destination. To conclude, the overall *sojourn time* or *system time* of load l between arrival and delivery

$$s_l = t_l^d - t_l$$

adds up as

$$s_l = w_l + \pi_l = q_l + x_l = q_l + e_l + \pi_l.$$

For each of these times $q_l, w_l, e_l, x_l, \pi_l, s_l$, we also consider the corresponding *relative* period normalized by the direct travel time d_l:

$$\hat{q}_l = \frac{q_l}{d_l}, \hat{w}_l = \frac{w_l}{d_l}, \hat{e}_l = \frac{e_l}{d_l}, \hat{x}_l = \frac{x_l}{d_l}, \hat{\pi}_l = \frac{\pi_l}{d_l}, \hat{s}_l = \frac{s_l}{d_l}.$$

LOAD SETS The *arrival set*

$$\mathcal{L}^0(t) = \{ l : t_l \leqslant t \}$$

is the set of all loads that have arrived by time t. Similarly, the *departure set*

$$\mathcal{L}^d(t) = \left\{ l : t_l^d \leqslant t \right\}$$

is the set of all loads that have departured by time t. Clearly, $\mathcal{L}^d(t) \subset \mathcal{L}^0(t)$. The *system set*

$$\mathcal{L}(t) = \left\{ l : t_l \leqslant t < t_l^d \right\} = \mathcal{L}^0(t) \setminus \mathcal{L}^d(t)$$

is the set of loads currently in the system at time t.

LOAD NUMBERS The *arrival number*

$$n_L^0(t) = |\mathcal{L}^0(t)|$$

is the number of loads that have arrived by time t. Similarly, the *departure number*

$$n_L^d(t) = |\mathcal{L}^d(t)|$$

is the number of loads that have departured by time t. The *system size*

$$l(t) = |\mathcal{L}(t)| = n_L^0(t) - n_L^d(t)$$

is the number of loads currently in the system. The system size at arrival epochs t_l, excluding the arrivals at t_l, is denoted

$$n_l = l(t_l^-).$$

The fraction of time the system size equals k up to time t (system size fraction) is

$$l_k(t) = \frac{1}{t}\int_0^t I_{\{\tilde{t}:l(\tilde{t})=k\}}\, dt'.$$

The number of arrivals to a system of size k up to time t (arrival system size frequency) is

$$n_k(t) = |\{\, l : t_l \leqslant t, n_l = k \,\}|.$$

SYSTEM SIZE BUSY PERIODS AND RETURN TIME Given an integer number $c \in \mathbb{N}$, what is the distribution of periods during which the D3T system has more than c loads? We call these periods *c-busy periods*. The intermediate periods are the *c-idle periods*. The start epochs of the c-busy periods up to time t are

$$\mathbb{t}^{b,c}(t) = \left\{\, \tilde{t} : \tilde{t} \leqslant t, l(\tilde{t}) \geqslant c, l(\tilde{t}^-) < c \,\right\},$$

and the start epochs of the c-idle periods up to time t are

$$\mathbb{t}^{i,c}(t) = \{\, 0 \,\} \cup \left\{\, \tilde{t} : \tilde{t} \leqslant t, l(\tilde{t}) < c, l(\tilde{t}^-) \geqslant c \,\right\}.$$

The start epochs of the c-idle periods are the end epochs of the c-busy periods, and vice versa. The symbol $l(\tilde{t}^-)$ refers to the right limit $\lim_{t\nearrow\tilde{t}} l(t)$. The time-ordered sequences are $(t_n^{b,c})_n$ and $(t_n^{i,c})_n$ with $0 \leqslant t_1^{b,c} \leqslant t_2^{b,c} \leqslant \ldots$ and $0 = t_0^{i,c} \leqslant t_1^{i,c} \leqslant t_2^{i,c} \leqslant \ldots$. Observe that by definition

$$0 = t_0^{i,c} \leqslant t_1^{b,c} \leqslant t_1^{i,c} \leqslant t_2^{b,c} \leqslant t_2^{i,c} \leqslant \ldots$$

The *system size c-busy periods* and *system size c-idle periods* are

$$\begin{aligned} b_n^c &= t_n^{i,c} - t_n^{b,c} \\ i_n^c &= t_n^{b,c} - t_{n-1}^{i,c}. \end{aligned}$$

The *system size return times* are the system size N-busy times

$$r_n = b_n^N,$$

where N is the number of transporters. The number of arrivals during a system size c-busy period up to time t is

$$n^{b,c}(t) = \{\, l : t_l \leqslant t, n_l \geqslant c \,\},$$

(*system size c-busy arrival number*) and the number of arrivals during a system size c-idle period up to time t is

$$n^{i,c}(t) = \{\, l : t_l \leqslant t, n_l < c \,\},$$

the *system size c-idle arrival number*. Obviously,

$$n_L^0(t) = n^{b,c}(t) + n^{i,c}(t).$$

The *system size c-busy arrival fraction* of arrivals up to time t is

$$p^{b,c}(t) = \frac{n^{b,c}(t)}{n_L^0(t)}.$$

The *system size delay fraction* is the system size N-busy arrival fraction up to time t,

$$p^b(t) = p^{b,N}(t),$$

where N is again the number of transporters.

PAYLOAD-WEIGHTED TRAVEL TIME The payload-weighted travel time of load l is

$$\psi_l = \int_{t_l^p}^{t_l^d} (n_{i_l}(t))^{-1}\, dt,$$

where $n_{i_l}(t)$ is the payload size trajectory of the transporter i_l that load l was assigned to.

6.3.3 Transporter observables

OVERVIEW Transporter observables relate the transport dynamics to system efficiency. Each transporter i processes a sequence of jobs $j_1, j_2, \dots, j_J$. Consider an arbitrary transporter i throughout, unless otherwise stated.

TRANSPORTER JOB EPOCHS The time-ordered sequences of *start epochs* t_{ij}^* and *end epochs* $t_{ij}^\dagger$ of the j-th job are

$$\begin{aligned}
\mathbb{t}_i^* &= \{\, t : (*, (i, \nu, \mathbb{L})) \in \mathbf{y}_M(t)\,\} = (t_{ij}^*)_j,\\
\mathbb{t}_i^\dagger &= \{\, t : (*, (i, \nu, \mathbb{L})) \in \mathbf{y}_M(t)\,\} = (t_{ij}^\dagger)_j.
\end{aligned}$$

By definition, it holds that $0 \leqslant t_{i1}^* \leqslant t_{i1}^\dagger \leqslant t_{i2}^* \leqslant t_{i2}^\dagger \leqslant \dots$. The *pick-up epoch* t_{ij}^p, *departure epoch* $t_{ij}^{\nearrow}$, *arrival epoch* $t_{ij}^{\searrow}$, *delivery epoch* t_{ij}^d of the j-th job are implicitely defined as

$$\begin{aligned}
(p, (i, \mathbb{P})) &\in \mathbf{y}_M\big|_{[t_{ij}^*, t_{ij}^\dagger)}(t_{ij}^p),\\
(\nearrow, (i, \tilde{\nu}, \tilde{\tau})) &\in \mathbf{y}_M\big|_{[t_{ij}^*, t_{ij}^\dagger)}(t_{ij}^{\nearrow}),\\
(\searrow, (i, \nu)) &\in \mathbf{y}_M\big|_{[t_{ij}^*, t_{ij}^\dagger)}(t_{ij}^{\searrow}),\\
(d, (i, \mathbb{D})) &\in \mathbf{y}_M\big|_{[t_{ij}^*, t_{ij}^\dagger)}(t_{ij}^d).
\end{aligned}$$

If both respective epochs are defined, the pick-up epoch coincides with the departure epoch, and the delivery epoch coincides with the job end epoch,

$$\begin{aligned}
t_{ij}^p &= t_{ij}^{\nearrow},\\
t_{ij}^d &= t_{ij}^\dagger.
\end{aligned}$$

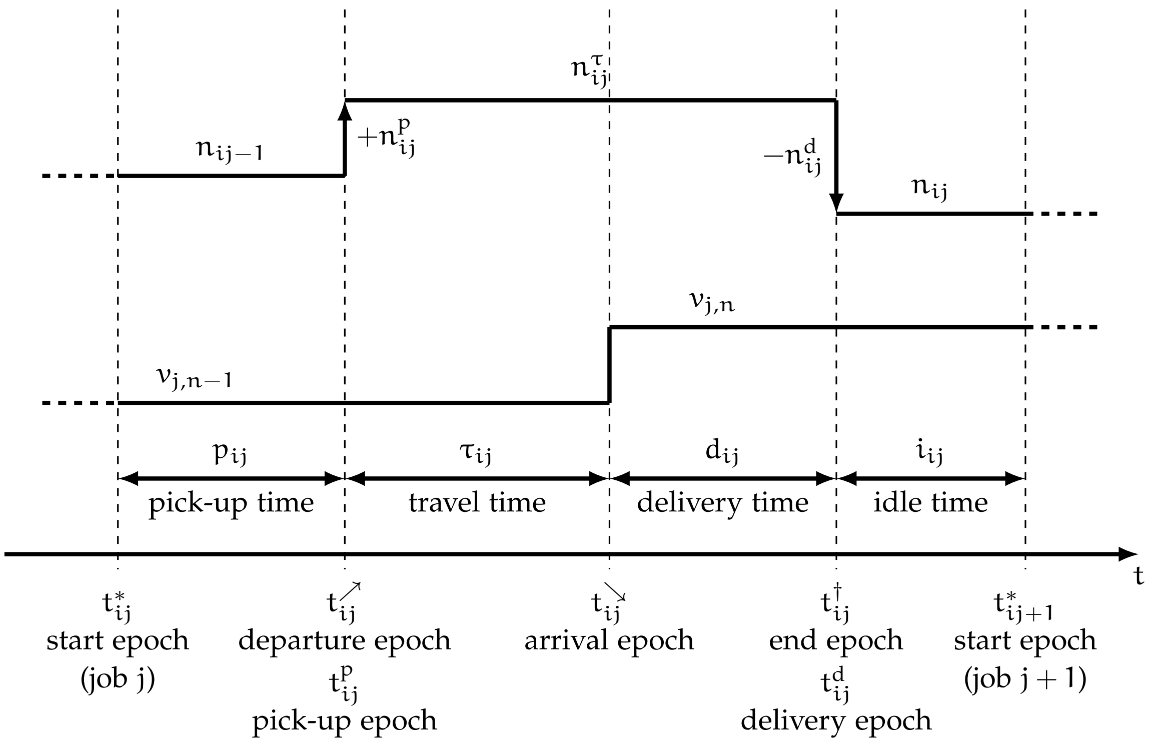

Figure 6.12: Epochs, times, and payload sizes as observables of transporter jobs in D3T models.

TRANSPORTER PAYLOAD For each transporter i, there is a payload trajectory $\mathbb{L}_i(t)$ which records the loads on-board the transporter for each time t. The payload trajectory $\mathbb{L}_i(t)$ is a piecewise constant trajectory that changes only at pick-up epochs t_{ij}^{p} and delivery epochs t_{ij}^{d}. The *job residual payload* $\mathbb{L}_{ij}$ is the payload of transporter i at the end of its j-th job. At the pick-up epoch t_{ij}^{p}, the pick-up set $\mathbb{P}_{ij}$ of loads is added to the payload and implicitely defined as

$$(p, (i, \mathbb{P}_{ij})) \in \mathbf{y}_M(t_{ij}^{p}).$$

Similarly, at the delivery epoch t_{ij}^{d}, the delivery set $\mathbb{D}_{ij}$ of loads is subtracted from the payload and implicitely defined as

$$(d, (i, \mathbb{D}_{ij})) \in \mathbf{y}_M(t_{ij}^{d}).$$

If a pick-up epoch t_{ij}^{p} is undefined, we formally let $t_{ij}^{p} \equiv t_{ij}^{\nearrow}$ and $\mathbb{P}_{ij} \equiv \emptyset$. Similarly, if a delivery epoch t_{ij}^{d} is undefined, we formally let $t_{ij}^{d} \equiv t_{ij}^{\dagger}$ and $\mathbb{D}_{ij} \equiv \emptyset$. The recursive definition of the job payloads $\mathbb{L}_{ij}$ is

$$\mathbb{L}_{ij} = \mathbb{L}_{ij-1} \cup \mathbb{P}_{ij} \setminus \mathbb{D}_{ij},$$

with the initial empty payload

$$\mathbb{L}_{i0} = \emptyset.$$

The *job transport payload* $\mathbb{L}_{ij}^{\tau}$ is the set of loads on-board transporter i during the travel time of its j-th job,

$$\mathbb{L}_{ij}^{\tau} = \mathbb{L}_{ij-1} \cup \mathbb{P}_{ij}.$$

The constant segments of the payload trajectory are

$$\begin{aligned}\mathbb{L}_i\big|_{[0,t^p_{i1})}(t) &= \emptyset,\\ \mathbb{L}_i\big|_{[t^p_{ij},t^d_{ij})}(t) &= \mathbb{L}^\tau_{ij},\\ \mathbb{L}_i\big|_{[t^d_{ij},t^p_{ij+1})}(t) &= \mathbb{L}_{ij}.\end{aligned}$$

For each transporter i, the *payload size trajectory* is the number of loads on-board the transporter at any time t,

$$n_i(t) = |\mathbb{L}_i(t)|.$$

As the transporter payload trajectory, the payload size trajectory is also piecewise constant. The *job residual payload size* n_{ij} is the number of loads the transporter i carries at the end of its j-th job,

$$n_{ij} = |\mathbb{L}_{ij}|.$$

The *pick-up number* n^p_{ij} is the number of loads the transporter i picks up during its j-th job,

$$n^p_{ij} = |\mathbb{P}_{ij}|.$$

Similary, the *delivery number* n^d_{ij} is the number of loads the transporter i delivers during its j-th job,

$$n^d_{ij} = |\mathbb{D}_{ij}|.$$

The *job transport payload size* n^τ_{ij} is the number of loads the transporter i is carrying during the travel time of its j-th job,

$$n^\tau_{ij} = |\mathbb{L}^\tau_{ij}|.$$

From the payload sets, we have the following relationships for the payload sizes:

$$\begin{aligned}n_{i0} &= 0,\\ n_{ij} &= n_{ij-1} + n^p_{ij} - n^d_{ij},\\ n^\tau_{ij} &= n_{ij-1} + n^p_{ij}.\end{aligned}$$

The constant segments of the payload size trajectory are

$$\begin{aligned}n_i\big|_{[0,t^p_{i1})}(t) &= 0,\\ n_i\big|_{[t^p_{ij},t^d_{ij})}(t) &= n^\tau_{ij},\\ n_i\big|_{[t^d_{ij},t^p_{ij+1})}(t) &= n_{ij}.\end{aligned}$$

For $k \geqslant 1$, the time-ordered epochs at which a transporter i starts to carry at least k loads are the k *occupancy epochs*

$$\begin{aligned}\mathfrak{t}^k_i(t) &= \left\{\,\tilde{t} \leqslant t : n_i(\tilde{t}) \geqslant k, n_i(\tilde{t}^-) \neq k\,\right\} =\\ &= \left\{\, t^p_{ij} \leqslant t : n^p_{ij} > 0, k > n_{ij-1} \geqslant k - n^p_{ij}\,\right\} = (t^k_{i,n})_n.\end{aligned}$$

The epochs at which the number of loads a transporter i carries drops below k are the k *occupancy end epochs*

$$\begin{aligned}\mathbb{t}_i^{\bar{k}}(t) &= \{\,0\,\} \cup \{\,\tilde{t} \leqslant t : n_i(\tilde{t}) < k, n_i(\tilde{t}^-) \geqslant k\,\} \\ &= \{\,0\,\} \cup \left\{\, t_{ij}^d \leqslant t : n_{ij}^d > 0, k \leqslant n_{ij}^\tau < k + n_{ij}^d \,\right\} = (t_{i,n}^{\bar{k}})_n\end{aligned}$$

with $t_{i,0}^{\bar{k}} = 0$.

It should hold that

$$0 = t_{i,0}^{\bar{k}} \leqslant t_{i,1}^{k} \leqslant t_{i,1}^{\bar{k}} \leqslant t_{i,2}^{k} \leqslant t_{i,2}^{\bar{k}} \leqslant \ldots.$$

The k *occupancy periods* $n_{i,n}^k$ are the periods during which a transporter i carries at least k loads,

$$n_{i,n}^k = t_{i,n}^{\bar{k}} - t_{i,n}^{k}.$$

The *empty periods* $e_{i,n}$ are the periods during which a transporter i is empty,

$$e_{i,n} = t_{i,n+1}^{1} - t_{i,n}^{\bar{1}}.$$

For any given k, the piecewise continuous k *occupancy trajectory* is

$$n_i^k(t) = \mathbf{1}_{\bigcup_n [t_{i,n}^k, t_{i,n}^{\bar{k}})}(t).$$

The *empty trajectory* e_i is piecewise continuous,

$$e_i(t) = \mathbf{1}_{\bigcup_n [t_{i,n}^{\bar{1}}, t_{i,n+1}^{1})}(t).$$

TRANSPORTER JOB TIMES In the following, consider the j-th job of transporter i. If the transporter picks up loads, that is, if the pick-up epoch t_{ij}^p is defined, the *pick-up time* is

$$p_{ij} = t_{ij}^p - t_{ij}^*.$$

Analogously, if the delivery epoch t_{ij}^d is defined, the *delivery time* is

$$d_{ij} = t_{ij}^d - t_{ij}^{\searrow}.$$

The *travel time* is

$$\tau_{ij} = t_{ij}^{\searrow} - t_{ij}^{\nearrow}.$$

The *idle time* between two successive jobs j and $j+1$ is

$$i_{ij} = t_{ij+1}^* - t_{ij}^\dagger.$$

The piecewise continuous *pick-up trajectory*, *delivery trajectory*, *travel trajectory*, and *interjob-idle trajectory* are

$$\begin{aligned}p_i(t) &= \mathbf{1}_{\bigcup_j [t_{ij}^*, t_{ij}^p)}(t) = \mathbf{1}_{\bigcup_j [t_{ij}^*, t_{ij}^{\nearrow})}(t),\\ d_i(t) &= \mathbf{1}_{\bigcup_j [t_{ij}^{\searrow}, t_{ij}^d)}(t) = \mathbf{1}_{\bigcup_j [t_{ij}^{\searrow}, t_{ij}^\dagger)}(t),\\ \tau_i(t) &= \mathbf{1}_{\bigcup_j [t_{ij}^{\nearrow}, t_{ij}^{\searrow})}(t),\\ j_i(t) &= \mathbf{1}_{\bigcup_j [t_{ij}^\dagger, t_{ij+1}^*)}(t),\end{aligned}$$

TRANSPORTER POSITIONS The *transporter job position* ν_{ij} is implicitely defined as

$$(\searrow, (i, \nu_{ij})) \in \mathbf{y}_M(t^{\searrow}_{ij})$$

with the initial condition $\nu_{i0} = \nu^0_i$.

The *transporter job scheduled destination* $\tilde{\nu}_{ij}$ and *transporter job scheduled travel time* $\tilde{\tau}_{ij}$ are implicitely defined as

$$(\nearrow, (i, \tilde{\nu}_{ij}, \tilde{\tau}_{ij})) \in \mathbf{y}_M(t^{\nearrow}_{ij}).$$

The *transporter job position trajectory* is piecewise continuous, and its constant segments are

$$\nu^J_i \Big|_{[0, t^{\searrow}_{i1})}(t) = \nu^0_i,$$
$$\nu^J_i \Big|_{[t^{\searrow}_{ij}, t^{\searrow}_{ij+1})}(t) = \nu_{ij}.$$

The *transporter position trajectory* is

$$\nu_i(t) = \begin{cases} \nu_{ij} & \text{if } t \in [t^{\searrow}_{ij}, t^{\nearrow}_{ij+1}) \\ \nu(\nu_{ij-1}, \tilde{\nu}_{ij}, \tilde{\tau}_{ij} - (t - t^{\nearrow}_{ij})) & \text{if } t \in [t^{\nearrow}_{ij}, t^{\searrow}_{ij}), \end{cases}$$

where $\tilde{\tau}_{ij} = d(\nu_{ij-1}, \tilde{\nu}_{ij})$.

CUMULATIVE AND AVERAGE TRANSPORTER TIMES Let the simulation start at time 0 and end at time T. We may discard an initial transient phase ending at t_0. For any given trajectory $x_i(t)$, we consider

- the *total trajectory* $x(t) = \sum_{i=1}^N x_i(t)$,
- the *cumulative trajectory* $X_i(t) = \int_0^t x_i(t')\, dt'$, and the *cumulative value* $X_i = X_i(T) - X_i(t_0)$,
- the *cumulative total trajectory* $X(t) = \sum_{i=1}^N X_i(t) = \int_0^t x(t')\, dt' = \sum_{i=1}^N \int_0^t x_i(t')\, dt'$, and the *cumulative total* $X = X(T) - X(t_0)$,
- the *time average* $\bar{X}_i = \frac{1}{T-t_0} X_i$,
- the *time-averaged total* $\bar{X} = \frac{1}{T-t_0} X$,
- the *transporter average trajectory* $\langle x \rangle_t = \frac{1}{N} x(t)$,
- the *transporter-averaged cumulative trajectory* $\langle X \rangle_t = \frac{1}{N} X(t)$, and the *transporter-averaged cumulative value* $\langle X \rangle = \langle X \rangle_T - \langle X \rangle_{t_0}$,
- the *(time and transporter) average* $\langle \bar{X} \rangle = \frac{1}{N} \bar{X} = \frac{1}{T-t_0} \langle X \rangle$.

6.4 EXAMPLE DISPATCHER MODEL

DISPATCHING POLICY The **Myopic Taxi FCFS Nearest-Transporter** *dispatching policy*

- is *myopic*: it
  - assigns and submits a request to a transporter at the same time,
  - only assigns requests to currently idle transporters,
  - assigns a request to a transporter as soon as there are idle transporters,
  - assigns a transporter to a request as soon as there are pending requests,
  - does not cancel any jobs that it submits;
- dispatches *taxis*: all transporters have unit capacity 1;
- serves requests on a First-Come-First-Serve basis: if all transporters are busy, it assigns the oldest request to the next transporter becoming idle,
- chooses the nearest idle transporter: if a transporter becomes idle and there are no pending requests, it assigns the next request to the idle transporter that is nearest to the request origin.

The component model is the **Myopic Taxi FCFS Nearest-Transporter (MTFN) Dispatcher D_{MTFN}**.

BUSY/IDLE EVENTS A transporter becomes busy when the dispatcher assigns a request to it. At the same time, the dispatcher submits the jobs to serve that request. However, the transporter becomes busy only when it was idle before. There is an exception to this rule: When the transporter has just become idle, i.e. when it signals an empty queue at the same time the new request arrives and the assignment is imminent, the transporter does does become neither idle nor busy, but remains busy. Similarly, a transporter becomes idle when the dispatcher receives an empty queue event, but has not assigned any new requests at the same time.

The dispatcher outputs all required events $\{\checkmark, +, \circlearrowleft, \bullet\}$. The dispatcher needs transporter initialization and empty queue events as input. It also needs to keep track of transporter positions through the arrival events: $\{?, \bigstar, \searrow, \emptyset\}$.

INTERNAL STATES The internal state $s = (T, T^*, V, R, I, I^*, A^*)$ stores

- the set of transporters T,
- the set of newly initialized transporters T^*,
- the transporter positions $V = (v_i)_{i \in T}$,
- the queue of pending requests R,
- the set of idle transporters I,
- the set of newly idle transporters I^*,
- the set of new assignments A^*.

As a convenience, we define the function $T(A^*) = \{\, i \mid \exists l : (l, i, \bigcirc_l, \square_l) \in A^* \,\}$.

TIME ADVANCEMENT The dispatcher remains passive unless there are requests to assign or newly idle transporters to signal. If so, the dispatcher is imminent:

$$\mathrm{ta}(s) = \begin{cases} 0 & \text{if } A^* \neq \{\,\} \vee I^* \neq \{\,\} \\ \infty & \text{otherwise} \end{cases}$$

OUTPUT We partition the output function into the port output functions:

$$\lambda(s) = \bigcup_{\mathbf{e} \in \mathbf{Y}_{\mathbf{D}_{\mathrm{MTFF}}}} y^b_{\mathbf{e}}$$

with the port output bags $y^b_{\mathbf{e}} = \{\, (\mathbf{e}, e) \mid e \in \lambda_{\mathbf{e}}(s) \,\}$.

$$\begin{aligned} \lambda_{+}(s) &= \left\{\, \left(i, \tilde{\mathbf{j}}_l, 0\right) \,\middle|\, (l, i, \bigcirc_l, \square_l) \in A^* \,\right\} \\ \lambda_{\checkmark}(s) &= \{\, (l, i) \mid \exists l, i\colon (l, i, \bigcirc_l, \square_l) \in A^* \,\} \\ \lambda_{\circlearrowright}(s) &= \{\, (i) \mid i \in T(A^*) \cap ((I \setminus I^*) \cup T^*) \,\} \\ \lambda_{\bullet}(s) &= \{\, (i) \mid i \in I^* \setminus T(A^*) \,\} \end{aligned}$$

with $\tilde{\mathbf{j}}_l = ((\emptyset, \bigcirc_l, \emptyset), (\{\, l \,\}, \square_l, \{\, l \,\}))$.

INTERNAL TRANSITION We partition the internal transition function δ_{int} into transition functions dealing with clearing assignments and idle transporters:

$$\begin{aligned} \delta_{\mathrm{int}} &= \delta_{\mathrm{int},A} \circ \delta_{\mathrm{int},I} \\ \delta_{\mathrm{int},I} &: I \mapsto I \setminus T(A^*), I^* \mapsto \emptyset, T^* \mapsto \emptyset \\ \delta_{\mathrm{int},A} &: A^* \mapsto \emptyset \end{aligned}$$

EXTERNAL TRANSITION We partition the external transition function δ_{ext} into transition functions dealing each with one event type, and the matching transition function δ_{match}:

$$\delta_{\mathrm{ext}} = \delta_{\mathrm{match}} \circ \delta_{\mathrm{ext},\emptyset} \circ \delta_{\mathrm{ext},?} \circ \delta_{\mathrm{ext},\searrow} \circ \delta_{\mathrm{ext},\bigstar}$$

$$\begin{aligned} \delta_{\mathrm{ext},\bigstar}\colon T^* &\mapsto \left\{\, i \,\middle|\, \exists v : (i, v) \in x^b_{\bigstar} \,\right\} \\ T &\mapsto T \cup \left\{\, i \,\middle|\, \exists v : (i, v) \in x^b_{\bigstar} \,\right\} \\ V &\mapsto V \cup \left\{\, v \,\middle|\, \exists i : (i, v) \in x^b_{\bigstar} \,\right\} \end{aligned}$$

$$\delta_{\mathrm{ext},\searrow}\colon v_i \mapsto \begin{cases} v^*_i & \text{if } \exists i, j : \left(i, j, v^*_i\right) \in x^b_{\searrow} \\ v_i & \text{otherwise} \end{cases}$$

$$\delta_{\mathrm{ext},?}\colon R = (r_1, \ldots, r_{|R|}) \mapsto (r_1, \ldots, r_{|R|}, r^*_1, \ldots, r^*_{|x^b_?|})$$

with $\left\{\, r^*_1, \ldots, r^*_{|x^b_?|} \,\right\} = x^b_?$.

$$\begin{aligned} \delta_{\mathrm{ext},\emptyset}\colon I^* &\mapsto \left\{\, i \,\middle|\, (i) \in x^b_{\emptyset} \,\right\}, \\ I &\mapsto I \cup \left\{\, i \,\middle|\, (i) \in x^b_{\emptyset} \,\right\} \end{aligned}$$

The port input bags are $x^b_{\mathbf{e}} = \left\{ e \;\middle|\; (\mathbf{e}, e) \in x^b \right\}$.

$$\begin{aligned}\delta_{\text{match}} \colon A^* &\mapsto \{ (l_k, i_k, \bigcirc_k, \square_k) \}_{k=1}^{n}, \\ R = (r_1, \dots, r_{|R|}) &\mapsto (r_{n+1}, \dots, r_{|R|}),\end{aligned}$$

with $n = \min\{ |R|, |I| \}$, $R = ((l_k, \bigcirc_k, \square_k))_k$, $i_k = \arg\min_{i \in I \setminus \{ i_1, \dots, i_{k-1} \}} d(v_i, \bigcirc_k)$ for $k \in \{ 1, \dots, n \}$.

CONFLUENT TRANSITION Handle internal transition first, and then external:

$$\delta_{\text{con}} = \delta_{\text{ext}} \circ \delta_{\text{int}}$$

EXAMPLE TRAJECTORY Let the transport space be a ring graph with 6 vertices, labelled 10 to 15. The distance between any two adjacent vertices shall be 1. Table 6.13 is an example of an input/output trajectory of such a dispatcher.

6.5 THE PYD3T LIBRARY

6.5.1 About pyd3t

pyd3t is a working computational implementation of the D3T Specification introduced in the previous Sections. It is a Python package that provides to the D3T simulationist all components, spaces and models to define her own model in a modular, toolbox-like fashion, as well as a DEVS simulator and interfaces to run these models and record and visualize their trajectories. Furthermore, pyd3t is extensible by design: the user is free to specify her own modules. In particular, I intended it to provide some basic dispatchers for testing, basic models and educational purposes, and for users to implement their own dispatchers. This extensibility is purely abstract: all the stock modules pyd3t ships are equivalent to any user-defined module as they use the same public interface.

Under the hood, pyd3t depends on an abstraction layer that provides the DEVS functionality, to which pyd3t adds the domain logic. This abstraction layer is the pydevs Python package. pydevs is a Python package that provides abstract DEVS base classes and simulation functionality as a Pythonic interface to the C++ adevs library. [146]

As of version 0.2, pyd3t consists of up to about 6,500 single lines of Python code, with up to about 7,500 single lines of additional Python testing code covering the codebase. The Appendix lists the implementations of the load source, the transporters, and the example dispatcher (Listing C.2, Listing C.3, Listing C.4).

6.5.2 An example simulation

To finally put the D3T framework and pyd3t into practice, we consider an example D3T model. The transport space is a directed circle—the S^1

Table 6.13: Example input/output trajectory of a myopic taxi FCFS-nearest transporter dispatcher. The jobs are $\tilde{\mathbf{j}}_l = ((\emptyset, \bigcirc_l, \emptyset), (\{l\}, \square_l, \{l\}))$.

t	c	y^b	x^b	ta
				∞
0	0		$\{(\bigstar,(2,10)), (\bigstar,(4,10)), (\emptyset,(2)),(\emptyset,(4)), (?,(1,11,13))\}$	0
0	1	$\{(+,(a,\tilde{\mathbf{j}}_1,0)),(\checkmark,(1,a)),(\circlearrowright,(a)),(\bullet,(b))\}$		∞
1	0		$\{(\searrow,(a,\cdot,11))\}$	∞
2.5	0		$\{(?,(3,14,15))\}$	0
2.5	1	$\{(+,(b,\tilde{\mathbf{j}}_3,0)),(\checkmark,(3,b)),(\circlearrowright,(b))\}$		∞
3	0		$\{(\searrow,(a,\cdot,13)), (\emptyset,(a))\}$	0
3	1	$\{(\bullet,(a))\}$		∞
4.5	0		$\{(\searrow,(b,\cdot,14))\}$	∞
5.5	0		$\{(\searrow,(b,\cdot,15)), (\emptyset,(b))\}$	0
5.5	1	$\{(\bullet,(b))\}$		∞
6	0		$\{(?,(5,10,12))\}$	0
6	1	$\{(+,(b,\tilde{\mathbf{j}}_5,0)),(\checkmark,(5,b)),(\circlearrowright,(b))\}$		∞
6.5	0		$\{(?,(7,13,11))\}$	0
6.5	1	$\{(+,(a,\tilde{\mathbf{j}}_7,0)),(\checkmark,(7,a)),(\circlearrowright,(a))\}$		∞
6.5	2		$\{(\searrow,(a,\cdot,13))\}$	∞
7	0		$\{(?,(9,15,13)), (\searrow,(b,\cdot,10))\}$	∞
8	0		$\{(?,(17,10,15))\}$	∞
8.5	0		$\{(\searrow,(a,\cdot,11)), (\emptyset,(a))\}$	0
8.5	1	$\{(+,(a,\tilde{\mathbf{j}}_9,0)),(\checkmark,(9,a))\}$		∞
9	0		$\{(\searrow,(b,\cdot,12)), (\emptyset,(b))\}$	0
9	1	$\{(+,(b,\tilde{\mathbf{j}}_{17},0)),(\checkmark,(17,b))\}$		∞
10.5	0		$\{(\searrow,(a,\cdot,15))\}$	∞
11	0		$\{(\searrow,(b,\cdot,10))\}$	∞
12	0		$\{(\searrow,(b,\cdot,15)), (\emptyset,(b))\}$	0
12	1	$\{(\bullet,(b))\}$		∞
12.5	0		$\{(\searrow,(a,\cdot,13)), (\emptyset,(a))\}$	0
12.5	1	$\{(\bullet,(a))\}$		∞

"sphere", on which transporters can only travel in one direction. The circumference of the circle is 1. Loads arrive according to a Poisson process with rate 1.75 in time, with origins and destinations independent and identically distributed uniformly on the circle, such that the average travel time is $\frac{1}{2}$. There are 2 transporters, starting at the same point. Loads are assigned to transporters according to the Myopic Taxi First-Come-First-Serve Nearest Transporter dispatching policy. We simulate the system for 1000 time units. In the following, we produce the code necessary to define this D3T model with pyd3t, its stock modules, and to run the simulation with pyd3t.

First, import the necessary Python packages:

```
import numpy as np
import pandas as pd

import d3t
from d3t.d3tsystem import D3TSystem
from d3t.dispatchers import MyopicTaxiFCFSNearestTransporterDispatcherModel
from d3t.space import DirectedS1
from d3t.observers import RawDataObserver
from d3t import statistics
```

Next, define the D3T model:

```
def load_generator_directeds1_uniform(rate=1.0):
    rng = np.random.RandomState(seed=42)
    while True:
        yield (
            rng.exponential(scale=1.0 / rate),
            rng.uniform(0.0, 1.0), rng.uniform(0.0, 1.0)
        )

space = DirectedS1()
representation = d3t.Representation(space=space)
observer_models = [
    (RawDataObserver, {'space': space}),
]

d3ts = D3TSystem(
    space=space,
    load_generators=[load_generator_directeds1_uniform(rate=1.75)],
    transporter_num=2,
    transporter_positions=2*[0.0, ],
    dispatcher_model=MyopicTaxiFCFSNearestTransporterDispatcherModel,
    dispatcher_initialization=dict(space=space),
    observer_models=observer_models,
)
```

Run the simulation:

```
until = 1000.0
d3ts.execute_until(until)
obs = d3ts._observers[0]
```

Extract recorded statistics into standard pandas dataframes for further analysis:

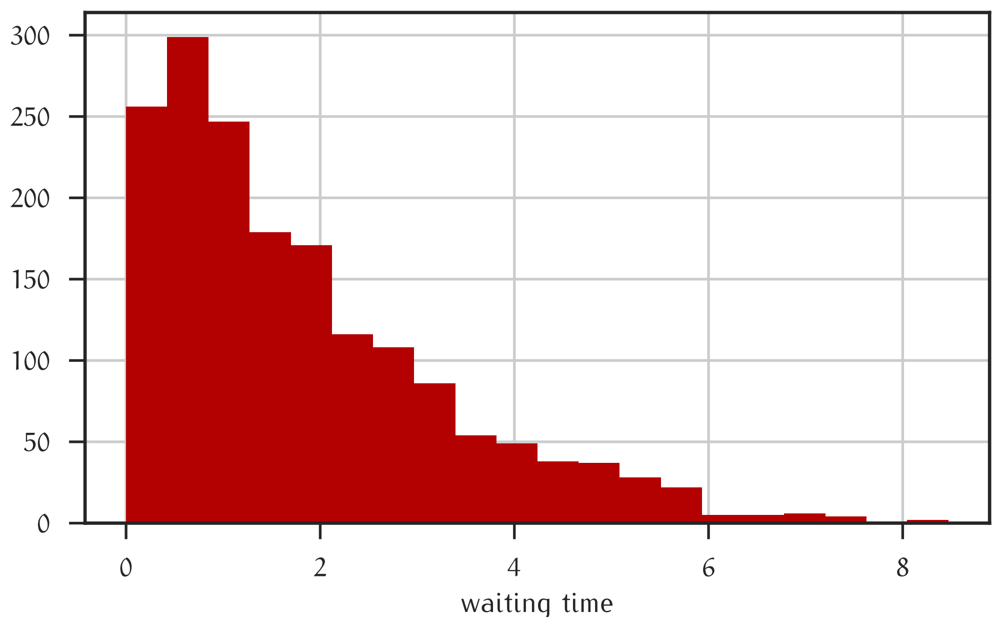

Figure 6.14: Histogram of the waiting times of a D3T example simulation.

```
raw_loads = pd.DataFrame(
    data=list(obs._loads.values()),
    index=list(obs._loads),
    columns=obs.LOAD_COLUMNS)
raw_transporters = pd.DataFrame(
    data=list(obs._transporters.values()),
    index=list(obs._transporters),
    columns=obs.TRANSPORTER_COLUMNS)
raw_jobs = pd.DataFrame(
    data=list(obs._jobs.values()),
    index=list(obs._jobs),
    columns=obs.JOB_COLUMNS)
raw_idleperiods = pd.DataFrame(
    data=obs._idleperiods, columns=obs.IDLEPERIOD_COLUMNS)
```

Compute statistics with pyd3t statistical routines:

```
loads = statistics.compute_loads(raw_loads, raw_jobs)
```

Plot a histogram of the waiting times (Figure 6.14), a bar chart of the frequencies of the system sizes upon arrivals of requests (Figure 6.15), and the total share of time 0, 1 or both transporters are busy (Figure 6.16):

```
loads['waiting time'].hist()
statistics.compute_arrival_system_size_frequencies(loads).plot()
statistics.compute_busy_transporter_number_time(
    statistics.compute_busy_transporter_number_trajectory(
        statistics.compute_transporter_busy_idle_periods(
            raw_idleperiods, until), until))
```

6.6 DISCUSSION

In this Chapter, I have introduced a formal domain-specific language to model demand-driven directed transport systems, and a Python package

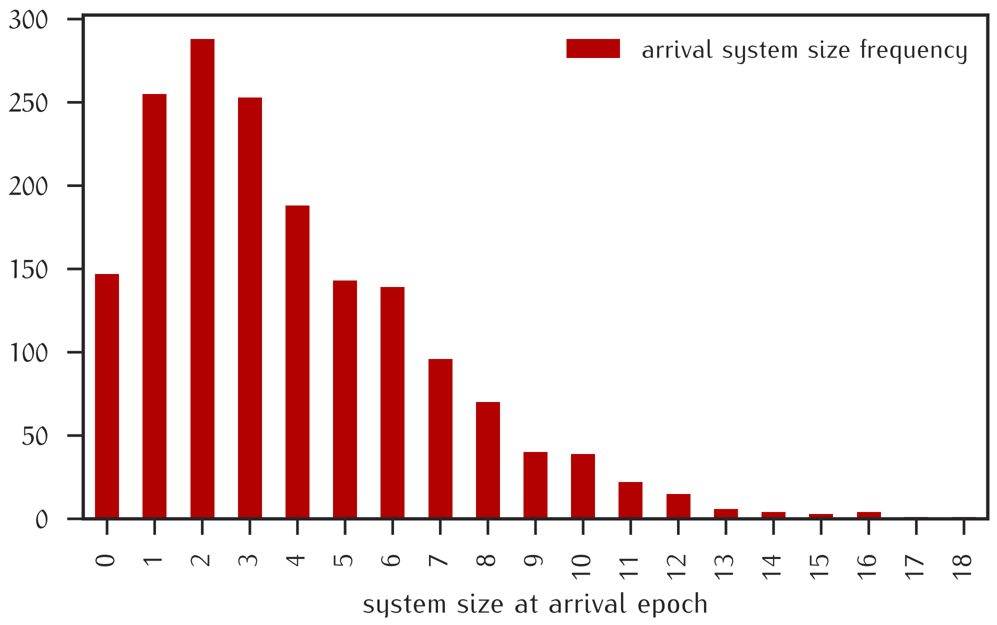

Figure 6.15: Frequencies of the system sizes at arrival epochs of a D3T example simulation.

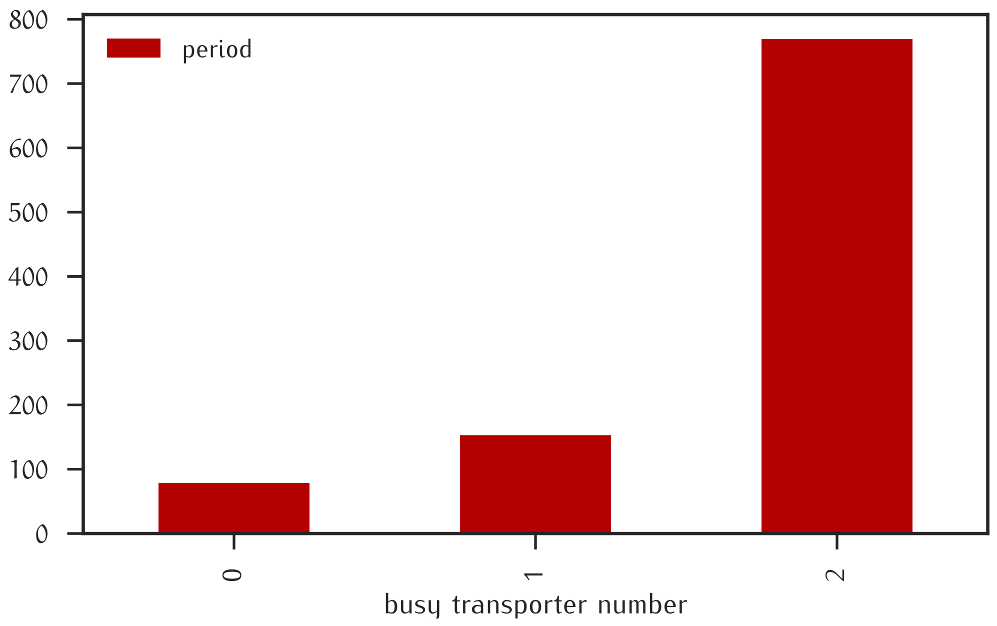

Figure 6.16: Total duration of busy transporter number periods in a D3T example simulation.

to implement and simulate these models. As the underlying discrete-event framework I chose the Discrete-Event System Specification (DEVS) for its formal theory, abstraction, modularity, comprehensiveness and proven and well-tested implementation as a C++ library. There are other frameworks to simulate discrete-event systems, for example, the Python simpy package, that do not offer such a formal background, and are typically implemented in one specific programming language. DEVS is implementation-agnostic and hence provides for specifying D3T models independent of their computational implementation, increasing abstraction and hence, conciseness and robustness of models in separating the different functional layers of formal definition, translation to a algorithmic implementation, and actual execution of the simulation of a model. This enhances reproducibility of computational studies of D3T systems carried out within the D3T framework. It also enables the D3T researcher to focus on what matters in their particular study. For example, if one wants to investigate the effects of different transport geometries or of different spatiotemporal request patterns on D3T system performance, pyd3t already provides the event-based D3T transport mechanism and hides all the detail that would amount to a lot of boilerplate code needed to be implemented and tested separately for each study. Abstraction and modularity ensures and enforces that components such as dispatching policies written for one particular study can be reused for other transport spaces. The main reason why the D3T framework seems rather technical is that this is precisely what is was designed for: to encapsule technical detail in a well-defined and well-tested framework, such that the user does not need to bother about it any more in individual computational studies. As we saw in the pyd3t example, just a few lines of code suffice to produce rich dynamics and statistics.

pyd3t is a prototypical implementation of the D3T framework. While it is ready for production, there are caveats that one needs to keep in mind. For example, the only abstraction the D3T framework and pyd3t provide for the transport space so far is the low-level function of metric distance. As spatial indexing becomes increasingly imperative for large-scale studies with a large number of pending requests, or a large number of nodes in a network, this is a computational bottleneck. For example, in Euclidian geometries each call to the distance function invokes a square root operation. A higher-level interface could alleviate this, such as providing an interface to spatial indexing and querying. Each transport space in turn would need to implement that interface, but could always resort to a fall-back dummy implementation that just evokes the distance function.

Both the D3T framework and pyd3t so far do not feature requests with individual time-windows (such as pick-up after a certain time, or delivery before a certain time). Formally, it is easy to extend the request event definition with the respective data. It is the dispatching policies that need to process and adher to these time windows.

Typically, the dispatching policy involves the most complicated and technical part of any D3T model implementation. Specifying and implementation dispatching policies comes with a steep learning curve. Compared to a vanilla implementation, the D3T framework surely adds further technicalities. However, D3T compatibility enables reusability in other D3T models and reduces development time—for example, the transporter is already implemented in pyd3t, and statistics and visualization also come for free.

7 CONCLUSION

In this Part of my Thesis, I have proposed and detailed a common high-level language to model and simulate demand-driven directed transport systems. The D3T framework and its Python implementation pyd3t encapsule and hide the technicalities of discrete-event systems and the domain-specific logic of D3T models behind a high-level modular interface. This enables focussed and concise computational studies of D3T systems. Furthermore, the common framework and codebase facilitate collaboration and building upon others' contributions. Perhaps the most immediate illustration of these principles is the fact that Marc Timme's Network Dynamics Group at MPI for Dynamics and Self-Organization has been assembling a team of domain scientists to both advance the D3T approach and to employ the framework in computational studies of collective mobility systems. To further reinforce this approach, pyd3t is scheduled to be released as free and open source software in due process.

The D3T framework contains basic models for myopic taxi dispatching policies. These policies transport only one load at a time, and they do neither take into account future requests, nor currently busy transporters which will become idle in the near future, when assigning requests to transporters. Further work needs to be done to implement forward-looking disciplines. For example, these are disciplines that proactively reject requests if the individual service quality is projected to be too low. Given the interest in collective mobility and on-demand ride-sharing systems, the immediate task at hand is to implement ride-sharing disciplines such as those proposed by Santi, Resta, Szell, Sobolevsky, Strogatz, and Ratti [34] and, in particular, Alonso-Mora, Samaranayake, Wallar, Frazzoli, and Rus [36]. The D3T team currently implements these dispatching policies in pyd3t extension modules.

Another intriguing extension of the D3T framework and pyd3t that would unlock another avenue of future research is to implement feedback of the transport dynamics onto the transport space. The current specification assumes no such interaction. Basically, the nodes and links in a network transport space have infinite capacity. In human mobility on street networks, the effect of individual motorized vehicles and the congestion they cause are represented by the average travel time along the respective link. (A first extension of the framework is to allow transport spaces with time-dependent travel times to allow for intraday variation.) A negative feedback could model finite capacities of links when D3T transporters dominate transport along those links. On the other hand, a positive feedback could model reinforcement, for example, when heavily used transport links get upgraded in man-made or in biological systems (ant trails, slime molds).

To conclude, D3T and pyd3t have already started facilitating reproducible computational studies of collective mobility and transport systems. I envision them to do even more so in the future, when both pyd3t and the first studies have been published.

Part IV

Temporal Percolation in Critical Queues

ABSTRACT

Percolation theory characterizes the growth of discrete-unit systems upon the addition of connections. How spatial systems and networks percolate to become extensively connected reveals essential features of the underlying growth process. In this Part I introduce *temporal* percolation to describe the growth of return times in stochastic dynamical systems such as random walks and queueing processes. I find that the critical point of the percolation is exactly at the boundary between recurrent and transient dynamics. Intriguingly, and in contrast to one-dimensional spatial percolation, finite clusters persist even beyond the critical point. Finite-size scaling analysis in the temporal dimension reveals where the system becomes unstable and the critical exponents of the transition. These results establish a paradigm for linking percolation phase transitions to instabilities of stochastic processes.

8 INTRODUCTION

Queues exist wherever several individual subject or objects demand exclusive service from a shared resource. Queueing theory links individual demand patterns to collective resource usage and performance predictions when these demands are stochastic in nature and contend for limited server resources. [147] While this theory of stochastic dynamical systems originated from Erlang's study of the operation of telephone exchanges at the beginning of the 20th century, [148] queues form in all kinds of socioeconomic, technical and biological systems such as in demand-driven public transport, [38] Internet data packet routing, [149] or gene expression. [150, 151] As in the long run, stable queueing systems typically accommodate all arriving requests, queues intermittently form due to the stochastic variability of the arrival and service processes. If, however, overall demand exceeds the throughput capacity, the system congests and ceases to function properly. The capacity is arguably the most fundamental steady-state performance measure of any given queueing system. It signifies the critical load below which the system is asymptotically stable, and above which congestion occurs almost surely. Queueing systems typically operate in a dynamical regime that balances individual demand for quality service and collective demand for efficient use of the available resources known as the quality-and-efficiency-driven (QED) regime first described by Halfin and Whitt. [152, 153] This regime is still subcritical (stable, uncongested) in finite queueing systems, but operates close to criticality. In fact, the larger the queueing system is, the closer the QED regime is to the critical point, eventually reaching it in the limiting case of a system with an infinite number of servers. [154] Studying the dynamics of these systems and informing their design for optimized system performance and individual utility henceforth calls for a methodology to determine the throughput capacity, charting the transition to congestion and identifying scaling behavior und universal properties.

Queueing theory excels at the analytical and numerical description of the performance measures of a queueing system at hand, depending on the utilization of the server resources. Naturally, it is concerned with queues under heavy load when ressources are well utilized while still providing satisfactory quality to the individual user. There are analytical expressions for the distributions of individual waiting times and system busy periods in simple queueing models where the critical load is known. [147, 155]

Yet, when it comes to more intricate queueing models, queueing theory lacks a general dynamical notion of a critical transition towards congestion. It hitherto does not treat the *structural* transition from stability to instability when increasing the arrival rate towards and beyond the critical point of a queueing system. Even more so, in numerical studies of queueing systems the simulated system at hand typically eludes such analytical treatment. Practitioners typically resort to applying a drift criterion to measure congestion in simulation. [128, 156] While queueing theory delivers general results for systems under heavy load, the singularities at the critical point, and how dynamical performance measures scale towards it with system parameters, are typically avoided in these treatments. This hampers a thorough dynamical understanding of the collective phenomenon of congestion, how it emerges from initially unsuspicious individual interaction such as the control policy of the servers, and which universal principles queueing models adher to irrespective of such detail. Understanding precursors helps to mitigate or avert imminent overload of such a system and to keep it in a stable operation regime.

Furthermore, computer simulations *a priori* do not allow to infer ensemble criticality from a sample of simulated trajectories. In fact, given a queueing or any other stochastic dynamical system, it is an open question how to infer its stability or instability from a number of such trajectories. Computer simulations draw sample paths of queues or other stochastic dynamical systems. While a full realization of such a path would stretch over the whole time axis from the origin to infinity, a simulation in finite time only produces trajectories as finite subsets of such paths. Given a single trajectory or a sample of trajectories of simulated queues or other stochastic dynamical systems in finite time, it is impossible to decide whether the system eventually overloads (becomes unstable) or not. Just as it is impossible to decide from simulating a percolation setting on a finite lattice whether the infinite system percolates almost surely or not. Specifically, when the dynamics is externally driven away from stability, the question is how to characterize and to pinpoint the transition to instability.

In the following, I develop a methodology to address these open problems. In essence, I augment queueing theory with statistical physics by mapping stochastic dynamical systems to a surprisingly well-known percolation setting in the paradigm of *temporal percolation*. For clarity and conciseness, I resort to the simplest model of a stochastic process relevant to the discussed phenomena: a random walk. Chapter 9 condenses the necessary background from the theory of stochastic processes and critical transitions. Chapter 10 introduces the temporal percolation paradigm. Chapter 11 assembles the theoretical and computational set of tools to study stochastic dynamical systems in the critical region, while Chapter 12 details and demonstrates our methodology with the random walk. Section 12.5 critically discusses the method before Chapter 13 lays out avenues of future research in the context of this Thesis and concludes.

9 FUNDAMENTALS

9.1 STABILITY OF STOCHASTIC DYNAMICAL SYSTEMS

Ever since Poincaré formulated his theorem, recurrence has been seen as a pervasive characteristic of dynamical systems. A dynamical system with volume-preserving flow and bounded orbits (that is, with finite invariant measure), returns for almost every initial condition in an open set to that open set infinitely often. [157] The dynamics of a time-discrete dynamical system on a measure space S is given by a measure-preserving transformation $T : S \to S$. The first return time to a set B in phase space S is defined as

$$\tau_B(x) = \inf\{\, n > 0 : T^n(x) \in B \,\}. \tag{9.1}$$

In hyperbolic and other systems, these return times have an exponential limit distribution, with successive return times being independent. [158–160] Furthermore, the recurrences already suffice to describe the long-term dynamics. [161] In particular, recurrences characterize deterministic chaotic dynamics as well as stochastic systems. [162–164]

Recurrence is one of the fundamental notions of stability of a dynamical system. As for nonlinear dynamical systems, for stochastic systems there are several concepts and methods to define and determine stability. [165] For example, the drift criterion measures the average change (the "drift") in some norm or Lyapunov function over time. In the one-sided random walk or in queueing systems, average zero drift signifies stability, whereas positive drift means the random walk or queue length diverges. The drift criterion is well-developed for Markov chains. [128, 165] Another notion of stability of stochastic systems are renovating events. These are points in time at which the process decouples from its past, that means it does not depend on states before that time any more. [166] In particular, recurrence is a weak notion of stability in Markov chains, and transience is a strong notion of instability, as the process almost surely diverges and never returns. [128] In fact, Markov chains exhibit a strict dichotomy: Under mild conditions, each Markov chain is either recurrent or transient.

A theoretical and practical question for any such system at hand is: is it stable or unstable – recurrent or transient? [167] In particular, we are interested in how tuning a system parameter (the driving) let the stochastic system turn unstable. In fact, the transition from stability to instability, or from recurrence to transience, constitutes a phase transition, whose critical point and exponents we want to determine. [82, 167, 168] As a stochastic process evolves in discrete time, we are going to conceive it as a one-dimensional system with sites representing the discrete points in time. Sites between sucessive returns form clusters, subject to percolation when the system turns transient and an infinite return period forms, just as infinite avalanches form in overcritical extremal models. [169] Phase transitions in one-dimensional spatial models have been well understood. [86, 170, 171]

We choose Markov chains on a countable space to introduce the temporal percolation paradigm and to develop the methodology. Markov chains are well understood, and at the same time general enough to model a plethora of stochastic phenomena. [127] Let X_n be a Markov chain on a countable state space S, and without loss of generality, let X_n be irreducible. Then X_n is either positive recurrent, that is, the probability to visit a set infinitely often is 1, the probability to visit a set at all is 1, and the expected return time is finite. Or, the chain X_n is null-recurrent, as the expected return time grows infinite. Finally, the chain X_n can be transient, when the probability to return to a set at all is less than 1 – such that the probability to not return at all any more is finite; consequently, the probability to return an infinite number of times is zero: almost surely, the chain will diverge. It is instructive to consider a (general) stochastic recursive sequence, [129] where the driving sequence represents the (stochastic) drive away from some recurrent set with some parameter ρ:

$$X_n = f(X_{n-1}, \xi_n, \rho) \tag{9.2}$$

Without loss of generality, let us assume the countable state space to be $\mathbb{N}$, such that transience is signified by diverging towards $+\infty$. In this case, we postulate that

$$\frac{\partial}{\partial \rho} f(x, \xi, \rho) > 0.$$

Hence, given a Markov chain and its strict dichotomy, there shall be a critical point ρ_c at which the chain turns null-recurrent and beyond which the chain becomes transient. We can think of ξ as capturing the parameter-independent drive, which basically could be that ξ_n is a sequence of random numbers, while f embodies the functional impact of the drive on the state X_n. This is consistent with using Common Random Numbers, or regarding the whole sequence as one random element in S^∞ which remains the same for different values of the external parameter ρ.

9.2 THE RANDOM WALK

9.2.1 Definition

The random walk is an ideal system to introduce and study temporal percolation. Specifically, the one-sided random walk, i. e. the random walk on $\mathbb{N}$, is a simple model with rich behavior. It is a Markov chain and as such, features the recurrence-transience dichotomy and the postulated temporal percolation transition. Furthermore, an exact combinatorial expression for its return-time distribution is available. The stochastic limit process of the one-sided random walk is reflected Brownian motion. [154] As such, it is a cornerstone for modelling queueing phenomena and flow systems in heavy traffic as well as physical phenomena which involve reflected Brownian motion. [154, 172, 173]

To begin with, let us define the random walk on the half-line:

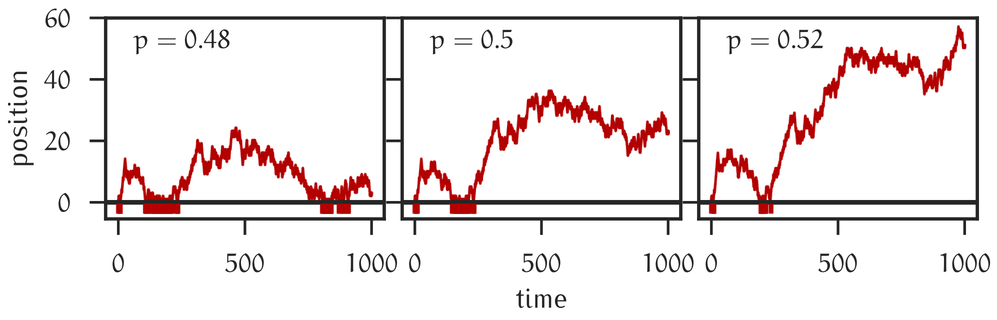

Figure 9.1: Typical trajectories of the random walk on the half line in the positive recurrent regime ($p = 0.48$), in the null recurrent regime ($p = \frac{1}{2}$), and in the transient regime ($p = 0.52$). All trajectories are drawn from the same realization (using common random numbers). Vertical bars in the lower panels highlight returns to the origin.

Definition 9.1. *Let* $p \in [0, 1]$ *and let* X_n *be the Markov chain on* $\mathbb{N}$ *with initial distribution concentrated at the origin, i.e.* $\mu(0) = 1$ *and hence,* $X_0 = 0$ *almost surely. Let the transition probabilities be*

$$\begin{aligned} P(x \to x+1) &= p \\ P(x \to x-1) &= 1-p && x > 0 \\ P(0 \to 0) &= 1-p \\ P(x \to y) &= 0 && \textit{otherwise.} \end{aligned}$$

Then X_n *is a* biased random walk on the half-line with parameter p *starting at the origin.*

The random walk is irreducible and hence exhibits the recurrence–transience dichotomy.

9.2.2 Recurrence and transience in finite-time trajectories

Let us consider typical trajectories of the random walk on the half-line for various parameters in the recurrent and transient regimes (Figure 9.1). While a typical trajectory in the positive recurrent regime frequently returns to the origin, a typical transient trajectory approximates linear growth with no return, with possibly a few returns to the origin in the beginning before eventually diverging. At first glance, it seems not to be difficult to tell these regimes apart. However, in finite simulation time, different realizations typically feature trajectories that counter-intuitively seem to belong to the transient regime for $p < \frac{1}{2}$ and others that seem to belong to the recurrent regime for $p > \frac{1}{2}$ (cf. Figure 9.2). When inspecting a number of finite-time trajectories at the critical point, the attempt to sharply classify a given system as recurrent or transient seems futile (cf. Figure 9.3). This becomes clear in particular when inspecting a specific finite-time trajectory, that may seem to embark on a long – possibly infinitely long – excursion away from the origin, only to return within a larger time window (cf. Figure 9.4).

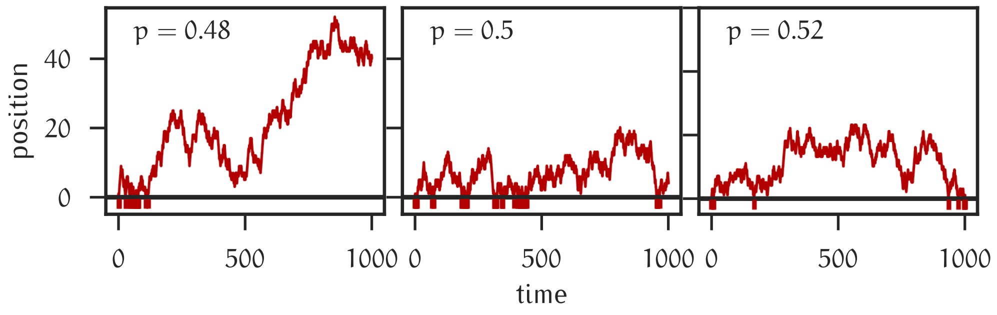

Figure 9.2: Counter-intuitive trajectories of the random walk on the half line in the positive recurrent regime ($p = 0.48$), in the null recurrent regime ($p = \frac{1}{2}$), and in the transient regime ($p = 0.52$). Trajectories are drawn from different realizations (different random numbers). Vertical bars in the lower panels highlight returns to the origin.

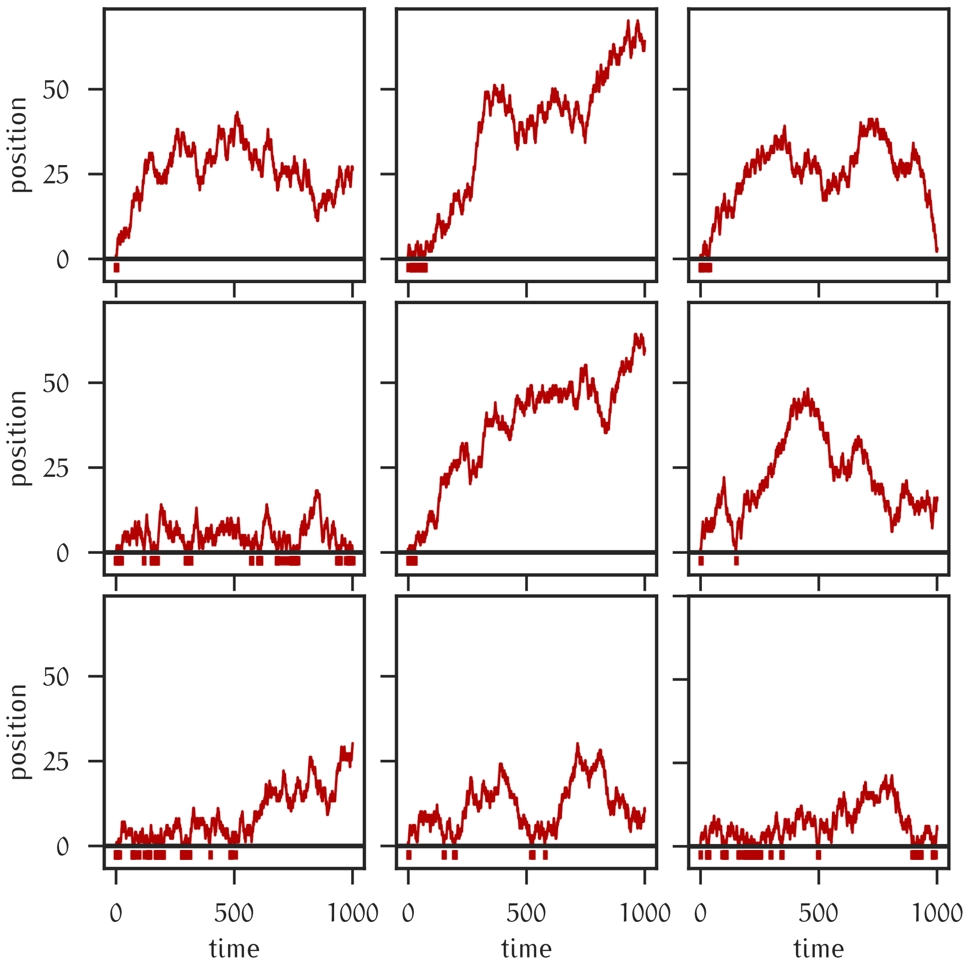

Figure 9.3: Typical trajectories of the critical random walk on the half line ($p = \frac{1}{2}$). Vertical bars in the lower panels highlight returns to the origin.

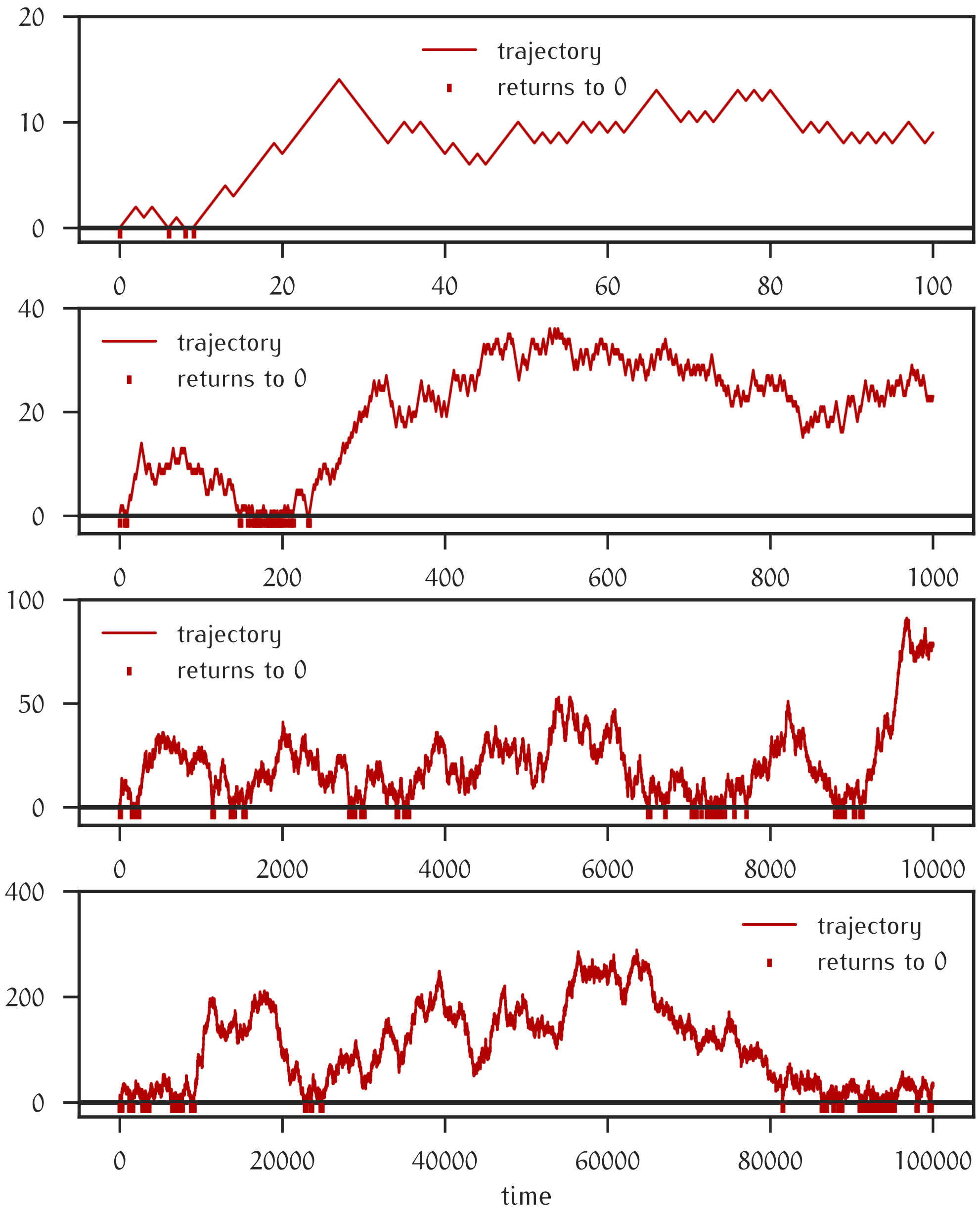

Figure 9.4: A single realization of the critical random walk on the half line ($p = \frac{1}{2}$) in successive 10 times larger time windows. Vertical bars in the lower panels highlight returns to the origin.

9.2.3 Analytical return time distribution

Let us consider the probability that the biased random walk X_n on the half-line with parameter p returns to the origin $\{0\}$ after n steps when starting at the origin, and th

$$f_n(p) \equiv f_n(0,0) P\{X_n = 0, X_{n-1} \neq 0, \ldots, X_1 \neq 0 \mid X_0 = 0\} \tag{9.3}$$

We have $f_1(p) = P(0 \to 0) = 1 - p$. For $n > 1$ observe that

$$f_n(p) = p f_{n-1}(1,0)$$

where $f_k(1,0)$ is the first-passage-time probability from 1 to 0, i. e. the probability to reach the origin when starting at 1 in exactly k steps. Note that this is zero for an even number of steps. For an uneven number of steps, we further have

$$f_{2k+1}(1,0) = (1-p)p^k(1-p)^k C_k$$

as a path of length $2k+1$ from 1 to 0 is a loop of length $2k$ from 1 to 1 without touching 0, and a final step from 1 to 0. The number of loops from 1 to 1 of length $2k$ without touching 0 is the number of paths that consist of k steps to the right $(+1)$ with probability p and k steps to the left (-1) with probability $1-p$, but with no initial segment of the path that has more steps to the left than to the right. This is exactly the number of Dyck paths of length $2k$, such that the integer coefficients C_k are the Catalan numbers [174–176] with

$$C_n = \frac{1}{n+1}\binom{2n}{n}. \tag{9.4}$$

The Catalan numbers follow the recursion

$$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n \tag{9.5}$$

and asymptotically grow as

$$C_n \sim 4^n n^{-3/2} \quad (n \to \infty). \tag{9.6}$$

Their generating function is [174]

$$C(x) = \sum_{n=0}^{\infty} C_n x^n = \frac{2}{1+\sqrt{1-4x}} = \frac{1-\sqrt{1-4x}}{2x} \qquad \left(0 < |x| \leqslant \frac{1}{4}\right) \tag{9.7}$$

and $C(x) = 1$ for $x = 0$. As $x = p(1-p)$ and $p \in [0,1]$, we have $0 < x \leqslant \frac{1}{4}$ for $p \in (0,1)$ and $x = 0$ for $p \in \{0,1\}$.

Hence, the return time distribution is

$$f_n(p) = \begin{cases} 1-p & n = 1 \\ C_{k-1} p^k (1-p)^k & n = 2k, k \in \{1,2,\ldots\} \\ 0 & \text{otherwise,} \end{cases} \tag{9.8}$$

with the recursion $f_1(p) = 1-p$, $f_2(p) = p(1-p)$ and

$$f_{n+2}(p) = \frac{n-1}{n+2} 4p(1-p) f_n(p) \quad n = 2,4,6,\ldots \tag{9.9}$$

The return time distribution asymptotically approaches

$$f_n(p) \sim n^{-3/2}(4p(1-p))^{n/2} \quad (n \to \infty) \tag{9.10}$$

for even n. The return probability is

$$\begin{aligned} L(p) \equiv L(0,0) = F_\infty(0,0) &= \sum_{n=1}^{\infty} f_n(p) \\ &= 1-p+p(1-p)\sum_{k=0}^{\infty} C_k p^k (1-p)^k. \end{aligned}$$

With $x = p(1-p)$ we have $L(p) = 1-p+p(1-p)C(x)$ and hence,

$$L(p) = \frac{3}{2} - p - \left|p - \frac{1}{2}\right| = \begin{cases} 1 & p \leqslant \frac{1}{2} \\ 2(1-p) & p \geqslant \frac{1}{2}. \end{cases} \tag{9.11}$$

The average return time given that the walk returns is

$$\bar{\tau} = \frac{1}{L(p)} \sum_{n=1}^{\infty} n f_n(p) = \frac{1-p}{L(p)} \left(1 + 2p \frac{d}{dx}(xC(x))\right)$$

$$\Rightarrow \bar{\tau} = \frac{1-p}{L(p)} \left(1 + \frac{p}{|p-\frac{1}{2}|}\right) = \begin{cases} 1-p+\frac{p(1-p)}{|p-\frac{1}{2}|} & p \leqslant \frac{1}{2} \\ \frac{1}{2} + \frac{p}{2|p-\frac{1}{2}|} & p \geqslant \frac{1}{2}. \end{cases} \tag{9.12}$$

For $p \to 0$ we have $\bar{\tau} \to 1$ and for $p \to 1$ we have $\bar{\tau} \to \frac{3}{2}$. As $p \to \frac{1}{2}$, the average return time diverges.

9.3 THE M/M/1 QUEUE

9.3.1 Introduction

The M/M/1 queue is to queueing theory what the ideal gas is to thermodynamics. [127, 147, 177, 178] It is a stochastic model of a queue with 1 server that sequentially and exclusively processes jobs. The service times, i. e. the time it takes for the server to process a job, are iid according to an exponential distribution with parameter μ, the service rate. Jobs arrive to the system according to an arrival process t_n, where the random variable t_n denotes the arrival time of the n-th job. Conventionally, an initial 0-th job arrives at time $t_0 = 0$. The inter-arrival times $\Delta t_n = t_n - t_{n-1}$ are independent and identically distributed according to the exponential distribution with rate λ. Both inter-arrival times and service times are exponentially distributed and independent, hence the notation M/M/1 queue ("M" for memoryless).

If a job arrives to an empty system, as the initial job, the server starts processing it immediately and exclusively. The random service time X_n is the time it takes the server to process the n-th job. Upon completion of service, the job leaves the system for good. If a job arrives to a system with a busy server, it queues. Service commences as soon as the server has processed all preceding jobs.

It is a hallmark of queueing theory that the M/M/1 queueing system will reach a steady state if the arrival rate is smaller than the service rate ($\lambda < \mu$ or $\rho = \frac{\lambda}{\mu} < 1$). In contrast, if the arrival rate equals or exceeds the service rate, the queue will almost surely eventually grow infinitely long ($\lambda \geqslant \mu$ or $\rho \geqslant 1$). In the following, we will study the transition from the stationary regime ($\rho < 1$) towards the transient regime ($\rho > 1$). Without loss of generality, we will let the service rate define the time scale such that $\mu = 1$ and $\lambda = \rho$.

9.3.2 System size Markov chain

Let the random variable L_n denote the number of jobs in the system upon arrival of the n-th job. L_n is also called the *system size*. The system size excludes the arriving job, but includes a job in service, if any. Initially, when the 0-th job arrives at $t = 0$, the system is empty, and $L_0 = 0$. The system size L_1 at the arrival of the next job is either 1 or 0, depending on whether the interarrival time Δt_1 is smaller than the service time X_0 of the 0-th job or vice versa. We immediately observe that from one arrival to the next arrival, the system size at most increases by 1, but may drop all the way to 0. This depends on how many jobs the server finishes to process within the interarrival time.

Due to memorylessness of the exponential distribution, the discrete-time stochastic process L_n is a (homogeneous) Markov chain on the natural numbers $\mathbb{N}$. There are several equivalent ways to describe the time evolution of L_n. We consider deriving L_n by direct calculation from the original system (without explicitly making use of it being a Markov chain), and by explicitly stating the transition probabilities of the Markov chain L_n.

The time evolution of the M/M/1 queue is determined by the random interarrival times Δt_n and the random service times X_n. As before, we have the random arrival times $t_n = t_{n-1} + \Delta t_n$. We further consider the random departure time d_n of the n-th job. If the system is empty upon arrival of the n-th job, service commences immediately and we have

$$P\{\, d_n = t_n + X_n \mid L_n = 0 \,\} = 1.$$

Now, let us consider as an auxiliary random variable the nonnegative residual work W_n in the system at the time of arrival of the n-th job. If the system is empty, we have $W_n = 0$. If the system is not empty, W_n denotes the time it takes until the system finishes processing all previous jobs and starts processing the n-th job. The random time W_n is given by the *Lindley recursion* from queueing theory

$$W_0 = 0, W_n = (W_{n-1} + X_{n-1} - \Delta t_n)^+\,, \tag{9.13}$$

where $(x)^+ = x$ for $x \geqslant 0$ and $(x)^+ = 0$ for $x \leqslant 0$. Now, we have the departure times

$$d_n = t_n + W_n + X_n. \tag{9.14}$$

Finally, the system size at any given time t is the number of jobs that have arrived up to time t minus the number of jobs that have finished up to time t: $L(t) = |\{ i : t_i < t \}| - |\{ i : d_i < t \}|$. This translates to discrete time as

$$L_n = n - |\{ i : d_i < t_n \}| . \tag{9.15}$$

We now consider the transition probabilities $P(i \to j) = P\{ L_n = j \mid L_{n-1} = i \}$ of the system size Markov chain L_n. As by the very definition of an arrival the system size does not increase between two successive arrivals at t_{n-1} and t_n other than by the initial increase by 1 at t_{n-1}, we immediately have $P(i \to j) = 0$ for $j > i + 1$. System size decreases by the number N of jobs that the server finishes within the interarrival period $\Delta t_n = t_n - t_{n-1}$. This number N is a random variable that depends on Δt_n and on the system size L_n at the last arrival. In particular, it does not depend on when service started as the service time follows a memoryless exponential distribution. The random variable N ranges between 0 (no job finished) and $L_{n-1} + 1$ (all jobs finished, including the $(n-1)$-th job arriving at t_{n-1}).

Now, the probability $P(j \to k)$ for $j \geqslant 0, 1 \leqslant k \leqslant j+1$ that of the $j+1$ jobs present in the system (including the arriving job), exactly $n = j - k + 1$ jobs (with $0 \leqslant n \leqslant j$) have finished service before the next arrival is

$$\begin{aligned} P(j \to k) &= P\{ N = j - k + 1 \mid L = j \} \\ &= \int_0^\infty P\{ N = j - k + 1 \mid L = j, \Delta t = t \} f(t)\, dt, \end{aligned} \tag{9.16}$$

where $f(t) = \rho \exp(-\rho t)$ is the probability density function of the exponential interarrival time distribution with arrival rate $\lambda = \rho$. The server processes jobs sequentially with independent service times identically distributed according to the exponential distribution with rate $\mu = 1$. Hence, the number of jobs the server processes in a time interval t at rate 1 is Poisson distributed with mean t. Specifically, the probability that the server processes $n = j - k + 1 \leqslant j$ jobs in a time interval t at rate 1 is $\frac{t^n}{n!} \exp(-t)$. Therefore,

$$P(j \to k) = \frac{1}{n!} \rho \int_0^\infty t^n \exp(-(1+\rho)t)\ dt = \rho (1+\rho)^{-(j-k+2)} \tag{9.17}$$

for $1 \leqslant k \leqslant j+1$. The probability to process all remaining $j+1$ jobs is

$$P(j \to 0) = 1 - \sum_{k=1}^{j+1} P(j \to k) = (1+\rho)^{-(j+1)}. \tag{9.18}$$

This completes the transition probabilities (see also Figure 9.5) as

$$P(j \to k) = \begin{cases} (1+\rho)^{-(j+1)} & (k = 0) \\ \rho(1+\rho)^{-(j-k+2)} & (1 \leqslant k \leqslant j+1) \\ 0 & (k > j+1). \end{cases} \tag{9.19}$$

In particular, the probability to increase the queue is $P(j \to j+1) = \frac{\rho}{1+\rho}$ which is exactly $\frac{1}{2}$ at the threshold $\rho = 1$. This exposes a similarity to the random walk at $p = \frac{1}{2}$. (In fact, both M/M/1 queue and random walk on the half-line have Brownian motion as a limit process in the critical region under certain mild conditions. [154])

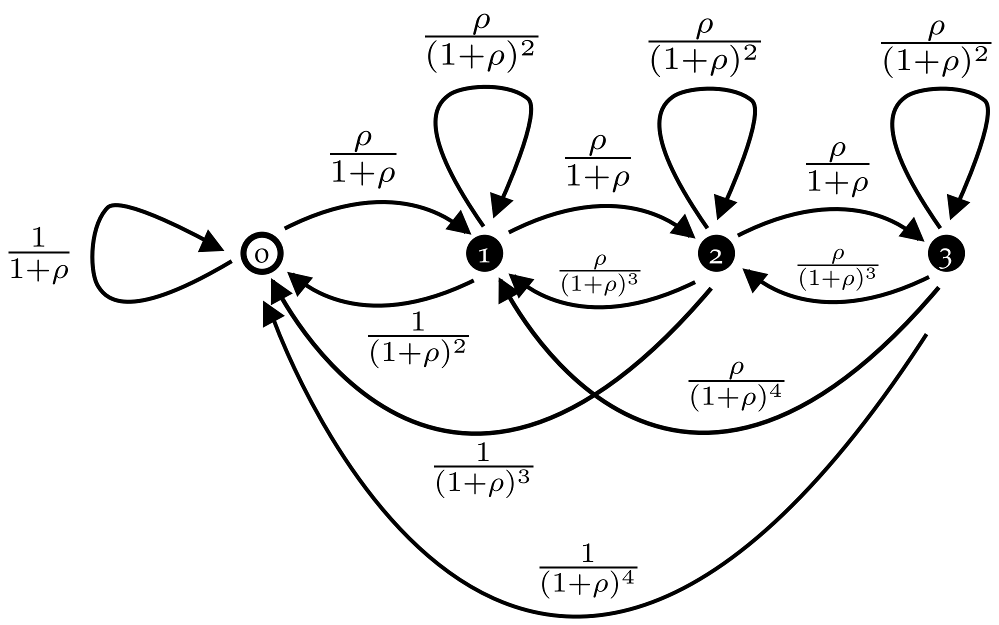

Figure 9.5: Markov chain of the system size L_n of an M/M/1 queue with parameter ρ depicted for system size up to 3, including the nonzero transition probabilities according to Equation 9.19.

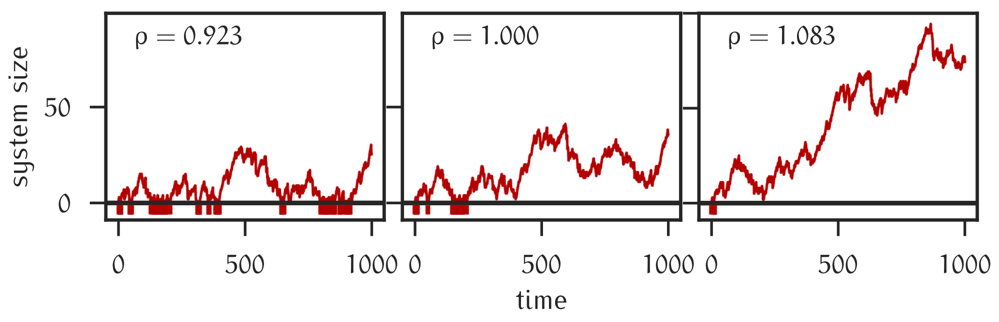

Figure 9.6: Typical trajectories of the M/M/1 queue system size Markov chain in the steady-state regime ($\rho = 0.923$), at full capacity ($\rho = 1$), and in the overloaded state ($\rho = 1.083$). All trajectories are drawn from the same realization (using common random numbers). Vertical bars in the lower panels highlight returns to the origin.

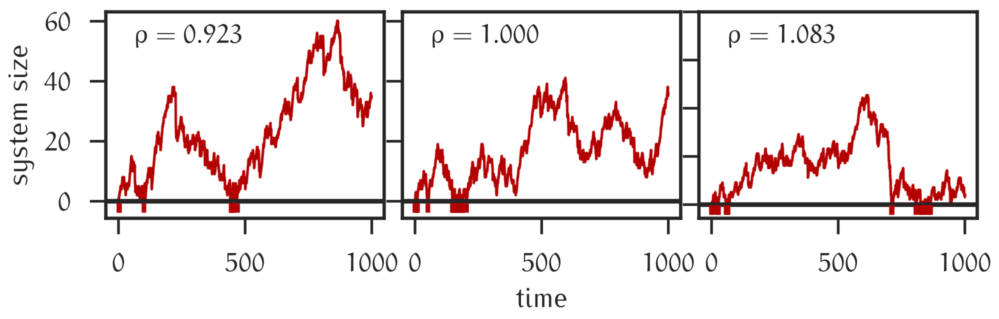

Figure 9.7: Counter-intuitive trajectories of the M/M/1 queue system size Markov chain in the steady-state regime ($\rho = 0.923$), at the threshold ($\rho = 1$), and in the overloaded state ($\rho = 1.083$). Trajectories are drawn from different realizations (different random numbers). Vertical bars in the lower panels highlight returns to the origin.

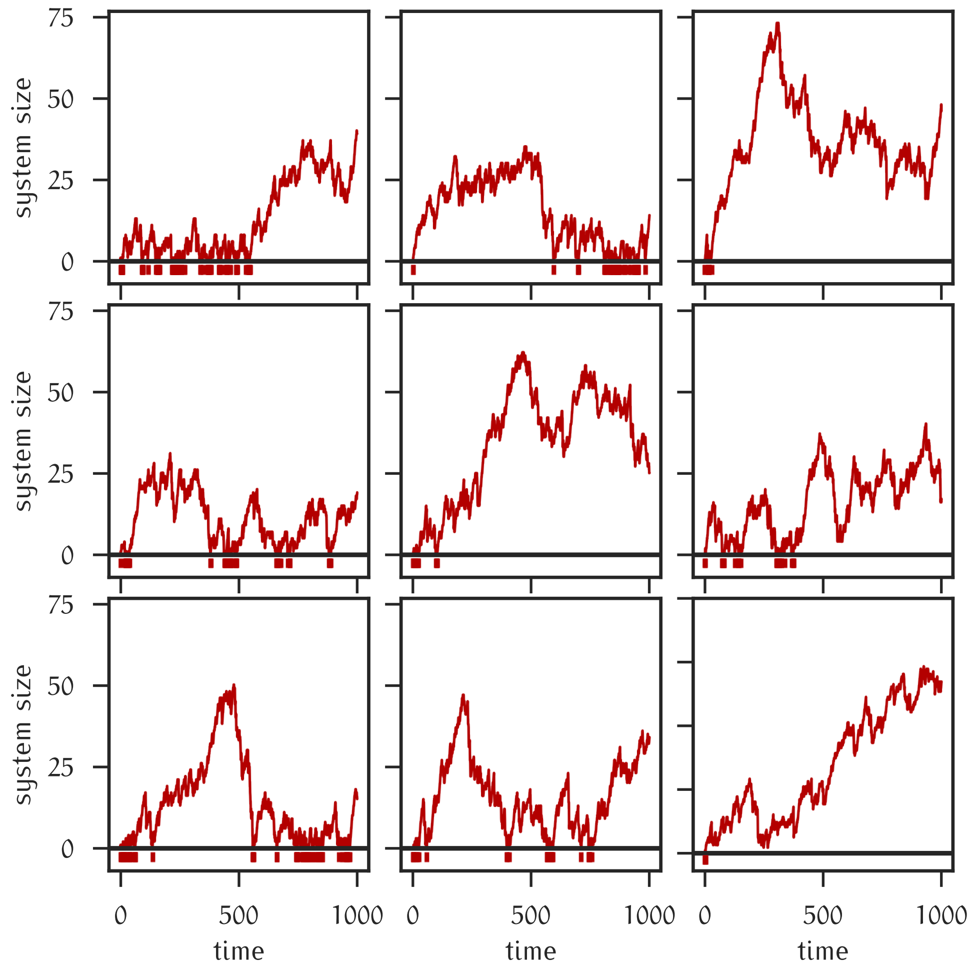

Figure 9.8: Typical trajectories of the critical M/M/1 queue system size Markov chain ($\rho = 1$). Vertical bars in the lower panels highlight returns to the origin.

As with the random walk on the half-line, the M/M/1 queue system size process renews at each return to the origin. Typical finite-time trajectories show the same qualitative behavior as for the random walk with frequent returns to the origin in the recurrent regime and approximate linear growth in the transient regime (Figure 9.6), although the picture is not clear (Figure 9.7). As for the random walk, it also holds for the M/M/1 queue system size process that inspecting a number of finite-time trajectories at the critical point does not allow for classification of recurrence or transience (Figure 9.8).

9.3.3 Analytical return time distribution

The probability that the n-th job arriving to the M/M/1 queue with parameter ρ is the first one to arrive to an empty system after the initial 0-th job is [179, 180]

$$f_n(\rho) = C_{n-1}\frac{\rho^{n-1}}{(1+\rho)^{2n-1}}. \tag{9.20}$$

To see this, first observe that each additional arrival after the initial arrival is associated with a factor $\frac{\rho}{1+\rho}$, while each departure is associated with a factor $\frac{1}{1+\rho}$. For the n-th job to see an empty system ($L_n = 0$), there have to be $n-1$ additional arrivals and n departures. As for the random walk, the number of Dyck paths to have $n-1$ arrivals (incrementing the system size) and $n-1$ departures (decrementing the system size) without touching 0 and a final departure to $L_n = 0$ is the Catalan number C_{n-1}. [180]

Identifying $p = \frac{\rho}{1+\rho}$ and $1-p = \frac{1}{1+\rho}$ establishes the equivalence to the random walk. As we defined the random walk to remain at the origin with probability $P(0 \to 0) = 1-p$, the return time probabilities of the random walk and the M/M/1 queue differ by a factor of $1-p$ and $\frac{1}{1+\rho}$, respectively. This is easily taken care of by demanding that $P(0 \to 1) = 1$ for the random walk.

The return time distribution for the M/M/1 queue asymptotically approaches

$$f_n(\rho) \sim n^{-3/2}\left(4\frac{\rho}{(1+\rho)^2}\right)^n \quad (n \to \infty). \tag{9.21}$$

For the threshold $\rho = 1$, the tail is a power law as $f_n(1) \sim n^{-3/2}$, hinting at a critical transition.

The return probability is

$$L(\rho) = \sum_{n=1}^{\infty}\sum_{n=1}^{\infty} f_n(\rho) = \frac{1}{1+\rho}C(x), \tag{9.22}$$

where $C(x)$ is the generating function of the Catalan numbers with $x = \frac{\rho}{(1+\rho)^2}$. We have

$$C(x) = (1+\rho)\frac{(1+\rho)-|1-\rho|}{2\rho} = \begin{cases} 1+\rho & \rho \leqslant 1 \\ \frac{1+\rho}{\rho} & \rho \geqslant 1, \end{cases} \tag{9.23}$$

and hence

$$L(\rho) = \begin{cases} 1 & \rho \leqslant 1, \\ \frac{1}{\rho} & \rho \geqslant 1. \end{cases} \tag{9.24}$$

The average return time given that the queue returns is

$$\bar{\tau} = \frac{1}{L(p)} \sum_{n=1}^{\infty} n f_n(\rho) = \frac{1}{L(p)(1+\rho)} \frac{d}{dx}(xC(x)) = \frac{1}{L(p)\,|1-\rho|} \tag{9.25}$$

such that

$$\bar{\tau} = \begin{cases} \frac{1}{1-\rho} & \rho < 1 \\ \frac{\rho}{\rho-1} & \rho > 1. \end{cases} \tag{9.26}$$

The average return time diverges at the critical point $\rho_c = 1$ as $(\rho - \rho_c)^{-\gamma}$ with critical exponent $\gamma = 1$.

9.4 AVALANCHES IN EXTREMAL MODELS

Avalanche dynamics in driven systems have been attributed to formation of complex spatiotemporal patterns over all length and time scales. [169, 181, 182] Avalanches also are a dynamical mechanism for self-organized systems to tune themselves to criticality. [183, 184] Instead developing and growing in a linear fashin, avalanche dynamics comprise sudden bursts of activity.

In extremal models, there is a global parameter f_0 that controls the size of the avalanches and defines a hierarchy of sub-avalanches. [169, 182, 183] In such models, every site is characterized by a random number f_i. Let the minimal value at some time n be $f_{\min}(n)$. An f_0 avalanche starts at time n when $f_{\min}(n)$ drops below f_0. The avalanche terminates after s time steps when $f_{\min}$ rises above f_0 again. [169, 182] Given a smaller threshold $f_0' < f_0$, every f_0' avalanche is entirely contained in an f_0 avalanche. Vice versa, every f_0 avalanche may contain several but non-overlapping f_0' avalanches. In these models, avalanche sizes are independent. [169, 182]

The critical state at f_c features avalanches of all sizes, with the avalanche size distribution following a power law as

$$P(s, f_c) \sim s^{-\tau}. \tag{9.27}$$

For the critical region the usual scaling ansatz applies, [169, 182]

$$P(s, f_0) = s^{-\tau} g(s(f_c - f_0)^{1/\sigma}), \tag{9.28}$$

with the scaling function $g(x) \to 0$ for $x \gg 1$ and $g(x) = \text{const.}$ for $x \to 0$. Furthermore, the average avalanche size diverges at the critical point as

$$\bar{S} \sim (f_c - f_0)^{-\gamma}, \tag{9.29}$$

and we have the relationship $\gamma = \frac{2-\tau}{\sigma}$ as in percolation theory. Similarly, the cutoff diverges as $s_\xi \sim (f_c - f_0)^{-1/\sigma}$. In one dimension, this is also the characteristic length scale ξ, which diverges as $\xi \sim (f_c - f_0)^{-\nu}$. Hence, we have $\nu = 1/\sigma$, and $\sigma = 2 - \tau$, as $d = 1, D = 1$ in one dimension. [169, 182]

Beyond the critical point, for $f_0 > f_c$, there is a finite probability that there is an infinite avalanche. This probability scales as $(f_c - f_0)^{\beta}$ for $f_0 > f_c$, with $\beta = \frac{\tau-1}{\sigma} = \frac{\tau-1}{2-\tau}$. [169]

When the control parameter f_0 is raised, avalanches merge. These merging dynamics is another view on the lead-up to the critical transition, similar to recent developments in percolation theory. [98, 169]

10 TEMPORAL PERCOLATION

10.1 MAPPING RETURN PERIODS TO CLUSTERS

In this Chapter, I introduce the mapping of recurrence dynamics of a stochastic process to a one-dimensional percolation setting. This mapping establishes an equivalence of the temporal transition from recurrence in a dynamical system to transience to the spatial percolation transition on a linear graph.

Specifically, given a time-discrete stochastic process X_n on some state space S, and given a return set $B \subset S$, each realization ω of the process induces a graph $G(\omega)$ with nodes $V = \mathbb{N} = \{0, 1, 2, \dots\}$ and edge set

$$E(\omega) = \{ (n, n+1) \mid X_{n+1}(\omega) \notin B \}. \tag{10.1}$$

In other words, the sites represent time and a bond connects two successive sites n and $n+1$ if and only if the system does not return to B in that time step $n \to n+1$ (Figure 10.1). Equivalently, each return of the process X_n induces a cluster of the same size as the time until the next return. Yet another mapping is that of the dynamics of the process X_n to a random binary return sequence R_n with $R_n = 1$ if $X_n \in B$ and $R_n = 0$ otherwise.

How does this percolation setting represent the recurrence–transience dichotomy of the original process? In the recurrent regime, the process almost surely returns infinitely often: sample paths of the process will almost surely map to infinitely many finite clusters. This corresponds to the subcritical regime of a percolation setting. At the critical point, while the process still returns infinitely often, the expected return time diverges, and so does the average size of the clusters. Finally, in the transient regime, the process has a finite probability less than one to return. This means that eventually, the process will not return any more almost surely. An infinite cluster emerges in the percolation setting.

Hence, mapping the stochastic process dynamics onto a one-dimensional percolation setting enables the description of the recurrence–transience transition in terms of a phase transition, and the usual statistical physics toolkit applies. Intriguingly, as the process still returns for a finite number of times in the transient regime, there are finite clusters remaining and coexisting with the infinite cluster even beyond the percolation transition. This differs from the conventional Bernoulli percolation setting in one dimension: this setting features a discontinuous transition without finite clusters, as the infinite cluster extends to infinity on both sides of the linear chain. As our temporal percolation setting is a one-sided chain, there is somehow paradoxically more room at the bottom for finite clusters to persist.

Besides the finite-cluster peculiarity, the infinite cluster of temporal percolation is similar to the infinite cluster of spatial Bernoulli percolation in

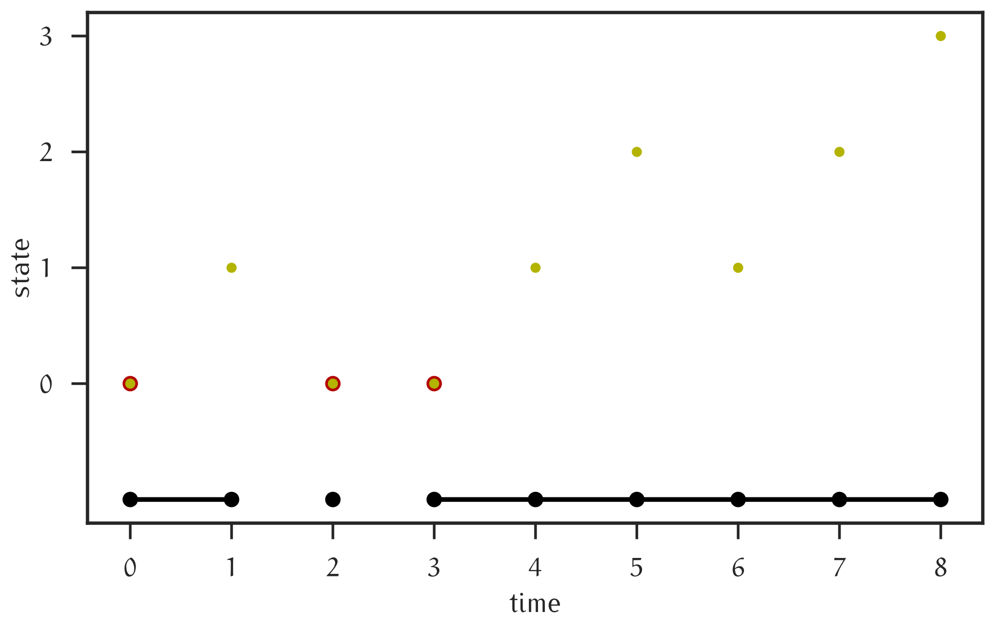

Figure 10.1: Mapping recurrence dynamics onto a one-dimensional percolation setting. A sample path of a random walk on the half-line is depicted for the first 8 time steps. Returns to the origin at times $0, 2, 3$ are highlighted. In the percolation setting below, each time step is a site and there is a bond between two sites if the process does not return in that time step. If the process does return, the link is missing and a new cluster starts.

one dimension. The spanning probability, or percolation probability, is the probability of the existence of an infinite cluster:

$$\Pi(\rho) = 1 - Q(x, B) = \begin{cases} 0 & \rho \leqslant \rho_c \\ 1 & \rho > \rho_c, \end{cases} \tag{10.2}$$

where ρ_c is the critical parameter value and $Q(x, B)$ the probability to return to B infinitely often, which is the same for almost all x in irreducible Markov chains. The order parameter is the relative size of the largest cluster, also called the percolation strength $P(\rho)$. In temporal percolation, finite clusters persist. However, there is almost surely a finite number of finite clusters, encompassing an expected finite number of sites, such that the infinite cluster accrues weight 1:

$$P(\rho) = \begin{cases} 0 & \rho \leqslant \rho_c \\ 1 & \rho > \rho_c. \end{cases} \tag{10.3}$$

10.2 RETURN TIMES AS FUNDAMENTAL QUANTITY

The return time distribution f_n is the fundamental quantity of temporal percolation, just as the cluster size distribution n_s is in spatial percolation. [86] Given a discrete-time stochastic process X_n on some state space S

with parameter ρ that controls the drive to transience, and given a return set $B \subset S$ and the initial state $x \in S$, the return time distribution is [127]

$$f_n(\rho) \equiv f_n(x, B) = P\{\tau_B(\rho) = n \mid X_0 = x\}. \tag{10.4}$$

The return probability is

$$L(\rho) \equiv L(x, B) \equiv \sum_{n=1}^{\infty} f_n(\rho). \tag{10.5}$$

In the recurrent regime, $L(\rho) = 1$, whereas in the transient regime $L(\rho) < 1$. The probability not to return is $1 - L(\rho)$, which takes up finite values between 0 and 1, while the probability that the process does eventually not return is $1 - Q(x, B) \equiv 1 - Q(\rho) \in \{0, 1\}$, as pointed out before. We do have the equivalence

$$L(\rho) < 1 \Leftrightarrow Q(\rho) = 0, \quad L(\rho) = 1 \Leftrightarrow Q(\rho) = 1, \tag{10.6}$$

where $Q(\rho)$ is the probability that the process returns infinitely often. This equivalence also holds at $\rho = \rho_c$ as $L(\rho_c) = 1, Q(\rho_c) = 1$. The average return time is the average of finite return times:

$$\bar{\tau}(\rho) = \mathbb{E}[\tau(\rho) \mid \tau(\rho) < \infty] = \frac{1}{L(\rho)} \sum_{n=1}^{\infty} n f_n(\rho). \tag{10.7}$$

As we have seen in extremal models and in the random walk and M/M/1 queue, the average return time diverges at the critical point, $\bar{\tau} \to \infty$ as $\rho \to \rho_c$.

Let us take a step back and see how this compares to percolation theory. In percolation theory, the cluster number n_s is an intensive quantity: it is the relative number of clusters of size s as a fraction of system size. In the temporal percolation setting, this would be the probability that any given site starts a finite cluster of size s. In the recurrent regime, there is a simple relationship between the distribution f_s and the cluster number n_s of return times and cluster sizes of size or duration s, respectively:

$$n_s(\rho) = \frac{f_s(\rho)}{\sum_s s f_s(\rho)} = \frac{f_s(\rho)}{L(\rho)\bar{\tau}(\rho)} \quad (\rho < \rho_c). \tag{10.8}$$

Note that with all quantities being finite except for $\bar{\tau}$ diverging at ρ_c, the cluster number n_s vanishes at the critical point. In the transient regime, there is an infinite cluster of weight 1, and hence we have $n_s = 0$.

10.3 SCALING RELATIONS

We have introduced the return time distribution as the fundamental quantity of temporal percolation. Just as the scaling theory of percolation clusters relates the critical exponents of the percolation transition to the cluster size distribution, and the theory of extermal models relates the critical exponents of self-organized critical spatiotemporal activity to the avalanche size distribution, we here relate the critical exponents of the temporal percolation transition to the return time distribution. [94, 169]

In line with these theories of critical transitions, and in line with the theory of stability of Markov chains, we postulate a critical transition in Markov chains signified by the return time distribution asymptotically following a power law as

$$f_n(\rho_c) \sim n^{-\tau}, \quad (n \to \infty) \tag{10.9}$$

with Fisher exponent τ. The scaling assumption is that the ratio $f_n(\rho)/f_n(\rho_c)$ is a function of the ratio $n/\xi(\rho)$ only, with the characteristic return time $\xi(\rho)$. As we are in one dimension, the characteristic return time $\xi(\rho)$ is also the temporal coherence scale of the infinite system, which diverges as

$$\xi(\rho) \sim |\rho - \rho_c|^{-\nu}, \quad (\rho \to \rho_c). \tag{10.10}$$

Hence, the scaling ansatz implies the return time distribution

$$f_n(\rho) = n^{-\tau} g\left(\frac{n}{\xi(\rho)}\right) \tag{10.11}$$

with the scaling function $g(x) \to 0$ for $x \to \infty$ and $g(x) = \text{const.}$ for $x \to 0$.

The probability of no return is 0 for $\rho \leqslant \rho_c$ and finite for $\rho > \rho_c$. It scales as

$$1 - L(\rho) \sim \sum_n n^{-\tau}\left(1 - g\left(\frac{n}{\xi(\rho)}\right)\right) \sim \sum_{n=\xi(\rho)}^{\infty} n^{-\tau} \sim \xi(\rho)^{1-\tau} \tag{10.12}$$

in the critical region ($\rho > \rho_c, \rho \to \rho_c$). Hence, the probability of no return scales as

$$1 - L(\rho) \sim (\rho - \rho_c)^{\beta}, \quad (\rho > \rho_c, \rho \to \rho_c) \tag{10.13}$$

with critical exponent $\beta = \nu(\tau - 1)$.

The average return time $\bar{\tau}(\rho)$ scales as

$$\bar{\tau}(\rho) = \sum_n n f_n(\rho) \sim \sum_n n^{-\tau+1} g\left(\frac{n}{\xi(\rho)}\right) \sim \sum_{n=1}^{\xi(\rho)} n^{-\tau+1} \sim \xi^{-\tau+2}, \tag{10.14}$$

and hence, $\bar{\tau}(\rho) \sim |\rho - \rho_c|^{-\gamma}$ with critical exponent $\gamma = \nu(2 - \tau)$.

10.4 MERGING RETURN PERIODS

We now consider the process of merging return periods in the temporal percolation setting when the control parameter ρ is tuned towards the transition at $\rho = \rho_c$. Without loss of generality, let us consider the Markov chain X_n on $\mathbb{N}$ with returns to the origin in the form of a stochastic recursive sequence

$$X_n = h(X_{n-1}, \zeta_n, \rho) \tag{10.15}$$

with independent and uniformly distributed random variables $\zeta_n \in [0, 1]$ and

$$\frac{\partial}{\partial \rho} h(x, \zeta, \rho) \geqslant 0, \quad \frac{\partial}{\partial \zeta} h(x, \zeta, \rho) \geqslant 0. \tag{10.16}$$

In the following, we will analyze sample paths $X_n(\omega)$ of the process determined by $X_0(\omega) = 0$ and $\zeta_n(\omega)$. For some parameter value $\rho < \rho_c$, consider the binary return sequence $R_n(\rho)$, or equivalently, the sequence of ρ-return times $\tau_i(\rho)$ defined as $\tau_i = \min\{n : n > \tau_{i-1}, R_n = 1\}$ with the initial ρ-return time $\tau_1 = \tau$ being the (first) return time.

We observe that almost surely, the ρ-return times $\tau_i(\rho)$ of any given realization ω of the process form a hierarchy as

$$\{\tau_i(\rho)\}_{i=1}^{\infty} \supseteq \{\tau_i(\rho')\}_{i=1}^{\infty} \tag{10.17}$$

for $\rho < \rho'$; every ρ'-return time is also a ρ-return time. This is analogous to avalanches in extremal models and clusters in percolation settings. We see this in the temporal percolation setting by noting that due to $\frac{\partial}{\partial\rho} h(x, \zeta, \rho) \geqslant 0$, the trajectory $X_n(\rho', \omega)$ dominates $X_n(\rho, \omega)$ as

$$X_n(\rho', \omega) \geqslant X_n(\rho, \omega), \quad (\rho' > \rho). \tag{10.18}$$

Furthermore, if and when two or more return periods merge, solely depends on the first of these return periods. The reason is that the return times $\tau_i(\rho)$ only depend on previous states (and being renewal points, in fact, only on the preceding renewal period starting at $\tau_{i-1}(\rho)$).

11 METHODS

11.1 FINITE-TIME SCALING ANALYSIS

11.1.1 Rationale

The aim of finite-time scaling analysis is to recover the critical point ρ_c and critical exponents β for the no-return probability $1 - L(\rho)$ and γ for the average return time $\bar{\tau}$ from Monte Carlo simulations of finite-time trajectories. Furthermore, finite-time scaling analysis produces estimates of the exponents ν of the temporal coherence scale and τ of the return time distribution. This is achieved by collapsing the recorded data for the relevant quantities onto a single master curve, respectively. To find the right choice of values for the critical parameter and exponents is subject to an optimization algorithm. Such an algorithm tunes the data collapse according to a goal function quantifying the goodness-of-collapse.

11.1.2 The finite-time scaling ansatz

We have established that the exponents of the return time distribution determine the critical exponents at the temporal percolation transition, and vice versa. As a phase transition only occurs in an infinite system, numerical simulation in finite time cannot probe the transition directly. Here, we adapt the conventional remedy of finite-size scaling analysis to the temporal percolation setting. This finite-*time* scaling analysis numerically recovers the critical point ρ_c and the critical exponents.

Let T be the temporal extent of a dynamical system, i. e. the number of time steps computed in a simulation. Let $A_T(\rho)$ be a quantity that diverges as $|\rho - \rho_c|^{-\zeta}$ in the critical region of the infinite system $(T \to \infty, \rho \to \rho_c)$. The finite-size scaling ansatz translates to [116–118]

$$A_T(\rho) = T^{\zeta/\nu}\tilde{f}\left(T^{1/\nu}(\rho - \rho_c)\right), \quad (T \to \infty, \rho \to \rho_c), \tag{11.1}$$

with the dimensionless scaling function $\tilde{f}$ and the critical exponent ν of the temporal coherence scale $\xi(\rho)$ in the infinite system. The scaling function controls the finite-time effects.

A Monte Carlo study yields data $a_i^{T,\rho}$ at system size T and parameter ρ for each run i. Let $a_{T,\rho}$ denote the average over all runs. Plotting $T^{-\zeta/\nu}a_{T,\rho}$ against $T^{1/\nu}(\rho - \rho_c)$ should let numerical data collapse onto a single master curve $\tilde{f}(x)$. For this to happen, the critical values ρ_c, ζ, ν need to be correct. These assumptions hold for $T \to \infty$, with systematic errors at finite sizes. [116, 117].

For quantities that jump at the critical point ρ_c, such as the size $P(\rho)$ of the largest cluster in temporal percolation, we have the finite-time scaling

$$P_T(\rho) = \bar{P}\left(T^{1/\nu}(\rho - \rho_c)\right) \tag{11.2}$$

with scaling function $\bar{P}(x)$, as the critical exponent ζ of $P(\rho)$ is zero. Hence, independent of system size, we have $P_T(\rho_c) = \bar{P}(0)$. The common intersection point of the measured data curves yields an estimate of the threshold ρ_c. This estimate is unbiased with regards to the critical exponents, and "should be free" from systematic errors due to finite system size. [116]

11.1.3 Quality of finite-time data collapse

The finite-time scaling ansatz (11.1) quantifies how a statistic $A_T(\rho)$ observed in finite-time trajectories scales with time T and parameter ρ according to a scaling function $\tilde{f}$, the critical parameter ρ_c, the critical exponent ζ of the quantity itself, and the critical exponent ν of the temporal coherence scale. [116, 117]

Finite-time scaling analysis takes numerical data a_{T_i,ρ_j} at system sizes T_i and parameter values ρ_j. Plotting $T_i^{-\zeta/\nu} a_{T_i,\rho_j}$ against $T_i^{1/\nu}(\rho - \rho_c)$ with the right choice of ρ_c, ν, ζ should let the data collapse onto a single curve. The single curve is the scaling function $\tilde{f}$ from the finite-time scaling ansatz. In the following, we present a measure by Houdayer and Hartmann [185] for the quality of the data collapse. Melchert [186] refers to some alternative measures, for example those in References [187, 188], and to some applications of these measures in the literature.

Houdayer and Hartmann [185] refine a method proposed by Kawashima and Ito [189]. They define the quality as the reduced χ^2 statistic

$$S = \frac{1}{\mathcal{N}} \sum_{i,j} \frac{(y_{ij} - Y_{ij})^2}{dy_{ij}^2 + dY_{ij}^2}, \tag{11.3}$$

where the values y_{ij}, dy_{ij} are the scaled observations and its standard errors at x_{ij}, and the values Y_{ij}, dY_{ij} are the estimated value of the master curve and its standard error at x_{ij}. The sum in the quality function S only involves terms for which the estimated value Y_{ij} of the master curve at x_{ij} is defined. The number of such terms is $\mathcal{N}$. The quality S is the mean square of the weighted deviations from the master curve. As we expect the individual deviations $y_{ij} - Y_{ij}$ to be of the order of the individual error $\sqrt{dy_{ij}^2 + dY_{ij}^2}$ for an optimal fit, the quality S should attain its minimum $S_{\min}$ at around 1 and be much larger otherwise. [190]

Let i enumerate the system sizes T_i, $i = 1, \ldots, k$, and let j enumerate the parameters ρ_j, $j = 1, \ldots, n$ with $\rho_1 < \cdots < \rho_n$. The scaled data are

$$y_{ij} = T_i^{-\zeta/\nu} a_{T_i,\rho_j} \tag{11.4}$$

$$dy_{ij} = T_i^{-\zeta/\nu} da_{T_i,\rho_j} \tag{11.5}$$

$$x_{ij} = T_i^{1/\nu}(\rho_j - \rho_c). \tag{11.6}$$

The master curve itself depends on the scaled data. For a given i or T_i, we estimate the master curve at x_{ij} by the two respective data from all other system sizes which respectively enclose x_{ij}: for each $i \neq j$, let j' be such that $x_{i'j'} \leqslant x_{ij} \leqslant x_{i'(j'+1)}$, and select the points $(x_{i'j'}, y_{i'j'}, dy_{i'j'})$, $(x_{i'(j'+1)}, y_{i'(j'+1)}, dy_{i'(j'+1)})$. Do not select points for some i' if there is no such j'. If there is no such j' for all i', the master curve remains undefined at x_{ij}.

Given the selected point (x_l, y_l, dy_l), the local approximation of the master curve is the linear fit

$$y = mx + b \tag{11.7}$$

with weighted least squares. [191] The weights w_l are the reciprocal variances, $w_l = 1/dy_{ij}^2$. The estimates and (co)variances of the slope m and intercept b are

$$\hat{b} = \frac{1}{\Delta}(K_{xx}K_y - K_x K_{xy})$$
$$\hat{m} = \frac{1}{\Delta}(KK_{xy} - K_x K_y)$$
$$\hat{\sigma}_b^2 = \frac{K_{xx}}{\Delta}, \hat{\sigma}_m^2 = \frac{K}{\Delta}, \hat{\sigma}_{bm} = -\frac{K_x}{\Delta}$$

with $K_{nm} = \sum_l w_l x_l^n y_l^m$, $K = K_{00}$, $K_x = K_{10}$, $K_y = K_{01}$, $K_{xx} = K_{20}$, $K_{xy} = K_{11}$, $\Delta = KK_{xx} - K_x^2$. [185] Hence, the estimated value of the master curve at x_{ij} is

$$Y_{ij} = \hat{m}x_{ij} + \hat{b} \tag{11.8}$$

with error propagation

$$dY_{ij}^2 = \hat{\sigma}^2 x_{ij}^2 + 2\hat{\sigma}_{bm} x_{ij} + \hat{\sigma}_b^2. \tag{11.9}$$

11.1.4 A refinement of the quality function

In this Thesis, I further refine the quality function (11.3) to let the data for each system size have equal weight. The original sum involves only terms for which the master curve is defined. As the number of missing terms in general differs from system size to system size, the sum implicitly weights system sizes differently. This is unintended behavior, especially when it comes to scalings with less dense coverage of the critical region at large system sizes.

To alleviate this, I modify the sum (11.3) as follows:

$$S' = \frac{1}{k}\sum_i \frac{1}{\mathcal{N}_i}\sum_j \frac{(y_{ij} - Y_{ij})^2}{dy_{ij}^2 + dY_{ij}^2}, \tag{11.10}$$

where the number of system sizes is k (as before), and $\mathcal{N}_i$ is the number of terms for the i-th system size. By separately averaging over all available terms for each system size, and only then averaging over all system sizes, the contributions of each system size have equal weight in the final sum.

11.1.5 Parameter estimation

Following Melchert [186], we employ the Nelder–Mead algorithm to minimize the quality function and estimate the critical parameter value ρ_c and exponents ν and ζ from Monte Carlo data.

The Nelder–Mead algorithm attempts to minimize a goal function $f : \mathbb{R}^n \to \mathbb{R}$ of an unconstrained optimization problem. [192] As it only evaluates function values, but no derivatives, the Nelder–Mead algorithm classifies as a direct search method. [193] Although the method generally lacks rigorous convergence properties, [194, 195] in practice the first few iterations often yield satisfactory results. [196] Typically, each iteration evaluates the goal function only once or twice, which is why the Nelder–Mead algorithm is comparatively fast if goal function evaluation is the computational bottleneck. [196, 197] Nelder and Mead [192] refined a simplex method by Spendley, Hext, and Himsworth [198]. A simplex is the generalization of triangles in $\mathbb{R}^2$ to n dimensions: in $\mathbb{R}^n$, a simplex is the convex hull of $n+1$ vertices $x_0, \ldots, x_n \in \mathbb{R}^n$. Starting with the initial simplex, the algorithm attempts to decrease the function values $f_i = f(x_i)$ at the vertices by a sequence of elementary transformations of the simplex along the local landscape. The algorithm succeeds when the simplex is sufficiently small (domain convergence), and/or when the function values f_i are sufficiently close (function-value convergence). The algorithm fails when it did not succeed after a given number of iterations or function evaluations. See Singer and Nelder [196] and references therein for a complete description of the algorithm and the simplex transformations.

In order to estimate the uncertainties of the critical paramter value ρ_c and critical exponents ν and ζ, we employ the method suggested by Spendley, Hext, and Himsworth [198] and Nelder and Mead [192]. Fitting a quadratic surface to the vertices and the midpoints of the edges of the final simplex yiealds an estimate for the variance–covariance matrix. The errors are the square roots of the diagonal terms. [190]

The Nelder–Mead algorithm needs an initial guess for ρ_c, ν, ζ. This part of the analysis requires human oversight and intervention. Inspecting the data, we will have an idea of the approximate location of the critical point. Given that the critical exponents of the percolation probability and the percolation strength should be zero, we determine ρ_c and ν by performing the finite-time scaling analyses on these data first (assuming that $\zeta = 0$). If the Nelder–Mead simplex gets stuck in a local minimum, it is recommended to simply restart the search, with slightly off, or considerably revised, initial values. [196]

11.2 THE FSSA PYTHON PACKAGE

11.2.1 About the package

pyfssa is a scientific Python package for algorithmic finite-size scaling analysis at critical transitions. [199] It partially reimplements and enhances the autoscale.py script by Melchert [186]. Is is open source and free software under the permissive ISC License. Unit tests cover about 95% of its codebase. The package documentation is available online. [200] As of 2015, Scientific Python implements the Nelder–Mead method for the `scipy.optimize.minimize` routine. [62–64] Note that this implementation returned the vertex with the lowest function value, but not the whole final simplex. I wrote and submitted a patch to the scipy package that has been incorporated by the release of version 0.17. [201] pyfssa has been cited in the literature. [202, 203]

The pyfssa package depends on the python-future, NumPy and SciPy packages. [61–64, 204] python-future ensures compatibility across Python and Legacy Python. NumPy provides the numerical data array structure and methods, and SciPy provides statistical and optimization methods. Installation of pyfssa is straightforward with the Python packaging mechanism and the pip setup tool. The user is free to download pyfssa either from the Python package index or from the sources stored at the public repositories at GitHub or Zenodo.

11.2.2 Finite-time scaling implementation

The pyfssa Python package implements the algorithmic finite-size scaling analysis as inspired and implemented by Melchert [186]. The Nelder–Mead algorithm numerically searches for those critical values that minimize the quality function by Houdayer and Hartmann [185].

The `fssa.scaledata` function (Listing C.12) scales finite-size data in order for the data to prospectively collapse onto a single universal scaling function, also known as "the" master curve.

The `fssa.quality` function (Listing C.12) assesses the quality of the data collapse onto a single curve. It returns the reduced χ^2 statistic for a data fit except that the master curve is fitted from the data itself. This is the implementation of the modified quality function S' of Equation 11.10.

Finally, the `fssa.autoscale` function (Listing C.12) frames the data collapse as an optimization problem and searches for the critical values that minimize the quality function.

11.2.3 Verification

In order to verify the implementation of the autoscaling algorithm in pyfssa, I tested it against a well-studied system for which all scaling exponents are available. [200] The system of choice is the paradigmatic bond percolation setting on a regular grid, in one dimension (linear chain) and in two dimensions (square lattice). I utilize the percolate Python package to generate the percolation finite-size data. [205] pypercolate is a scientific Python package that implements the Newman–Ziff algorithm for Monte Carlo simulation of percolation on graphs. [206] The finite system sizes of the linear chain (number of nodes) are $2^6, 2^8, \ldots 2^{20}$, and the linear extensions of the square lattices are $2^3, 2^4, \ldots, 2^{10}$. The total number of runs is 10^4, respectively. The occupation probabilites are $\left\{ 1 - 10^{-(1+i/10)} : i = 0, \ldots, 100 \right\}$ for the linear chain and $\left\{ \frac{1}{2} + \frac{i}{500} : i = -50, \ldots, +50 \right\}$ for the square lattice. I obtain finite-

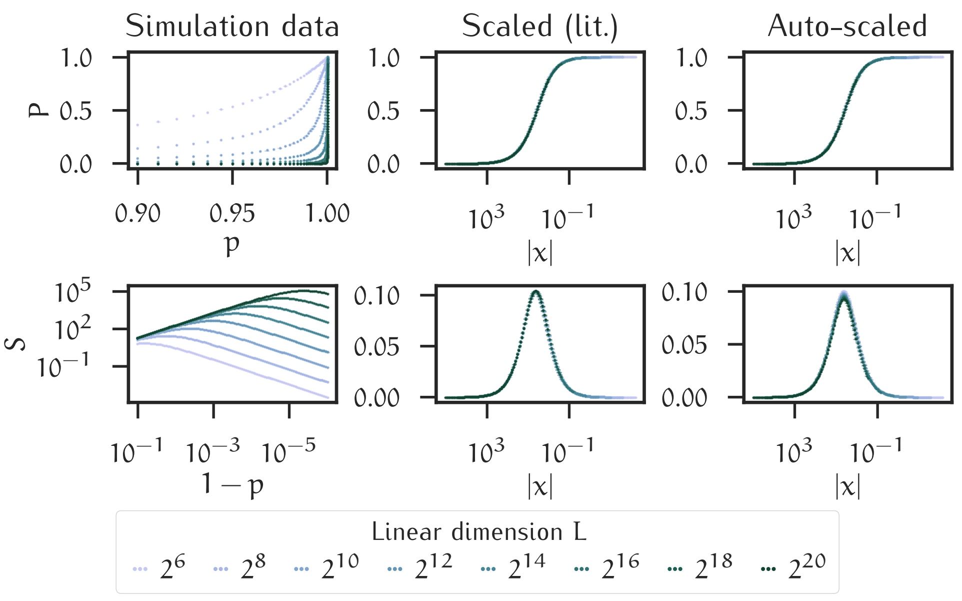

Figure 11.1: pyfssa verification study for Bernoulli bond percolation on the linear chain: Finite-size data from 10^4 simulations runs at different system sizes L for the percolation strength P (size of largest cluster, upper row) and the average cluster size S (lower row) on the linear chain. The numerical data is scaled with the exponents from the literature (center column) [96], and auto-scaled by pyfssa (right column), with $x = L^{1/\nu}(p - p_c)$.

Table 11.3: Critical point and scaling exponents for the bond percolation problem on the linear chain, as given by the literature [96] and as determined by auto-scaling finite-size simulation data with pyfssa.

1D	literature	auto-scaled Π	auto-scaled P	auto-scaled S
p_c	1	–	1.00000003(12)	1.0000002(14)
ν	1	–	1.003(7)	1.003437(5)
β	0	–	$2.2(1.8) \cdot 10^{-4}$	–
γ	1	–	–	1.012(8)

size data for the percolation probability Π, the percolation strength (largest cluster size) P, and the average cluster size S. Using the values for the critical exponents from the literature, I scale the data by the `fssa.scaledata` routine and visually assess the data collapse onto a single master curve. Furthermore, I algorithmically determine the critical exponents by the `fssa.autoscale` routine. For better fits, I omit data from small system sizes and only feed data from systems of more than $2^{14} \approx 10^4$ sites. I initialize the Nelder–Mead search with the literature values.

I extract the exponents and their errors from the return value of the `fssa.autoscale` routine (Table 11.3 and Table 11.4).

For the linear chain, pyfssa recovers the critical point both for the percolation strength and the average cluster size, where the analytical value $p_c = 1.0$ lies within the negligibly small error intervals of sizes 10^{-6} and 10^{-7}. The percolation strength scaling further recovers the ν scaling exponent within the small error interval of order of magnitude 10^{-3}. While the interval for ν derived from autoscaling the S data does not contain the literature value 1.0,

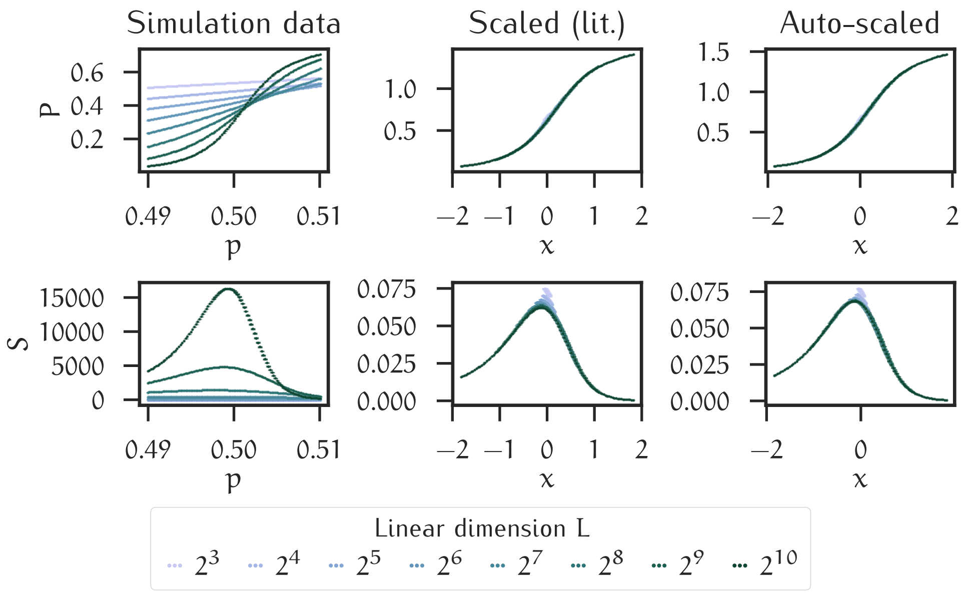

Figure 11.2: pyfssa verification study for Bernoulli bond percolation on the linear chain: Finite-size data from 10^4 simulations runs at different system sizes L^2 for the percolation strength P (size of largest cluster, upper row) and the average cluster size S (lower row) on the $L \times L$ square grid. The numerical data is scaled with the exponents from the literature (center column) [79], and auto-scaled by pyfssa (right column), with $x = L^{1/\nu}(p - p_c)$.

Table 11.4: Critical point and scaling exponents for the bond percolation problem on the square lattice, as given by the literature [79] and as determined by auto-scaling finite-size simulation data with pyfssa.

2D	literature	auto-scaled Π	auto-scaled P	auto-scaled S
p_c	$\frac{1}{2}$	0.5000(11)	0.49999(15)	0.50007(9)
ν	1.333	1.328(36)	1.33(2)	1.33(3)
β	−0.133	–	−0.140(12)	–
γ	2.4	–	–	2.38(5)

it deviates only by a small fraction of 10^{-3}. The intervals for both the order parameter exponent β and the susceptibility exponent γ do not contain the true values. However, the algorithmically determined values for β and γ deviate from the true values by only a small fraction of the order of 10^{-4} and 10^{-2}, respectively.

For the square lattice, the intervals for all exponents and all magnitudes contain the literature value and are rather small: 10^{-2} for the critical exponents ν, β and γ; 10^{-4} for the critical point p_c.

In the light of this agreement, we need to keep in mind that I initialized the auto-scaling algorithm with the literature values. This certainly helps making the Nelder–Mead optimization results and error estimates well-defined. All the usual advice with the Nelder–Mead search method applies here. For example, the search is prone to getting stuck in a local minimum. Note that the goal of this verification study is not to demonstrate the feasibility of the Nelder–Mead method. The goal is to show that the algorithm reproduces the known exponents in suitable initial conditions.

To conclude verification of pyfssa, both visual inspection (Figure 11.1 and Figure 11.2) as well as algorithmic scaling analysis of the bond percolation data reproduce well-established results from the literature. Even though the finite-size scaling analyst needs to take the usual care of manually (visually) approaching data collapse in order to initialize the Nelder–Mead search, I conclude the foregoing computational analysis to support that pyfssa is correctly implemented.

11.3 ALGORITHMIC TEMPORAL PERCOLATION ANALYSIS

11.3.1 Introduction

In the following, I present a method to algorithmically analyse the temporal percolation transition. At the heart of the method is the usual Monte Carlo finite-size scaling analysis, so the major piece is to adapt finite-time trajectories of stochastic processes to that methodology conceptually and computationally. Here, I first define the input to the algorithmic temporal percolation analysis, before I derive and review the relevant statistics for finite-time analysis.

11.3.2 Input

The input to algorithmic temporal percolation analysis consists of

- a list of increasing finite simulation times T_i,
- a list of increasing control parameter values ρ_j covering the critical region, and
- a list of seeds or initial conditions ω_k for the simulator to generate sample paths of the process X_n,
- a simulator ϕ of a time-discrete dynamical system X_n ($n \in \{0, 1, 2, \dots\}$),

with $T_{\max} = \max_i T_i$. The simulator ϕ takes as input the finite simulation time T, the control parameter ρ, the seed ω. Basically, the simulator ϕ evolves the system according to the stochastic recursive sequence $X_{n+1} = f(X_n, \zeta_n, \rho)$, where the ζ_n are determined by the random seed ω, such that $\zeta_n = \zeta_n(\omega)$. For parameter ρ_j and realization ω_k, the simulator shall (deterministically) generate a finite binary return sequence R_n with $R_n = 1$ signifying a return ($X_n(\omega)$ is in the return set) and $R_n = 0$ not. Note that we use the same set of realizations ω_k for all parameters ρ_j and finite times T_i. Conceptually, this is consistent with the view of stochastic processes being random elements on the sequence space of the state space (rather than the equivalent view of stochastic processes as collections of time-ordered random variables on the state space). Numerically, this is the variance reduction method of common random numbers, [207] which is, for example, also employed in the Newman–Ziff algorithm. [206] In particular, for each parameter ρ_j and realization ω_k, the return sequence R_n for the maximal simulation time $T_{\max}$ already contains the return sequences for all other times T_i.

For each triple $ijk \equiv (T_i, \rho_j, \omega_k)$ we recursively map the finite return sequence R_n (generated for $T_{\max}$ and truncated at the individual T_i) to a finite sequence of return epochs t_l as $t_0 = 0$ and $t_{l+1} = \min\{\, n > t_l : R_n = 1 \,\}$. Let L_{ijk} denote the number of such returns for $n > 0$ within the finite simulation time T_i. The return periods $\tau_{ijk,l}$ are

$$\tau_{ijk,l+1} = \begin{cases} t_{jk,l+1} - t_{jk,l} & \text{for } l < L_{ijk}, \\ T_i - t_{jk,l} & \text{for } l = L_{ijk}. \end{cases} \tag{11.11}$$

Note that in this mapping, the length of the first return period is agnostic to whether there is a return at $n = 0$ or not. There are $L_{ijk} + 1$ return periods, including the additional return period after the last return. Let $\tau_{ijk,L}$ denote the last return period $\tau_{ijk,L} = \tau_{ijk,L_{ijk}+1}$. If the last return coincides with the finite simulation time, that is, $t_{jk,L_{ijk}} = T_i$, then the last return period $\tau_{ijk,L} = 0$. However, if $t_{jk,L_{ijk}} < T_i$, we have a last return period that is truncated at T_i—this last return period $\tau_{ijk,L}$ could already be the onset of an infinite return period in the transient regime of the infinite system ($T \to \infty$).

11.3.3 Finite-time statistics

Given the $L_{ijk} + 1$ return periods $\tau_{ijk,l}$ including the last (truncated) return period $\tau_{ijk,L}$ at finite time T_i, parameter value ρ_j and run ω_k, we first calculate the statistics for a single run k and subsequently, for averaging over all runs.

We say a system at T_i has a "spanning cluster", expressed by the Boolean variable Π_{ijk}, if the last return period exceeds some threshold of the order of the system size:

$$\Pi_{ijk} = \begin{cases} 1 & \text{if } \tau_{ijk,L} > (1 - e^{-1}) T_i \approx 0.632 \cdot T_i, \\ 0 & \text{otherwise.} \end{cases} \tag{11.12}$$

The relative size of the largest return period (including the truncated last return period) is

$$P_{ijk} = \frac{1}{T_i} \max \left\{ \tau_{ijk,l} : l = 1, \ldots, L_{ijk} + 1 \right\}. \tag{11.13}$$

The m-th empirical raw moment of the return periods is

$$M_{ijk,m} = \sum_{l=1}^{L_{ijk}} (\tau_{ijk,l})^m, \tag{11.14}$$

which excludes the last (and possibly infinite) return period.

In the next step, the temporal percolation method averages over all K runs to arrive at data points A_{ij} for finite time T_i and each parameter value ρ_j. Statistical uncertainty of these averages are expressed in the form of (frequentist) confidence intervals, or alternatively, Bayesian credible intervals (see VanderPlas [208] and references therein for a discussion on why frequentist intervals should be avoided). For both intervals, the parameter α quantifies the extent of the uncertainty. For a confidence interval, the value $1 - \alpha$ is the probability for the random confidence interval to contain the fixed true average. For a credible interval, the value $1 - \alpha$ is the probability that the (random) average is contained in the interval fixed by the data. We quantify uncertainty on the 1σ level, such that $\alpha \approx 0.317$.

The finite-time temporal percolation strength P_{ij} is the average size of the largest return period:

$$P_{ij} = \frac{1}{K} \sum_k P_{ijk}, \tag{11.15}$$

with the standard normal confidence interval based on the sample variance $\frac{1}{K-1} \sum_k (P_{ijk} - P_{ij})^2$.

The spanning probability Π_{ij} is a Binomial proportion, i.e. a series of K independent Bernoulli trials with some (unknown) success probability p. As Cameron [209] puts it the normal approximation to confidence intervals "suffers a *systematic* decline in performance (...) towards extreme values of p near 0 and 1, generating binomial [confidence intervals] with effective coverage far below the desired level." (see also References [210, 211]). This is another reason to employ Bayesian inference. [212] For K independent Bernoulli trials with success probability p, the likelihood to have k successes given p is the binomial distribution

$$P(k \mid p) = \binom{K}{k} p^k (1-p)^{K-k} = B(a, b), \tag{11.16}$$

where $B(a, b)$ is the Beta distribution with parameters $a = k + 1$ and $b = N - k + 1$. Assuming a uniform prior $P(p) = 1$, the posterior is [212]

$$P(p \mid k) = P(k \mid p) = B(a, b). \tag{11.17}$$

Finally, the spanning probability is the posterior mean

$$\Pi_{ij} = \frac{\sum_k \Pi_{ijk} + 1}{K + 2} \tag{11.18}$$

with $1-\alpha$ credible interval $(\Pi_{ij}^{\downarrow}, \Pi_{ij}^{\uparrow})$ given by

$$\int_0^{\Pi_{ij}^{\downarrow}} B(a,b)\,dp = \int_{\Pi_{ij}^{\uparrow}}^1 B(a,b)\,dp = \frac{\alpha}{2}. \tag{11.19}$$

We determine an upper bound for the probability of no return $\bar{L}_{ij}$ as the maximum-likelihood estimate of the parameter of a geometric distribution, as the observation of L_{ijk} returns implies L_{ijk} subsequent failures to *not* return in independent Bernoulli trials:

$$\bar{L}_{ij} = \frac{K}{\sum_k L_{ijk} + K}. \tag{11.20}$$

This is an upper bound as this assumes that every run terminates with a fail to return, even though the return probability might well be 1, and hence, the no-return parameter be 0. This is also the posterior mean for the conjugate prior $B(0,0)$ distribution in Bayesian inference. The posterior is the $B(a,b)$ distribution with $a = K$ and $b = \sum_k L_{ijk}$. The Bayesian $1-\alpha$ credible interval $(\bar{L}_{ij}^{\downarrow}, \bar{L}_{ij}^{\uparrow})$ specifies the uncertainty in the upper bound as

$$\int_0^{\bar{L}_{ij}^{\downarrow}} B(a,b)\,dp = \int_{\bar{L}_{ij}^{\uparrow}}^1 B(a,b)\,dp = \frac{\alpha}{2}. \tag{11.21}$$

Averaging the m-th raw moments $M_{ijk,m}$ needs to normalize to the number of returns. The straightforward way to define the average m-th raw moment as

$$M_{ij,m} = \frac{1}{K}\sum_k \frac{M_{ijk,m}}{L_{ijk}} \tag{11.22}$$

with the normal confidence interval quantifying the uncertainty. As this is prone to numerical instability especially at the critical point with a small number of return periods merging to form the last return period, an alternative definition of a "combined" average is

$$\tilde{M}_{ij,m} = \frac{1}{K}\frac{\sum_k M_{ijk,m}}{\sum_k L_{ijk}}. \tag{11.23}$$

The latter definition averages over the raw moments first and only then normalizes them by the total number of returns in all runs. To calculate a confidence interval, we employ the computationally fast Approximate Bootstrap Confidence (ABC) method. [213]

11.4 THE PYTEMPER PYTHON PACKAGE

11.4.1 About the package

pytemper is a Scientific Python package for finite-time analysis of the recurrence–transience transition in the temporal percolation paradigm. [214] I developed it to computationally implement the temporal percolation method as detailed in the preceding Section. It is intended and designed to be open sourced as free software under a permissive license. Unit tests cover about 83% of its codebase.

The pytemper package depends on the python-future, NumPy, SciPy and scikits.bootstrap packages. [61–64, 204, 215]

11.4.2 Implementation of the temporal percolation analysis

The `pytemper.single_run_statistics` routine (Listing C.8) processes a list of return periods $\tau_{ijk,l}$ for a list of finite return times T_i. It returns a structured NumPy array with finite-tif return periods $\tau_{ijk,l}$ for a list of finite return times T_i. statistics. [216] The `pytemper.statistics` routine (Listing C.9) processes statistics of single runs to yield finite-time averages. It also returns a structured NumPy array.

The pytemper packages further implements example simulators (Listing C.10) such as the random walk (`pytemper.examples.random_walk_trajectories`) or the M/M/1 queue system size Markov chain (`pytemper.examples.mm1q_trajectories`). They take as input the number of runs K, the number of steps to simulate (T_{max}), a list of control parameters ρ_j and a master seed to generate the K realizations ω_k. They return a NumPy array of finite-time trajectories.

The computational pipeline is as follows:

1. A simulator generates finite-time trajectories.
2. The trajectories are mapped onto returns periods.
3. `pytemper.single_run_statistics` map return periods to finite-time statistics for every single run
4. `pytemper.stats` process single-run statistics to form averages
5. The finite-time scaling analysis is run, e. g. with `pyfssa.autoscale`.

11.4.3 First steps with pytemper

As a dry run for the pytemper packages, let us consider a toy example of a small set of return periods:

```
>>> import pytemper

>>> ts = [1, 10, 100]
>>> return_periods = [2, 2, 2, 10, 124]

>>> single_run_stats = pytemper.single_run_statistics(
...     ts=ts,
...     return_periods=return_periods)
>>> for stat in single_run_stats.dtype.names:
...     stat, single_run_stats[stat]
...
('last_return_period', array([  1.,   4.,  84.]))
('has_spanning_cluster', array([ True, False,  True], dtype=bool))
('max_finite_return_period', array([  0.,   2.,  10.]))
```

```
('max_return_period', array([  1.,   4.,  84.]))
('number_of_returns', array([0, 3, 4], dtype=uint64))
('moments', array([[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          0.00000000e+00],
       [  6.00000000e+00,   1.20000000e+01,   2.40000000e+01,
          4.80000000e+01],
       [  1.60000000e+01,   1.12000000e+02,   1.02400000e+03,
          1.00480000e+04]]))

now exiting Console...
```

We see that as promised, the `single_run_statistics` function extracts the temporal percolation statistics for a single run. Note that while these usually increase monotonously with T_i, a spanning cluster detected at some T might disappear in a larger T' if a return period ends between T and T'. As for raw moments, all up to the fourth moment are recorded, where the first to fourth moments are stored in the `moments` field of the returned NumPy array, and the zeroth moment is the number of returns. For example, for $T = 1$ there are no returns, and hence, all moments are 0.

12 CONNECTING THE DOTS

12.1 TEMPORAL PERCOLATION OF THE RANDOM WALK

In the following, we will apply the conceptual and computational temporal percolation methodology developed in the preceding chapters to the random walk on the half-line. As we noted before, the paradigmatic M/M/1 queueing system is similar to the random walk up to the point of having the same analytical exponents of the return time distribution and the temporal percolation transition. This is why I focus on an extensive treatment of the random walk here.

Before I outline the structure of this Chapter, let me briefly consider the merging of return periods when the control parameter is increased. First, in the random walk, an increase of the control parameter p means that for a given sample path ω, at one time step n the step will flip from $X_n \to X_n - 1$ to $X_n \to X_n + 1$ (or, if $X_n = 0$, a step that flips from $0 \to 0$ to $0 \to 1$). This single flip entails a merger with all subsequent return periods (clusters) until and including the second return period of length 1. The reason for this is that a return period of length 1 means there is a step $0 \to 0$, that "consumes" one additional step as it means that it will become a step $1 \to 0$ (or $2 \to 1$). (For flips at the origin, is is until and including the next return period at length 1.) The probability that any given return period is imminent to initiate a merger with subsequent return periods is proportional to the size of the imminent return period (that initiates the merger). Let $n = 2k$ be the length of a return period for $n > 1$. Then, it has k steps $+1$ and k steps -1. As p is increased, each of the k steps -1 has an independent and identical probability to flip when p is increased by dp, which is $\frac{dp}{1-p}$. Hence, the probability for a return period to initate a merge when increasing p is proportional to its size. This is known as the microscopic mechanism of *preferential attachment* in cluster growth and has been shown to yield macroscopic power-law distribution in cluster sizes. [42, 217, 218] We retreat to this familiar observation as we note that the master equation involves sums over convolutions of variable order: when a return period initiates a merge, it merges with a random number of other return periods.

I demonstrate the computational temporal percolation method as follows. First, we inspect the finite-time return period statistics of a single realization of the critical random walk on the half-line at different finite times in Section 12.2.1. Section 12.2.2 presents finite-time return period and temporal percolation statistics of a single realization at different values of the control parameter p in the critical region. Section 12.2.3 presents finite-time temporal percolation statistics for a small ensemble of realizations, before Section 12.3 considers the average statistics of a large ensemble of realizations. Finally, Section 12.4 presents a full-scale finite-time scaling analysis of the average

Table 12.1: Single-run finite-time statistics of a critical random walk (see Figure 9.4 for the trajectories). Indices $j, k = 1$ as these are data for one single run ω at one parameter value $\rho = \frac{1}{2}$.

T_i	10^2	10^3	10^4	10^5
L_{ijk}	3	38	284	728
$\tau_{ijk,L}$	91	767	878	218
Π_{ijk}	1	1	0	0
P_{ijk}	0.91	0.77	0.29	0.57
$M_{ijk,1}$	9	233	9122	99782

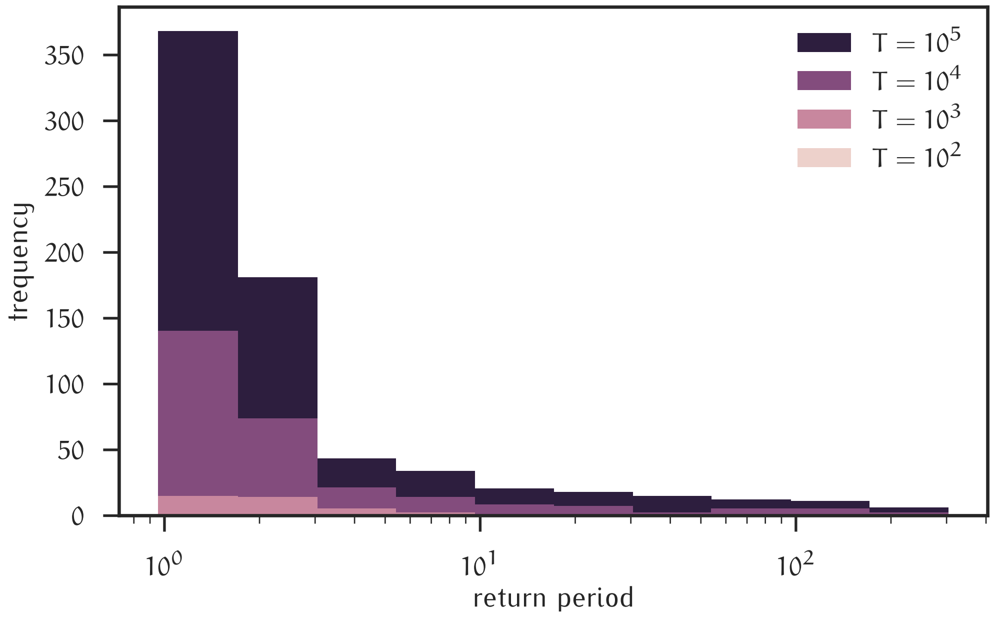

Figure 12.2: Histogram of the return periods τ_{ijk} of one realization of the critical random walk on the half-line ($p = \frac{1}{2}$) for different overall finite simulation time T_i. The histograms are overlayed and non-cumulative; e. g., the number of return periods between 1.778 and 3.162 until $T = 10^5$ is 181. The x axis and the bin edges are log-scaled. The bars are centered at the geometric mean of the bins edges.

temporal percolation statistics of a large ensemble of realizations at large system sizes.

12.2 SINGLE-RUN STATISTICS

12.2.1 Single-run statistics at finite times

Straight-forward Monte Carlo simulation yields a single realization ω of the critical random walk on the half-line ($p = \frac{1}{2}$) up to a total number of steps (or finite time) $T_{\max} = 10^5$ (see Figure 9.4 for the trajectory). As the finite time T_i grows, the number of returns L_{ijk} grows, and so do the raw moments $M_{ijk,m}$; the last return period $\tau_{ijk,L}$, whether there is a spanning cluster or not (Π_{ijk}), or the relative size of the largest return period P_{ijk} do

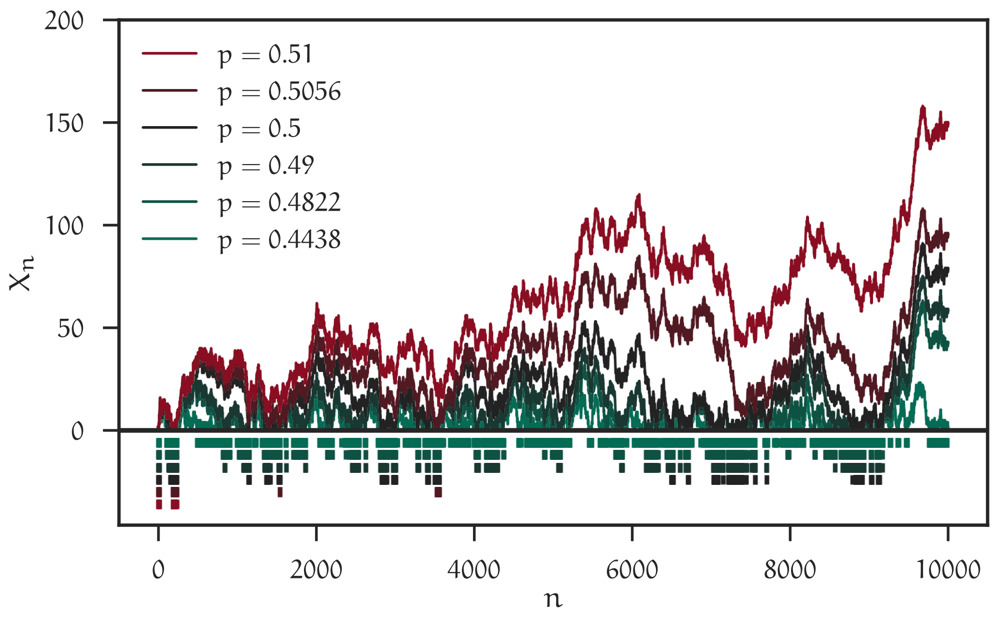

Figure 12.3: A realization of a random walk on the half-line at different values of the parameter p in the critical region. Upper panel: Finite-time trajectories. Lower panel: Vertical bars colored as the trajectory at the respective parameter value highlight returns to the origin (top to bottom: growing p, less returns).

not necessarily grow (see Table 12.1 for the full statistics). The frequencies of return periods τ_{ijk} in a histogram grow with system size T_i and are reminiscent of a power law (as expected from the analytical return time distribution at the critical point; see Figure 12.2).

12.2.2 Single runs for multiple parameter values

We extend the temporal percolation treatment from a single realization ω at the critical parameter value (and different simulation times) to a single realization ω at various values of the control parameter p in the critical region (see Figure 12.3). (In fact, I use the same realization as in the last Section.) The histograms of the return periods at different values of p feature power-law-esque behavior with finite-time and/or exponential cut-offs for critical p and below, while above $p = \frac{1}{2}$ finite return periods are small (if there are any; see Figure 12.4).

We further produce the temporal percolation statistics which exhibits familiar finite-size behavior (Figure 12.5): The larger the system size, the sharper the transition, as signified by the relative size of the largest return period P_{ijk}, the relative numbers of returns L_{ijk}/T_i, or the mean finite return period $M_{ijk,1}/L_{ijk}$. Note the second peak for large T in the mean return period (Figure 12.5, Panel E): This peak signifies a large return period remaining and growing in relative size as short return periods merge, reducing the normalization factor of the total number of returns L_{ijk}.

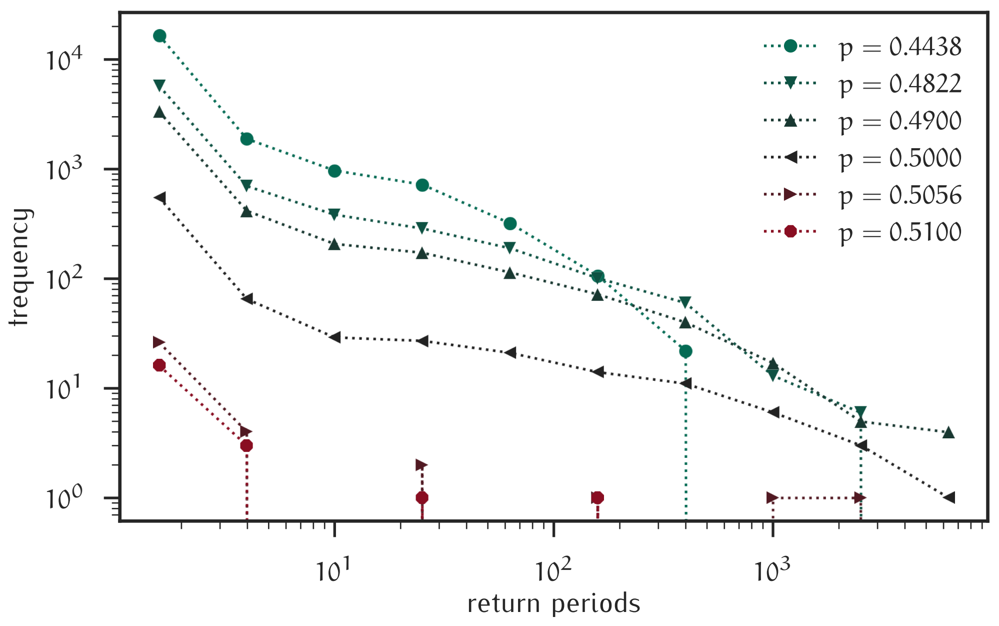

Figure 12.4: Histograms of the return periods of finite-time trajectories ($T_{max} = 10^5$) of a single realization of the random walk on the half-line for multiple parameter values p in the critical region. Both axes, the bin edges and the frequencies, are log-scaled. Missing data points signify frequency 0. The data points are centered at the geometric mean of the bin edges. The lines are a guide for the eyes.

12.2.3 Single-run statistics on multiple realizations

Once more, we extend our temporal percolation analysis: Now, we compare the temporal percolation statistics for a small number of realizations of the random walk, again at several values of the parameter p in the critical region (see Figure 12.6 and Figure 12.7).

The temporal percolation statistics of the different realizations ω_k at large finite system size are qualitatively and quantitatively similar (Figure 12.7).

12.3 FULL ENSEMBLE STATISTICS

Having carefully investigated and compared temporal percolation statistics for several finite times T_i, parameter values p_j, and realizations ω_k, we now turn to a larger ensemble of 400 realizations of the random walk on the half-line for several finite system sizes T_i up to $T_{max} = 10^4$ and several parameter values in the critical region.

The averaged histograms of the return periods resemble power laws (Figure 12.9) up to the full simulation time.

The averaged finite-time temporal percolation statistics allow to visually localize the transition as the points of intersection of the percolation probability Π_{ij} or the percolation strength P_{ij}; whereas the peak in the average return time $M_{ij,1}$ is not very pronounced (yet) and exhibits large variance (Figure 12.10). On the contrary, the combined-average finite return period $\bar{M}_{ij,1}$ is numerically stable, as expected. However, algorithmic finite-size

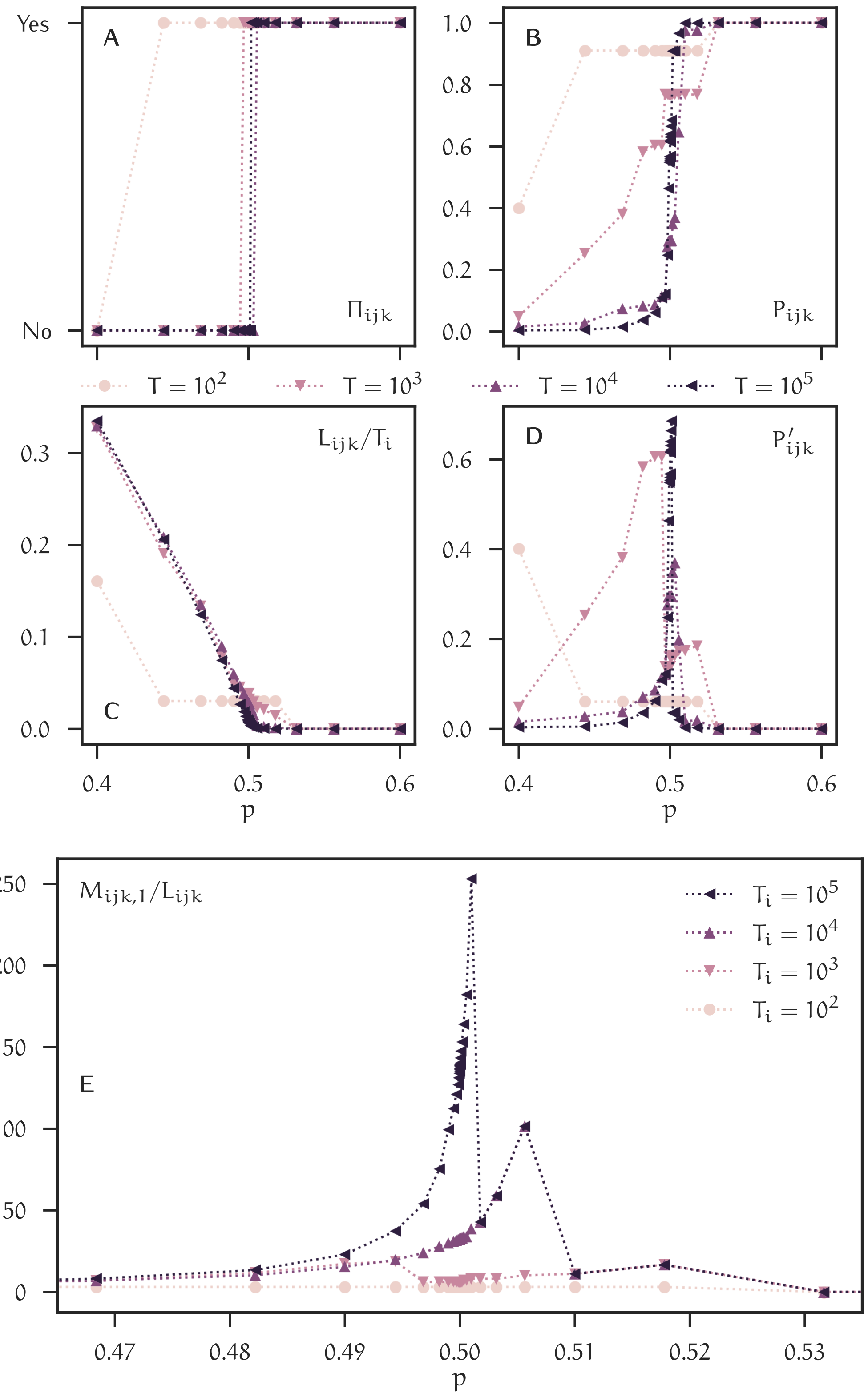

Figure 12.5: Finite-time statistics of a single realization ω_k of the random walk on the half-line for several finite system sizes T_i and parameter values p_j in the critical region. Panel **A** shows whether there is a spanning cluster or not (Π_{ijk}, where "Yes" means $\Pi_{ijk} = 1$ and "No" means $\Pi_{ijk} = 0$). Panel **B** shows the relative size of the largest return period (including the truncated last return period) P_{ijk}. Panel **C** shows the relative number of returns per time step L_{ijk}/T_i. For comparison, Panel **D** shows the relative size of the largest finite return period (excluding the truncated and possibly infinite last return period) $P'_{ijk} = \frac{1}{T_i} \max_{l \leqslant L_{ijk}} \tau_{ijk,l}$. Finally, Panel **E** shows the mean finite return period $M_{ijk,1}/L_{ijk}$. Lines are a guide for the eyes.

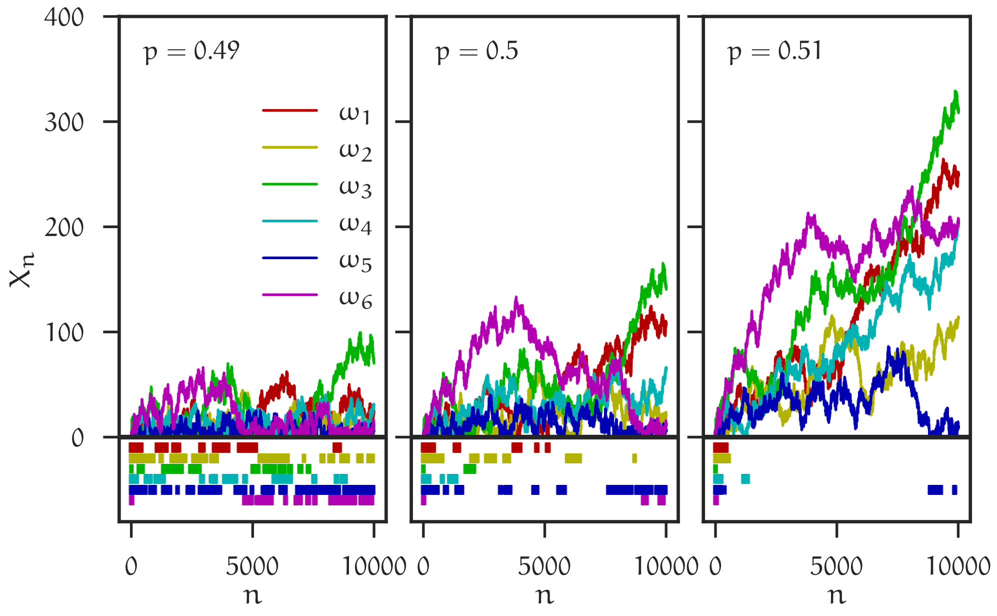

Figure 12.6: A small ensemble of 6 realizations ω_k of the random walk on the half-line at several values of the parameter p in the critical region. Upper panels: Finite-time trajectories. Lower panels: Vertical bars colored as the trajectory of the respective realization ω_k highlight returns to the origin.

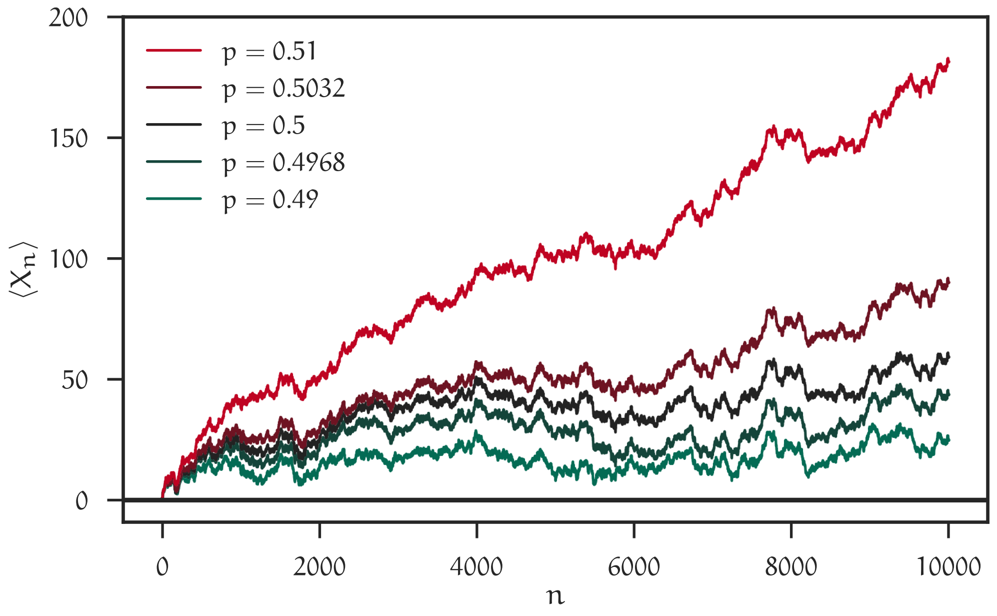

Figure 12.7: Average position $\langle X_n \rangle = \frac{1}{K} \sum_k X_n(\omega_k)$ of a small ensemble of $K = 6$ realizations of the random walk on the half-line at several values of the parameter p in the critical region (see Figure 12.6 for the individual trajectories)

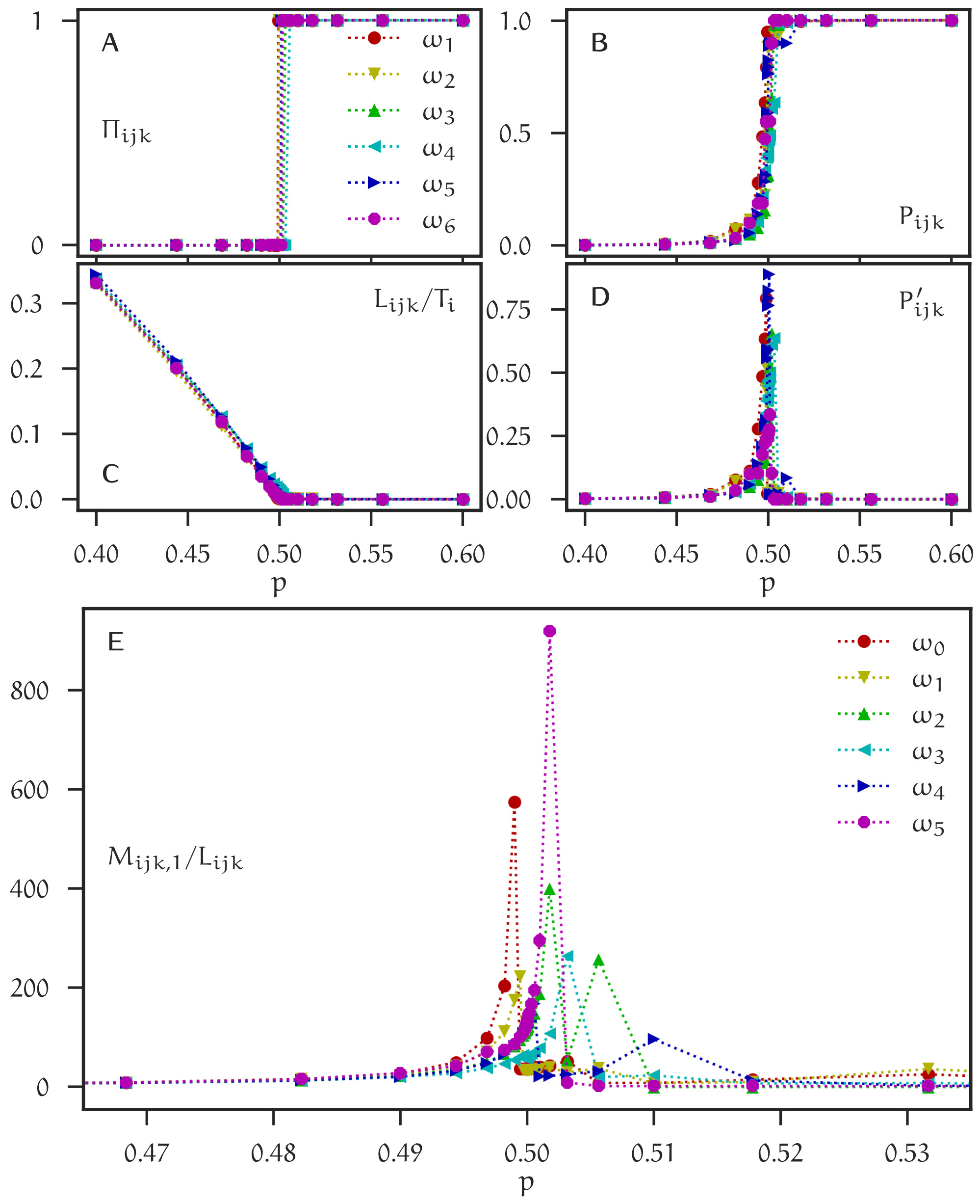

Figure 12.8: Finite-time statistics for each run ω_k of a small ensemble of $K = 6$ realizations of the random walk on the half-line at several values of the parameter p_j in the critical region (see Figure 12.6 for the individual trajectories) for finite time $T_i = 10^5$. Panel **A** shows whether there is a spanning cluster or not (Π_{ijk}). Panel **B** shows the relative size of the largest return period (including the truncated last return period) P_{ijk}. Panel **C** shows the relative number of returns per time step L_{ijk}/T_i. For comparison, Panel **D** shows the relative size of the largest finite return period (excluding the truncated and possibly infinite last return period) $P'_{ijk} = \frac{1}{T_i} \max_{l \leqslant L_{ijk}} \tau_{ijk,l}$. Finally, Panel **E** shows the mean finite return period $M_{ijk,1}/L_{ijk}$ for each realization ω_k. Note the magnification in the x scale as compared to the upper panels. Lines are a guide for the eyes.

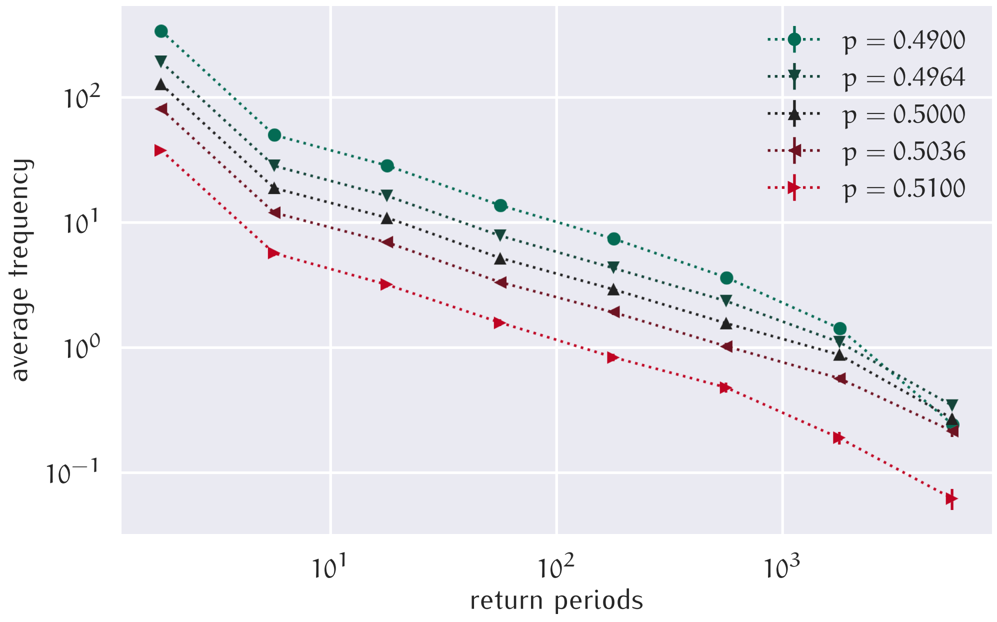

Figure 12.9: Average histograms of the return periods of finite-time trajectories of an ensemble of 400 realizations of the random walk on the half-line over a simulation time of $T = 10^4$ for multiple parameter values p in the critical region. Each data point is the average of the number of return times in the respective bin over all realizations. Both axes, the bin edges and the frequencies, are log-scaled. The data points are centered at the geometric mean of the bin edges. Missing errorbars are smaller than the marker at the respective data point. The lines are a guide for the eyes.

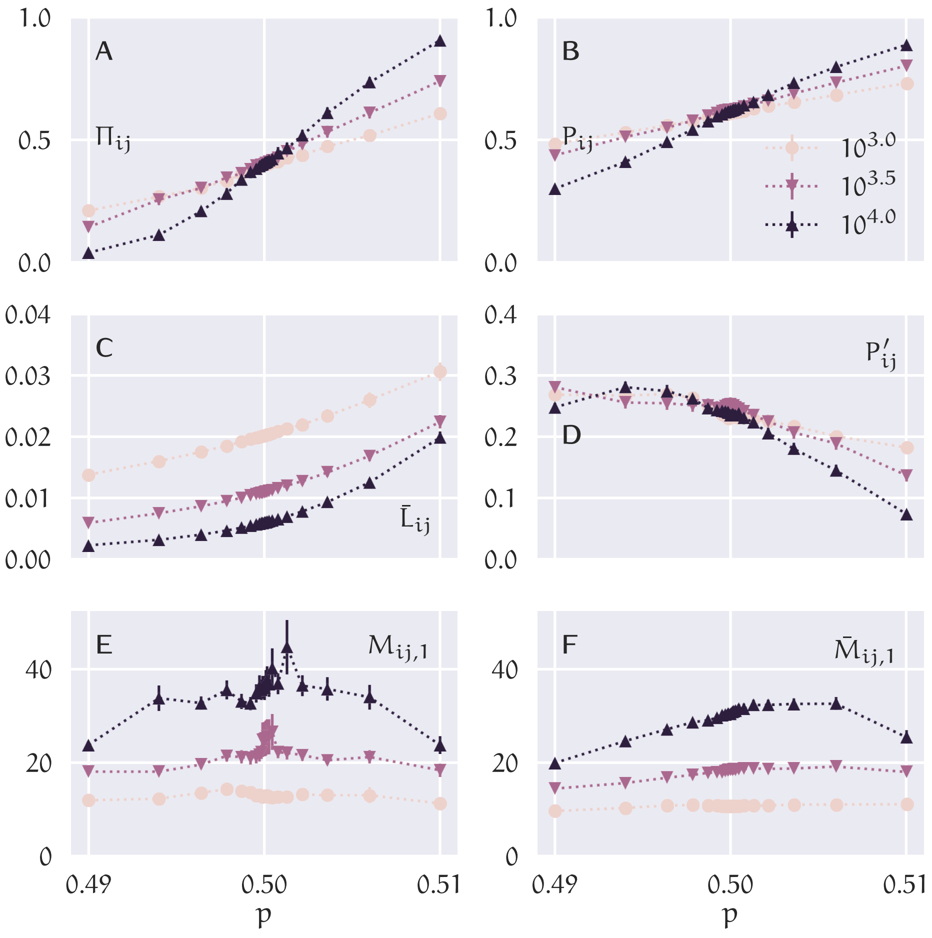

Figure 12.10: Average finite-time temporal percolation statistics of an ensemble of 400 realizations of the random walk on the half-line for several finite system sizes T_i and parameter values p_j in the critical region. Panel **A** shows the percolation probability Π_{ij}. Panel **B** shows the percolation strength P_{ij} (the average relative size of the largest return period, including the truncated last return period). Panel **C** shows the probability of no return $\bar{L}_{ij}$. For comparison, Panel **D** shows the average relative size of the largest finite return period P'_{ij} (excluding the truncated and possibly infinite last return period). Finally, Panel **E** shows the average first raw moment, or the average finite return period, $M_{ij,1}$. For comparison, Panel **F** shows the combined-average finite return period, $\bar{M}_{ij,1}$. Missing errorbars are smaller than the marker at the respective data point. Lines are a guide for the eyes.

Table 12.13: Results of the algorithmic finite-time analysis of temporal percolation statistics of a random walk on the half-line in 400 runs up to finite simulation time $T_{max} = 10^6$. The analysis yields the critical point p_c, the critical exponent ν of the temporal coherence scale and the critical exponent $\beta = \zeta$ of the percolation strength P and the critical exponent $\gamma = \zeta$ of the average return time $\bar{\tau}$, and their respective errors. For the scaling procedure, finite-time data at $T_i = 10^{5.0}, 10^{5.5}, 10^{6.0}$ were used.

	P	$\bar{\tau}$
p_c	0.4999	0.4999
dp_c	0.0002	0.0013
ν	2.0407	2.0651
$d\nu$	0.1785	0.2943
ζ	−0.0001	1.0023
$d\zeta$	0.0050	0.2337

scaling analysis needs larger system sizes to be reliable, and in particular, it needs more pronounced peak in the susceptibility (average return time).

12.4 FULL-SCALE TEMPORAL PERCOLATION ANALYSIS

We now embark on a full-scale finite-time temporal percolation scaling analysis of an ensemble of 400 realizations of the random walk on the half-line for several large system sizes T_i up to $T_{max} = 10^6$ and various parameter values p_j in the critical region.

The average temporal percolation statistics such as the percolation probability Π_{ij}, percolation strength P_{ij}, probability of no return $\bar{L}_{ij}$ and combined-average return time $\bar{M}_{ij,1}$ feature more pronounced steps or peaks than before (Figure 12.11). As before, the average first raw moment $M_{ij,1}$ remains numerically unstable.

The algorithmic finite-time analysis with pyfssa yields critical values for the critical point p_c and the scaling exponent ν of the temporal coherence length, as well as the scaling exponents β and γ for the percolation strength P and the average return time $\bar{\tau}$ (Table 12.13). The agreement of the critical point p_c with the analytical value $\frac{1}{2}$ is formidable with an absolute deviation of 10^{-4}. The numerical exponent ν deviates from the analytical value 2 by about 3 %. The scaling exponents β and γ deviate by 10^{-4} or $2 \cdot 10^{-3}$ from their analytical values 0 and 1, respectively. All analytical values lie within the error range of the numerical values. Relative errors are comparatively large ($> 5\%$) for ν and in particular for γ at 23%.

Visual inspection of the collapse of the scaled data of percolation strength P and average return time $\bar{\tau}$ confirms the numerical agreement of finite-time temporal percolation analysis with analytical values (Figure 12.12): Scaling with the analytical critical values $p_c = 0.5, \nu = 2.0, \beta = 0, \gamma = 1$ instead yields qualitatively identical plots with almost imperceptible quantitative differences.

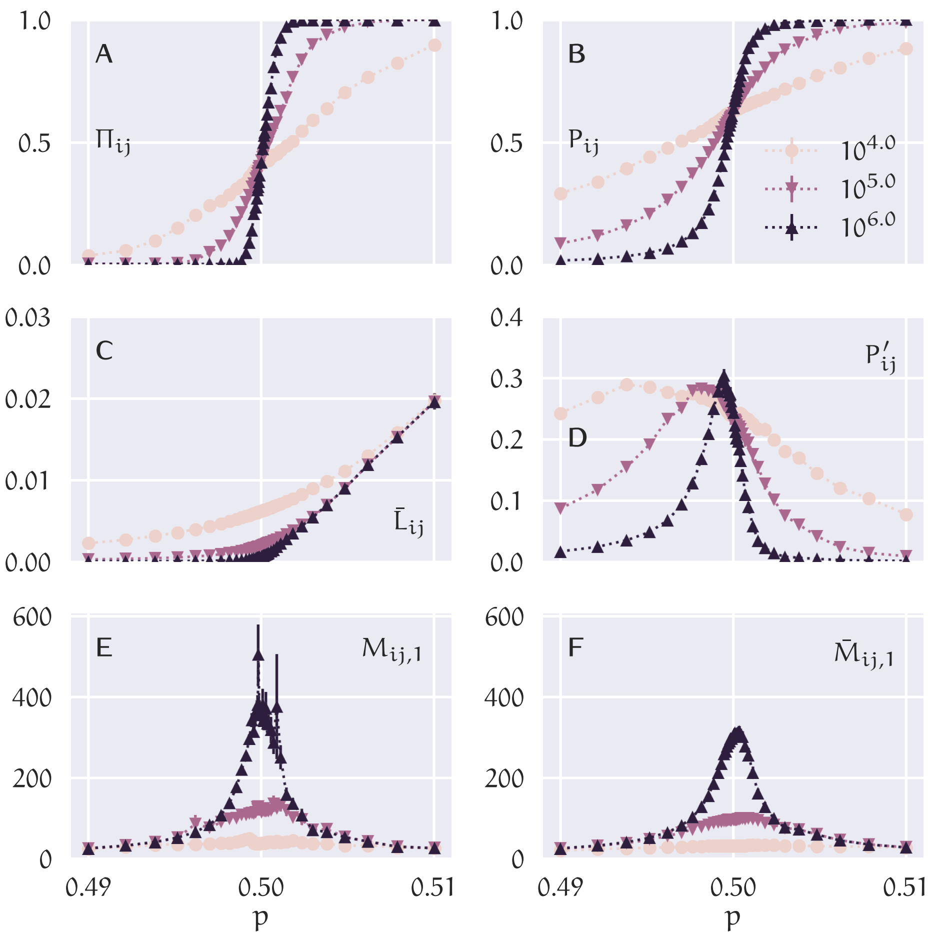

Figure 12.11: Average finite-time temporal percolation statistics of an ensemble of 400 realizations of the random walk on the half-line for several large finite system sizes T_i (cf. Figure 12.10) and parameter values p_j in the critical region. Panel **A** shows the percolation probability Π_{ij}. Panel **B** shows the percolation strength P_{ij} (the average relative size of the largest return period, including the truncated last return period). Panel **C** shows the probability of no return $\bar{L}_{ij}$. For comparison, Panel **D** shows the average relative size of the largest finite return period P'_{ij} (excluding the truncated and possibly infinite last return period). Finally, Panel **E** shows the average first raw moment, or the average finite return period, $M_{ij,1}$. For comparison, Panel **F** shows the combined-average finite return period, $\bar{M}_{ij,1}$. Missing errorbars are smaller than the marker at the respective data point. Lines are a guide for the eyes.

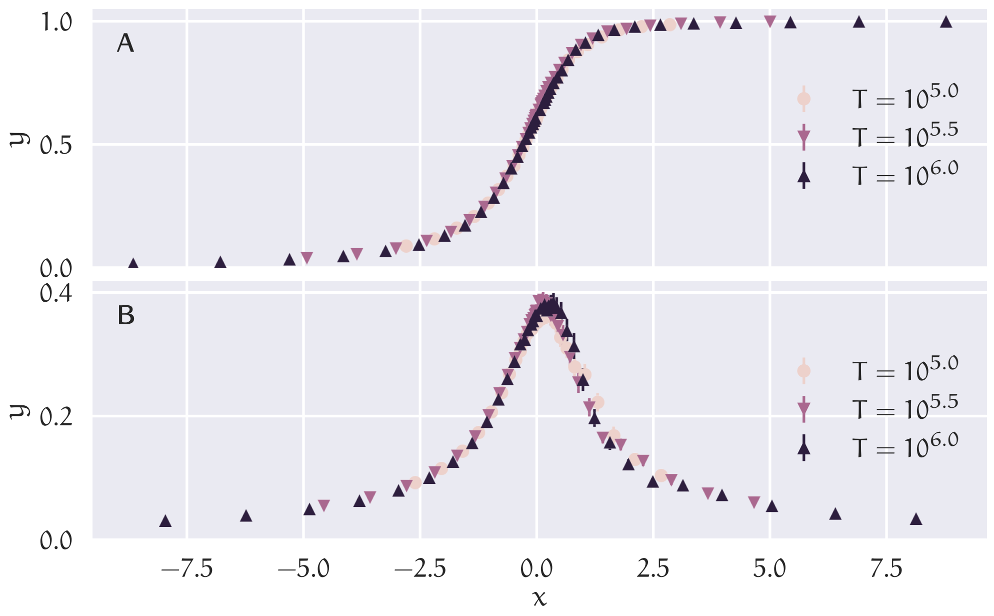

Figure 12.12: Scaled average finite-time temporal percolation statistics of an ensemble of 400 realizations of the random walk on the half-line for several large finite system sizes T_i and parameter values p_j in the critical region. Scaled with the auto-scaled exponents. Panel **A** depicts the scaled average finite-time percolation strength P_{ij} with algorithmically determined critical values $p_c = 0.4999, \nu = 2.0407, \beta = \zeta = -0.0001$. Panel **B** depicts the scaled average return time $\bar{\tau} = \bar{M}_{ij,1}$ with algorithmically determined critical values $p_c = 0.4999, \nu = 2.0651, \gamma = \zeta = 1.0023$. Missing errorbars are smaller than the marker at the respective data point.

12.5 DISCUSSION

The precedings sections have shown that the temporal percolation methodology recovers the analytical exponents of the random walk and its return times in the critical region.

The extensive numerical treatment and Monte Carlo simulation study of the random walk on the half-line in the preceding sections confirm the analytical predictions of a critical transition from recurrence to transience. I also explored the rough edges of the approach such as numerical instabilities in the susceptibility (average return time), or estimating the uncertainties, and established a framework that works.

I developed and demonstrated the temporal percolation methodology with the random walk on the half-line as the paradigmatic stochastic system exhibiting the recurrence–transience dichotomy. I introduced temporal percolation statistics such as the probability of no return that augment the classic percolation statistics to capture the peculiarities of temporal percolation on the half-line. Numerical analysis confirms that finite clusters do persist; even though the order parameter (the relative size of the largest cluster) jumps to 1 at the transition, the finite return probability signifies finite clusters even beyond the critical transition.

What I did not show so far is how exactly the Markovianness of the stochastic process, the iid return times, the exponents $\nu = 2, \tau = \frac{3}{2}, \gamma = 1$, the recurrence–transience dichotomy interrelate to give rise to critical behavior at a unique critical point. For example, does $\nu = \frac{1}{2-\tau}$ always hold in temporal percolation in one dimensions, as suggested by Maslov [169], or does it hold only for Markovian systems? Further finite-time scaling studies of stochastic systems at the transition to instability are needed, to explore the range of exponents, and ultimately, universality classes. At the same time, one could test for independence of the return times, and relate these results to the goodness of fit of the data collapse in the finite-time scaling analysis. There might also be systems that do not exhibit a sharp transition at all.

There still remains room for improving the algorithmic finite-time scaling analysis, in particular the estimation of the critical parameters and exponents. While the procedure pyfssa implements is well-tested and rests on a well-proven methodology, it remains numerically fragile. For example, the Nelder–Mead search method still needs careful manual inspection and fine-tuning of initial conditions to produce a positive definite variance–covariance matrix to define uncertainties for the estimated parameters. This so far hampers full integration into an automatic computational pipeline. Employing modern methods of statistical inference and parameter estimation such as Approximate Bayesian Computation should address these shortcomings of the current implementation.

13 CONCLUSION

In this Part, I established, implemented and illuminated the intricate connection of dynamic instablities of stochastic processes and the geometric phase transition of percolation theory. As finite clusters persist even beyond the critical point, this temporal percolation is an example of a hybrid transition in one dimension: it displays characteristics of a first-order phase transition as the order parameter of relative size of the largest cluster discontinuously grows from 0 to 1 at the critical point, while it features signature behavior of a critical transition: the return times—the fundamental quantity of temporal percolation—scale to all sizes at the critical point, as they distribute according to a power law.

This is certainly good news. Not only do we have the standard statistical physics toolkit to pinpoint and analyze such a transition in computational Monte Carlo finite-size studies available now, but also do we have a critical transition that exhibits early-warning signs (precursors) to detect and possibly mitigate such a transition. Recall that in queueing systems, such a transition to transience means congestion which is a bad thing as it entails the breakdown of proper functioning of the system.

What I also outlined is that it is possible to establish a computational pipeline to algorithmically analyze such systems. This opens up the prospect of running full-scale parameter scans of queueing systems with the throughput capacity (as determined by the critical point) automatically detected at each point in parameter space. This Work is also an endeavour in establishing reproducible computational research procedures. [219–222] In devising an algorithmic temporal percolation and finite-size analysis scheme, manual data manipulation and intervention of the numerical experimentalist is intended to be eliminated, as it is subjective and prone to not being recorded and accounted for in later proceedings. While we managed to implement semi-automatical finite-size scaling, we also discussed how to implement automatic finite-size scaling with statistical inference methods in the future.

Among the open conceptual problems that remain to be studied is how we actually measure throughput capacity of non-Markovian systems. For example, a queue more general than the paradigmatic M/M/1 queue is the G/G/1 queue with a general (not necessarily Poisson) arrival process and independent service times identically distributed according to a general (not necessarily exponential) distribution. The return times of the G/G/1 queue have been shown to also follow a power law of cumulative exponent $\frac{1}{2}$ at the critical point of full utilization $\rho = 1$ (where the average inter-arrival time equals the average service time). [223, 224] Furthermore, if the service times X_n themselves are distributed according to a power law $P\{X > x\} \sim x^{-\nu}$ with exponent $\nu \in (1, 2)$, then the return time will be distributed according to a power law $P\{\tau > t\} \sim t^{-1/\nu}$ with reciprocal exponent; the heavier-tailed the service time, the lighter-tailed is the return time. [224] I conjecture that our methodology does not rely on strict Markovianness and independence of return times. Instead, I conjecture that the stochastic dynamical system at hand only needs to ensure sufficient mixing such that a steady-state and infinitely long trajectory eventually visits all regions of state space. This is what weak, crude or semi-regeneration of the system at hand entails, which means that subsequent return periods do not necessarily need to be independent. [123, 177, 225–227] For example, in spatial queueing systems such as transport systems even an empty system retains some memory of the preceding return period in the form of the transporter positions. However, I expect the system to lose that memory after some time, and hence, to feature the recurrence–transience dichotomy of temporally percolating systems. As already discussed, finite-time scaling analysis of further systems is needed to pursue this hypothesis.

Another avenue of research is how a queueing system with time-dependent arrival rate approaches the critical transition of the steady-state (constant-rate) system, and how interventions such as rejecting requests need to be designed to actually mitigate the transition instead of leading to an even more abrupt—explosive—transition. [83, 97]

Part V

Conclusion

14 BRIEF OVERALL CONCLUSION

This Thesis demonstrates and tackles a route to the computational study of collective mobility and demand-driven transport systems by network and statistical physics domain scientists. While studies within the Demand-Driven Directed Transport (D3T) framework still involve the careful design and implementation of numerical simulation, a typical study employing existing transport spaces and dispatching policies implemented as pyd3t modules should be of the same order of complexity as a comparable study of any other network dynamical system. This immediately opens avenues of future research into the interplay of the network structure and the transport dynamics on that network. A prominent example of an emergent effect is Braess' paradox, which arises when (supposedly) enhancing the network topology by adding or strenghening a link has the counterintuitive systemic effect of less overall transport capacity. [228]

Since recently, the scientific and socioeconomic interest and need for on-demand ride-sharing and self-organizing logistic systems grows, thus demanding a thorough understanding of their intricate structural and dynamical properties. Studying collective mobility systems governed by ride-sharing dispatching policies such as those proposed by Santi, Resta, Szell, Sobolevsky, Strogatz, and Ratti [34] and Alonso-Mora, Samaranayake, Wallar, Frazzoli, and Rus [36] in the D3T framework and temporal percolation (stochastic stability) framework of this Thesis complements the data-driven engineering approach in the current literature by studying the mechanisms that lead from microscopic rules to systemic effects. For example, in the strive to avoid congestive collapse as investigated by Hyytiä, Penttinen, and Sulonen [38], one needs to carefully study precursors of the critical transition to overloading the system. When delaying such a transition by a priori localized interventions, one needs to be aware of the possibility of abrupt congestive collapse as the analogue of an explosive transition in percolation settings. [97]

Given the modelling, simulation and critical transition toolkit of this Thesis, I project basic research into collective mobility systems to chart the parameter space and the dynamical regimes of simple, myopic taxi systems to provide base models and benchmarks for any more involved dispatching policy (such as ride-sharing). This includes the dependence of system performance and dynamical properties on the transport geometry, as well as how these scale with parameters such as the request rate, the size of the transport space, the number and the capacity of the transporters. An enhanced taxi dispatching policy that proactively rejects transport requests if service quality is too poor provides another baseline for ride-sharing policies with rejections. A typical quest in these studies is to identify dimensionless quantities embodying universal scaling behavior. Emergent effects—new physics—would be signified by critical transitions or bifurcations that separate different dynamical regimes. Such systematic studies of collective mobility systems are currently being conducted by the D3T team of the Network Dynamics Group at the MPI for Dynamics and Self-Organization, employing the tools developed in this Thesis as envisaged.

Part VI

Appendix

A DISCRETE-EVENT SYSTEMS

A.1 EXAMPLES OF SIMPLE DEVS MODELS

This Section presents a few examples from Zeigler, Kim, and Praehofer [130].

A.1.1 Passive

The *passive DEVS model* has arbitrary input X which it completely ignores. It never produces output, and forever remains in its single passive state:

$$M_{\text{passive}} = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \text{ta}),$$

with

$$\begin{aligned}
S &= \{\text{passive}\},\\
\delta_{\text{int}}(\text{passive}) &= \text{passive},\\
\delta_{\text{ext}}(\text{passive}, e, x^b) &= \text{passive},\\
\delta_{\text{con}}(\text{passive}, x^b) &= \text{passive},\\
\lambda(\text{passive}) &= \emptyset\\
\text{ta}(\text{passive}) &= \infty.
\end{aligned}$$

A.1.2 Generator

Like the passive DEVS model, a *generator DEVS model* also ignores input. It autonomously *generates* certain output. In this example, the model generates a single output $Y = \{1\}$ with some period $T > 0$:

$$M_{\text{generator}} = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \text{ta}),$$

with

$$\begin{aligned}
Y &= \{1\},\\
S &= \mathbb{R}_0^+,\\
\delta_{\text{int}}(s) &= T,\\
\delta_{\text{ext}}(s, e, x^b) &= s - e,\\
\delta_{\text{con}}(s, x^b) &= T,\\
\lambda(s) &= \{1\},\\
\text{ta}(s) &= s.
\end{aligned}$$

The internal state variable s records the time until the next output event. This is important to handle external transitions due to input. The input is indeed ignored. However, the external transition resets the lifespan of the internal state. Hence, the model needs to update the internal state to reflect the reduced time until the next output event.

A.1.3 Processor

A *processor DEVS model* complements a generator: this model "processes" arbitrary input events (*jobs*) $X = \mathbb{J}$ one at a time. In this example, it takes the time $\theta > 0$ to process an arbitrary job:

$$M_{\text{processor}} = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, ta),$$

with

$$\begin{aligned}
Y = X &= \mathbb{J},\\
S &= \{\text{idle}, \text{busy}\} \times [0, \theta] \times \mathbb{J},\\
\delta_{\text{int}}(\tilde{s}, \sigma, j) &= (\text{idle}, \theta, j),\\
\delta_{\text{ext}}(\tilde{s}, \sigma, j, e, x^b) &= \begin{cases} (\text{busy}, \theta, f(x^b)) & \text{if } \tilde{s} = \text{idle}\\ (\text{busy}, \sigma - e, j) & \text{otherwise}(\tilde{s} = \text{busy}),\end{cases}
\end{aligned}$$

where $f : \mathcal{P}^X \to X$ selects one of the jobs and discards the others,

$$\begin{aligned}
f(x^b) &= x \in x^b,\\
\delta_{\text{con}}(\tilde{s}, \sigma, j, x^b) &= \delta_{\text{ext}}(\delta_{\text{int}}(\tilde{s}, \sigma, j), 0, x^b),\\
\lambda\,(\text{busy}, \sigma, j) &= \{j\},\\
ta(\tilde{s}, \sigma, j) &= \begin{cases} \sigma & \text{if } \tilde{s} = \text{busy},\\ \infty & \text{otherwise}(\tilde{s} = \text{idle}.\end{cases}
\end{aligned}$$

This model only accepts a job if it is idle: While processing, the model is busy and discards incoming jobs. After the time θ, it outputs the job and returns to the idle state. If jobs are input simultaneously, the external transition functions simply selects one of them according to some function f, and discards the others. The confluent transition functions handles collisions as follows:

1. It finishes the active job (internal transition from busy to idle phase).

2. Subsequently, it processes the submitted job (external transition from idle to busy phase).

A.2 EXAMPLE OF A COUPLED DEVS MODEL

Consider a pipeline of a generator g and a processor p DEVS model as a simple example of a DEVS coupled model. The pipeline has no input ($X_N = \emptyset$), and the same simple output as the generator (and processor): $Y_g = X_p = Y_p = Y_N = \{1\}$.

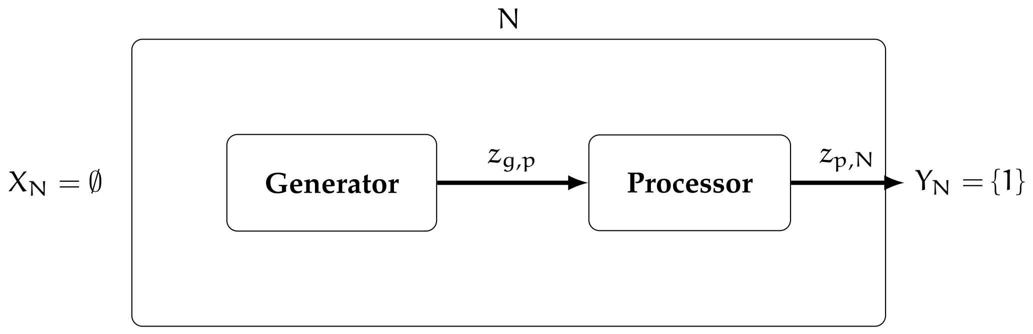

Figure A.1: An example DEVS network model.

$$\begin{aligned}
X_N &= \emptyset, \\
Y_g = X_p = Y_p = Y_N &= \{1\}, \\
D &= \{g, p\}, \\
\mathcal{I} &= \{I_g, I_p, I_N\}, \\
I_g &= \emptyset, \\
I_p &= \{g\}, \\
I_N &= \{p\}, \\
Z &= \left\{ z_{g,p}, z_{p,N} \right\}, \\
z_{g,p} &= \mathrm{id}, \\
z_{p,N} &= \mathrm{id}.
\end{aligned}$$

A.3 TIME-EVOLUTION OF A DEVS MODEL

A table-based time evolution of a DEVS model illustrates the introduced concepts and constitutes a simulator at the same time. Let M be an example DEVS model with

$$\begin{aligned}
S &= \{s_0, s_1\}, \\
\delta_{\mathrm{int}}(s_0) &= s_1, \delta_{\mathrm{int}}(s_1) = s_0, \\
\lambda(s) &= 1, \\
\mathrm{ta}(s_0) &= 0, \mathrm{ta}(s_1) = 1,
\end{aligned}$$

and no input:

Table A.2: Table-based time evolution of an example DEVS model without external input.

τ	$q(\tau)$	ta	y	x	δ	$\Delta((s_0, e_0), x)$
						$\delta((s_1, 0), x\|_{[(0,0),(2,0))})$
$(0, 0)$	$(s_1, 0)$	1	$\hat{\varnothing}$	$\hat{\varnothing}$	II	$\delta\left((s_1, 1), x\|_{[(1,0),(2,0))}\right)$
$(1, 0)$	$(s_1, 1)$	1	$\{1\}$	$\hat{\varnothing}$	III	$\delta\left((\delta_{\mathrm{int}}(s_1), 0), x\|_{[(1,1),(2,0))}\right)$
$(1, 1)$	$(s_0, 0)$	0	$\{1\}$	$\hat{\varnothing}$	III	$\delta\left((\delta_{\mathrm{int}}(s_0), 0), x\|_{[(1,2),(2,0))}\right)$
$(1, 2)$	$(s_1, 0)$	1	$\hat{\varnothing}$	$\hat{\varnothing}$	II	$\delta\left((s_1, 1), x\|_{[(2,0),(2,0))}\right)$
$(2, 0)$					I	$(s_1, 1)$

Consider the simulation of another simple example DEVS model, the processor. Let $\mathbb{J} = \{1\}$ be a trivial job set, and $\theta = 1$ be the job execution time. The initial total state is $q(\tau = (0,0)) = (s_0, e_0) = (\tilde{s}_0, \sigma_0, e_0) = (I, 1, 0)$. For simplicity, omit the trivial job variable $j = 1$ in the internal state.

Let there be input at times $\tau_1 = (1,0), \tau_2 = (2,2), \tau_3 = (2.5,0), \tau_4 = (3,0)$, and consider the time evolution of the system over the time interval $[(0,0),(3,1))$. The input trajectory segment x on this interval splits into five primitive event segments, $x = x_0 \cdot \dots \cdot x_4$.

Table A.3: Table-based time evolution of the example processor DEVS model with external input.

τ	$q(\tau)$	ta	y	x	δ	$\Delta((s_0,e_0),x)$
						$\delta((I,1,0),x_0)$
$(0,0)$	$(I,1,0)$	∞	$\hat{\varnothing}$	$\hat{\varnothing}$	I	$(I,1,1); \delta((I,1,1),x_1)$
$(1,0)$	$(I,1,1)$	∞	$\hat{\varnothing}$	$\{1\}$	III	$\delta\left((\delta_{\text{ext}}(I,1,1,\{1\}),0), x_1\|_{[(1,1),(2,2))}\right)$
$(1,1)$	$(B,1,0)$	1	$\hat{\varnothing}$	$\hat{\varnothing}$	II	$\delta\left((B,1,1), x_1\|_{[(2,0),(2,2))}\right)$
$(2,0)$	$(B,1,1)$	1	$\{1\}$	$\hat{\varnothing}$	III	$\delta\left((\delta_{\text{int}}(B,1),0), x_1\|_{[(2,1),(2,2))}\right)$
$(2,1)$	$(I,1,0)$	∞	$\hat{\varnothing}$	$\hat{\varnothing}$	I	$(I,1,0); \delta((I,1,0),x_2)$
$(2,2)$	$(I,1,0)$	∞	$\hat{\varnothing}$	$\{1\}$	III	$\delta\left((\delta_{\text{ext}}(I,1,0,\{1\}),0), x_2\|_{[(2,3),(2.5,0))}\right)$
$(2,3)$	$(B,1,0)$	1	$\hat{\varnothing}$	$\hat{\varnothing}$	I	$(B,1,0.5); \delta((B,1,0.5),x_3)$
$(2.5,0)$	$(B,1,0.5)$	∞	$\hat{\varnothing}$	$\{1\}$	III	$\delta\left((\delta_{\text{ext}}(B,1,0.5,\{1\}),0), x_3\|_{[(2.5,1),(3,0))}\right)$
$(2.5,1)$	$(B,0.5,0)$	0.5	$\hat{\varnothing}$	$\hat{\varnothing}$	I	$(B,0.5,0.5); \delta((B,0.5,0.5),x_4)$
$(3,0)$	$(B,0.5,0.5)$	0.5	$\{1\}$	$\{1\}$	III	$\delta\left((\delta_{\text{con}}(B,0.5,\{1\}),0), x_4\|_{[(3,1),(3,1))}\right)$
$(3,1)$					I	$(B,1,0)$

A.4 REDUCING A DEVS NETWORK MODEL

This Section covers the dynamical interpretation of a network model. It presents the generic procedure by Zeigler, Kim, and Praehofer [130] and Nutaro [133], to construct the resultant M_N of a network model N. The resultant M_N is an atomic DEVS model. Conveniently, the construction of the resultant yields a simulation algorithm for a network model. The ability to reduce any given network model to an atomic DEVS model is also referred to as *closure under coupling*.

A.4.0.1 *Input set and output set*

The possible input and output of the resultant are the network input set X_N and output set Y_N.

A.4.0.2 *Internal states*

The internal state of the resultant is defined by the total internal states of all network components $d \in D$:

$$S_r = \bigtimes_{d \in D} Q_d,$$

where $Q_d \subset S_d \times \overline{\mathbb{R}}_0$. Here, as in the following definitions, the property of a component which is a network model, is identified with the corresponding property of the network resultant. Hence, the internal states S_d of a component d are

$$S_d = \begin{cases} S_M & \text{if d atomic model M,} \\ S_{r'} & \text{if d network model N', r' resultant of N'.} \end{cases}$$

In general

$$(S_d, \delta_{\text{int}}^d, \delta_{\text{ext}}^d, \delta_{\text{con}}^d, \lambda_d, ta_d) = \begin{cases} (S_M, \delta_{\text{int}}{}^M, \delta_{\text{ext}}{}^M, \delta_{\text{con}}^M, \lambda_M, ta_M) & \text{if d atomic model M} \\ (S_{r'}, \delta_{\text{int}}{}^{r'}, \delta_{\text{ext}}{}^{r'}, \delta_{\text{con}}^{r'}, \lambda_{r'}, ta_{r'}) & \text{if d network model N',} \\ & \text{r' resultant of N'.} \end{cases}$$

A.4.0.3 *Time advancement function*

The lifespan of the current internal state s_r of the resultant is the minimum lifespan of the current internal states of the network components

$$ta_r(s_r) = \min_{d \in D} ta_d(s_d) - e_d.$$

Again, the time advancement function ta_d of a network component d is recursively defined as

$$ta_d = \begin{cases} ta_M & \text{if d atomic model M,} \\ ta_{r'} & \text{if d network model N', r' resultant of N'.} \end{cases}$$

A.4.0.4 *Output function*

Each internal transition of the network is triggered by an internal transition of a component. The time to the next such an event is $ta_r(s_r)$. All the components that schedule an internal transition at that time are called *imminent*:

$$\text{IMM} = \{ d \in D \mid ta_d(s_d) - e_d = ta_r(s_r) \} = \arg\min_{d \in D} ta_d(s_d) - e_d.$$

Of all the components $d \in I_N$ coupled to the network output, it is the imminent components $I_N \cap \text{IMM}$ that determine the output at this particular transition:

$$\lambda_r(s_r) = \biguplus_{d \in I_N \cap \text{IMM}} z_{d,N}(\lambda_d(s_d)),$$

where the output function λ_d of the component is again recursively defined, and the union $\biguplus$ is a sum of multisets.

A.4.0.5 *Internal transitions*

The internal transition function not only has to take internal transitions of the imminent components into account, but also external transitions of components triggered by output of the imminent components. Specifically, component $d \in D$ receives input $x^b_d \in \mathcal{P}^{X_d}$ from the imminent components $d' \in I_d \cap IMM \setminus \{N\}$:

$$x^b_d = \biguplus_{d' \in I_d \cap IMM \setminus \{N\}} z_{d',d}(\lambda_{d'}(s_{d'})).$$

Since the network state is defined component-wise, transitions also occur component-wise. An internal transition changes the total state $(s_d, e_d) \mapsto (s'_d, e'_d)$. The transition depends on whether the component d is imminent or not, and whether it receives input or not: the component undergoes an internal, external, or confluent transition, or merely advances its elapsed time:

$$\overset{r}{\delta}_{int}(s_r) = s'_r,$$
$$(s'_d, e'_d) = \begin{cases} (\delta_{int}{}^d(s_d), 0) & \text{if } d \in IMM, x^b_d = \emptyset, \\ (\delta^d_{con}(s_d, x^b_d), 0) & \text{if } d \in IMM, x^b_d \neq \emptyset, \\ (s_d, e_d + ta_r(s_r)) & \text{if } d \notin IMM, x^b_d = \emptyset, \\ (\delta_{ext}{}^d(s_d, e_d + ta(s_r), x^b_d), 0) & \text{if } d \notin IMM, x^b_d \neq \emptyset. \end{cases}$$

A.4.0.6 *External transitions*

At times when there is not any internal transition, external input x^b_r to the network triggers an external transition of the network model. The absence of a network internal transition implies the absence of component internal and confluent transitions. Thus, all components $d \in D$ coupled to the network input ($N \in I_d$), and actually receive input ($z_{N,d}(x^b_r) \neq \emptyset$), undergo an external transition. All other components merely advance their elapsed times:

$$\overset{r}{\delta}_{ext}(s_r, e_r, x^b_r) = s'_r$$
$$(s'_d, e'_d) = \begin{cases} (\delta_{ext}{}^d(s_d, e_d + e_r, z_{N,d}(x^b_r)), 0) & \text{if } N \in I_d \wedge z_{N,d}(x^b_r) \neq \emptyset, \\ (s_d, e_d + e_r) & \text{otherwise.} \end{cases}$$

A.4.0.7 *Confluent transitions*

A confluent transition of the network occurs when the network receives external input x^b_r at the time of an internal transition. Considering the definition of the confluent transition function, the only change to the internal transition function is that components $d \in D$ coupled to the network input ($N \in I_d$) have possible additional external input $z_{N,d}(x^b_r)$. For each network component $d \in D$, the *confluent input function* $\zeta_d : \mathcal{P}^{X_d} \to \mathcal{P}^{X_d}$ simply adds external input to the input x^b_d the component receives from other network components:

$$\zeta_d(x^b_d) = \begin{cases} x^b_d \uplus z_{N,d}(x^b_r) & \text{if } N \in I_d, \\ x^b_d & \text{otherwise.} \end{cases}$$

By the multiset addition $\uplus$, the confluent input function ensures hierarchical consistency [8].

Based on the internal transition function, the confluent transition function only substitutes x_d^b by $\zeta_d(x_d^b)$:

$$\delta_{con}^r(s_r, x_r^b) = s_r'$$

$$(s_d', e_d') = \begin{cases} ({\delta_{int}}^d(s_d), 0) & \text{if } d \in \text{IMM}, \zeta_d(x_d^b) = \emptyset, \\ (\delta_{con}^d(s_d, \zeta_d(x_d^b)), 0) & \text{if } d \in \text{IMM}, \zeta_d(x_d^b) \neq \emptyset, \\ (s_d, e_d + ta_r(s_r)) & \text{if } d \notin \text{IMM}, \zeta_d(x_d^b) = \emptyset, \\ ({\delta_{ext}}^d(s_d, e_d + ta(s_r), \zeta_d(x_d^b)), 0) & \text{if } d \notin \text{IMM}, \zeta_d(x_d^b) \neq \emptyset. \end{cases}$$

A.4.1 Generic transition function of a network model

The generic transition function δ_r of the resultant of a DEVS network model is:

$$\delta_r(s_r, e_r, x_r^b) = s_r'$$

$$(s_d', e_d') = \begin{cases} (\delta_d(s_d, e_d, \zeta_d(x_d^b)), 0) & \text{if } d \in \text{IMM} \vee \zeta_d(x_d^b) \neq \emptyset, \\ (s_d, e_d + ta_r(s_r)) & \text{otherwise.} \end{cases}$$

$$M_N = (X_N, Y_N, S_r, \overset{r}{\delta}_{int}, \overset{r}{\delta}_{ext}, \delta_{con}^r, \lambda_r, ta_r).$$

A.5 LIST OF DEVS SYMBOLS

Table A.4: List of symbols for DEVS modeling and simulation.

Symbol	Value / Domain	Meaning
$\emptyset$	$= \{\,\}$	the empty set, or empty bag
$\hat{\varnothing}$		the nonevent
$\mathbb{R}$		the real numbers
$\mathbb{R}_0^+$	$= [0, \infty) = \{x \in \mathbb{R} : x \geqslant 0\}$	the nonnegative real numbers
$\overline{\mathbb{R}}_0$	$= [0, \infty] = \{x \in \mathbb{R} : x \geqslant 0\} \cup \{\infty\}$	the extended nonnegative real numbers
M	$= (X, S, Y, ta, \lambda, \delta_{int}, \delta_{ext}, \delta_{con})$	*DEVS model*
X		*input set of a DEVS model*
x	$\in X$	input event
X^b	$= \{x^b : x \in x^b \Rightarrow x \in X\}$	the set of all bags on X
x^b	$\in X^b$	input bag
S		*set of states of a DEVS model*
Y		*output set of a DEVS model*
y	$\in Y$	output event
Y^b	$= \{y^b : y \in y^b \Rightarrow y \in Y\}$	the set of all bags on Y
y^b	$\in Y^b$	output bag

Continued on next page

Table A.4 – continued from previous page

Symbol	Value / Domain	Meaning
ta	$: S \to \overline{\mathbb{R}}_0$	*time advancement function of a DEVS model*
λ	$: S \to Y^b$	*output function of a DEVS model*
δ_{int}	$: S \to S$	*internal transition function of a DEVS model*
e	$\in [0, \text{ta}(s)]$	time elapsed since the last state transition
Q	$= \{ (s,e) : s \in S, 0 \leqslant e < \text{ta}(s) \} \subset S \times \overline{\mathbb{R}}_0$	*set of total states of a DEVS model*
$\overline{Q}$	$= \{ (s,e) : s \in S, 0 \leqslant e \leqslant \text{ta}(s) \} \subset S \times \overline{\mathbb{R}}_0$	*closed set of total states of a DEVS model*
δ_{ext}	$: Q \times X^b \to S$	*external transition function of a DEVS model*
δ_{con}	$: S \times X^b \to S$	*confluent transition function of a DEVS model*
t	$\in \mathbb{R}_0^+$	absolute time
δ	$: \overline{Q} \times X^b \to S$	*generic transition function of a DEVS model*
$\mathbf{X}$		set of input ports of a *DEVS model with ports*
$\mathbf{Y}$		set of output ports of a *DEVS model with ports*
p	$\in \mathbf{X} \cup \mathbf{Y}$	input port or output port of a *DEVS model with ports*
X_p		input set for input port p of a *DEVS model with ports*
Y_p		output set for output port p of a *DEVS model with ports*
v	$\in X_p \cup Y_p$	input/output event at the input/output port p of a *DEVS model with ports*
N	$= (X_N, Y_N, D, \mathcal{I}, Z)$	*DEVS network model*
X_N		*input set of a network model*
Y_N		*output set of a network model*
D		*set of components of a network model*
d	$\in D$	*component of a network model*
I_d	$\subset D \cup \{ N \} \setminus \{ d \}$	set of influencers of component d of a network model N
$\mathcal{I}$	$= \{ I_d \subset D \cup \{ N \} \setminus \{ d \} : d \in D \cup \{ N \} \}$	family of sets of influencers
$z_{d',d}$	$: Y^b_{d'} \to X^b_d$	coupling function of a network model N, coupling component d′ output to component d input
$z_{N,d}$	$: X^b_N \to X^b_d$	coupling function of a network model N, coupling network input to component d input
$z_{d',N}$	$: Y^b_{d'} \to Y^b_N$	coupling function of a network model N, coupling component d′ output to network output
Z	$= \{ z_{d',d} : d' \in I_d, d \in D \cup \{ N \} \}$	*set of coupling functions of a network model*
$\mathbb{T}$	$= \mathbb{R}_0^+ \times \mathbb{N}_0$	time base for discrete-event simulation
x	$: \mathbb{T} \to X^b \cup \{ \hat{\varnothing} \}$	*input trajectory of a DEVS model*
y	$: \mathbb{T} \to Y^b \cup \{ \hat{\varnothing} \}$	*output trajectory of a DEVS model*

B D3T FRAMEWORK

B.1 CONTINUOUS AND NETWORK TRANSPORT SPACES

Demand-driven directed transport (D3T) takes place in a physical space. The mathematical model of such a *transport space* is a *hemimetric space* $\mathcal{M}$ with a *hemimetric* d:

Definition B.1 (*hemimetric space* [145]). *A* hemimetric *is a function* $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}_0^+$ *on a set* $\mathcal{M}$ *if*
$d(x, x) = 0\ \forall x \in \mathcal{M}$ *(*reflexivity*),*
and if it satisfies the oriented triangular inequality
$d(x, z) \leqslant d(x, y) + d(y, z)$
for any $x, y, z \in \mathcal{M}$*.*
The space $\mathcal{M}$ *endowed with the hemimetric* d *is a* hemimetric space*.*

In particular, a hemimetric is a metric that does not need to be neither discernible nor symmetric.

Definition B.2 (*metric, semimetric, quasi-metric* [145]). *A* metric *is a* hemimetric *which additionally satisfies*
$d(x, y) = 0 \Rightarrow x = y$ *(*identity of indiscernibles*)*
and
$d(x, y) = d(y, x)$ *(*symmetry*).*
A semimetric *is a* hemimetric *which is symmetric. It does not need to identify indiscernibles.*
A quasi-metric *is a* hemimetric *which identifies indiscernibles. It does not need to be symmetric.*
The space $\mathcal{M}$ *endowed with a metric (semimetric, quasi-metric)* d *is a* metric space *(*semimetric space, quasi-metric space*).*

Additionally, the transport space $\mathcal{M}$ shall be either "continuous" (*geodesic*), or discrete (a *network*).

B.1.1 Geodesic hemimetric spaces

The following presentation is largely taken from Deza and Deza [145].

A geodesic hemimetric space has a strong intrinsic property: its hemimetric as a measure of "direct" distance between any two points x and y equals the length of the shortest path from x to y. The shortest path is also called a *geodesic directed segment*.

Definition B.3 (geodesic hemimetric space). *A* geodesic hemimetric space $\mathcal{M}$ *is a hemimetric space for which any two distinct points* x, y *are connected by a* geodesic directed segment *from* x *to* y*.*

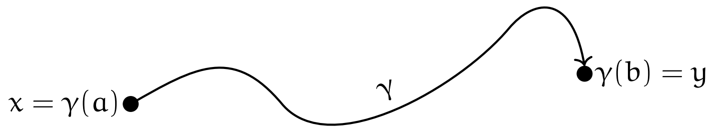

Figure B.1: An example path in the Euclidian plane.

In particular, for any intermediary point $z \in \gamma([0, d(x, y)])$ on a geodesic directed segment γ from $x = \gamma(0)$ to $y = \gamma(d(x, y))$ we have $d(x, z) + d(z, y) = d(x, y)$.

Note that the geodesic directed segment joining two points need not be unique. For example, consider antipodes on a sphere which are joined by infinitely many great circles.

Definition B.4 (Path)**.** *Let x, y be two distinct points in a hemimetric space $\mathcal{M}$. A* (directed) path *is a left-continuous and injective function $\gamma : I \to \mathcal{M}$. The interval $I = [a, b] \subset \mathbb{R}$ such that $\gamma(a) = x$ and $\gamma(b) = y$.*

Definition B.5 (Length of a path)**.** *Let $\gamma : [a, b] \to \mathcal{M}$ be a (directed) path. The* length $l(\gamma) \in \overline{\mathbb{R}}_0$ *of γ is the supremum of the sums of the distances across all directed line segments $\gamma([t_i, t_{i+1}])$. The supremum is taken over all finite decompositions $a = t_0 < t_1 < \ldots < t_n = b$ of $[a, b]$ ($n \in \mathbb{N}_0$): $l(\gamma) = \sup_{a=t_0<t_1<\ldots<t_n=b} \sum_{i=1}^{n} d(\gamma(t_{i-1}), \gamma(t_i))$*

By the triangle inequality holding for the hemimetric d, the *length* of the directed path γ from $\gamma(a)$ to $\gamma(b)$ is at least the (direct) *distance* between these points: $l(\gamma) \geqslant d(\gamma(a), \gamma(b))$.

The length does not depend on the parametrization: Let $\tilde{\gamma} : \tilde{I} = [\tilde{a}, \tilde{b}] \to \gamma(I)$ and $\gamma : I \to \gamma(I)$ be two directed paths with the same image $\gamma(I) \subset \mathcal{M}$, and the same direction, i.e. $\tilde{\gamma}(\tilde{a}) = \gamma(a), \tilde{\gamma}(\tilde{b}) = \gamma(b)$. Then we regard $\tilde{\gamma}$ and γ as the same path, only with different parametrizations $\gamma(\tilde{t}), \gamma(t)$.

Definition B.6 (Rectifiable path and normalized path)**.** *A (directed) path γ with finite length $l(\gamma) \in \mathbb{R}_0^+$ is called* rectifiable*. For such a rectifiable path $\gamma : [a, b] \to \mathcal{M}$, we define the* natural parameter *by $s = s(t) = l(\gamma|_{[a,a+t]})$, such that $s(a) = 0, s(b) = l(\gamma)$. We refer to a path with this* natural parametrization *$\gamma = \gamma(s) : [0, l(\gamma)] \to \mathcal{M}$ as* normalized *or* of unit speed*: since the path segment $\gamma([t, t'])$ for any $t, t' \in [0, l(\gamma)]$ is of length $l(\gamma|_{[t,t']}) = |t' - t|$.*

Finally, we arrive at the definition for a geodesic directed segment:

Definition B.7 (Geodesic directed segment)**.** *Let x, y be two distinct points in a hemimetric space $\mathcal{M}$. A* geodesic directed segment *or* shortest directed path *from x to y is a normalized directed path $\gamma : [0, d(x, y)] \to \mathcal{M}$ from $x = \gamma(0)$ to $y = \gamma(d(x, y))$ with length $l(\gamma) = d(x, y)$.*

B.1.1.1 *Examples*

In particular, the Euclidian space $\mathbb{R}^n$ with the standard metric $d(x, y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$ is a geodesic metric space for all $n \in \mathbb{N}$. In contrast, the n-sphere $S^n = \{x \in \mathbb{R}^{n+1} | \sum_{i=1}^{n+1} x_i = 1\}$ with the standard metric $d|_{S^n}$

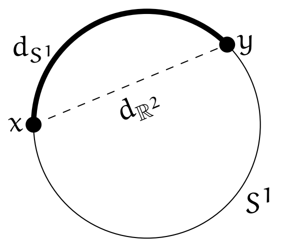

Figure B.2: The circle is a geodesic metric space under the spherical metric that measures lengths of arcs. It is not under the Euclidian metric which measures lengths of chords.

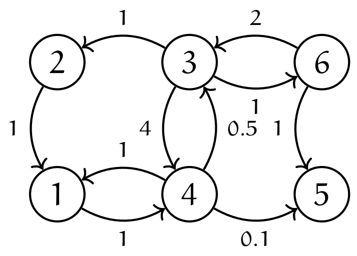

Figure B.3: A sample weighted directed digraph with 6 vertices.

restricted to S^n is still path-connected, but not geodesic: Take the border of the unit circle S^1, shortest paths on which are clearly longer as the "direct" shortcut through the inner of the circle.

Nevertheless, S^n with the *spherical metric* $d_{S^n}(x,y) = \arccos|\sum_{i=1}^{n+1} x_i y_i|$ is a geodesic metric space, since $d_{S^n}(x,y)$ is the length of the great circle arc (which are the shortest paths in spheres) joining x and y.

B.1.2 Digraphs

Definition B.8 (*Weighted digraph* [229]). *A* weighted directed graph *or* weighted digraph $G = (V, E, \omega)$ *has a finite set of* vertices V, *a finite set of* directed edges *or* arcs $E \subset V \times V$ *and a mapping* $\omega : E \to \mathbb{R}_0^+$ *which assigns a nonnegative weight* $\omega(e)$ *to each arc* e.

The digraph is simple *if it does not contain any loops (*$\forall v : (v,v) \notin E$*), nor any multiple arcs (*E *is a set, not a multiset).*

Definition B.9 (*Path* [229]). *A* path *in a digraph* $G = (V, E, \omega)$ *from vertex* v_1 *to vertex* v_n *is a finite alternating sequence* $v_1 e_1 v_2 e_2 v_3 \cdots v_{n-1} e_{n-1} v_n$ *of vertices*

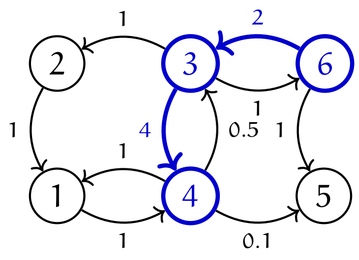

Figure B.4: A sample path from vertex 6 to vertex 4 in the sample digraph (bold vertices and arcs).

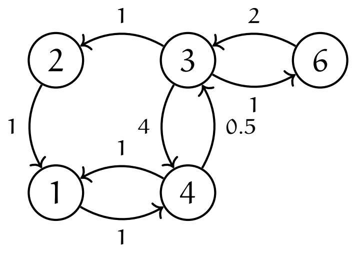

Figure B.5: Removing vertex 5 transforms the sample digraph into a network.

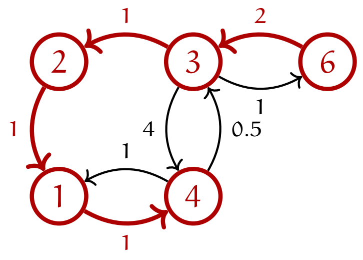

Figure B.6: Shortest path of length 5 from vertex 6 to vertex 4 in the sample network.

$v_i \in V, i \in \{1, \dots, n\}$ *and arcs* $e_i \in E, i \in \{1, \dots, n-1\}$ *for some* $n \in \mathbb{N}$, *such that* $e_i = (v_i, v_{i+1})\ \forall i \in \{1, \dots, n-1\}$ *and all vertices are pairwise distinct.*

If a path from vertex v *to vertex* v' *exists, we say that* v' *is* reachable *from* v.

Definition B.10 (*Strongly connected digraph* [229])**.** *A digraph* $G = (V, E, \omega)$ *is* strongly connected *if every vertex is reachable from any other vertex.*

B.1.3 Networks

While geodesic metric spaces allow for continuous movements along geodesic segments, networks are discrete metric spaces which only allow jumps along *paths* of links (*arcs*) between their elements (*nodes* or *vertices*).

Definition B.11 (*Network*)**.** *A* network $G = (V, E, \omega)$ *is a weighted digraph which is simple and strongly connected.*

The network is undirected, *if* $\forall e = (v, v') \in E : e' = (v', v) \in E, \omega(e') = \omega(e)$.

Definition B.12 (*path length, shortest path*)**.** *In a network* (V, E, ω), *the* length *of a path* $(v \equiv v_1 e_1 v_2 \cdots v_{n-1} e_{n-1} v_n \equiv v')$ *(for some* n*) from* v *to* v' *is the arc weight sum* $\sum_{i=1}^{n-1} \omega(e_i)$ *along the path.*

A shortest path *from the* source vertex v *to the* sink vertex v' *is a path* $(v \equiv v_1 e_1 v_2 \cdots v_{n-1} e_{n-1} v_n \equiv v')$ *(for some* n*) of minimal length of all paths from* v *to* v'.

The symbol $d_{vv'}$ *denotes the length of a shortest path from* v *to* v'.

Now all is set to endow each network $G = (V, E, \omega)$ with a network-specific hemimetric

$$d_G(v, v') = \min_{\text{paths}(v e_1 v_2 \cdots v_{n-1} e_{n-1} v'), n \in \mathbb{N}} \sum_{i=1}^{n-1} \omega(e_i),$$

yielding a hemimetric space.

It is easy to show that for any intermediary vertex u along a shortest path from v to v', we have

$$d(v,u) + d(u,v') = d(v,v').$$

Note that contrary to geodesic directed segments in geodesic hemimetric spaces, which consist of infinitely many points, a shortest path in a network only contains a finite number of vertices. Hence, for almost all numbers $\sigma \in [0, d(v,v')]$, there is not any intermediary vertex u such that $\sigma = d(u,v')$ (σ being the remaining time to v').

B.1.4 Intermediary positions

The *intermediary position function* $\nu(v,\tilde{v},\sigma)$ returns the nearest intermediary element u of the transport space $\mathcal{M}$ along the geodesic directed segment or shortest path γ from v to $\tilde{v}$, with remaining time to destination $\sigma \in [0, d(v,\tilde{v})]$:

$$\nu(v,\tilde{v},\sigma) = \underset{u\in\gamma:d(u,\tilde{v})\leqslant\sigma}{\arg\max} \ d(u,\tilde{v}).$$

The *time-to intermediary position function* $\theta(v,\tilde{v},\sigma)$ returns the time to the nearest intermediary element u:

$$\theta(v,\tilde{v},\sigma) = \sigma - d(\nu(v,\tilde{v},\sigma),\tilde{v}).$$

For geodesic transport spaces, the intermediary point u is the current position on the geodesic directed segment γ, with $\sigma = d(u,\tilde{v})$ such that $\theta(v,\tilde{v},\sigma) \equiv 0$.

For networks, the intermediary vertex u on the shortest path $\gamma = (ve_1v_2 \cdots v_{n-1}e_{n-1}\tilde{v})$ (for some n) is v_k with k such that $d(v_k,\tilde{v}) \leqslant \sigma$ and $d(v_{k-1},\tilde{v}) > \sigma$. In other words, the intermediary vertex is the next vertex to jump to along the path, with remaining jump time $\theta(v,\tilde{v},\sigma)$.

B.2 LIST OF D3T SYMBOLS

Symbol	Value/Domain	Meaning
$\emptyset$	$=\{\,\}$	the empty set
$\mathcal{F}(\cdot)$		set of the finite or empty subsets of a set $\cdot$
$\mathbb{N}$	$=\{1,2,3,\ldots\}$	the natural numbers
$\overline{\mathbb{N}}$	$=\mathbb{N}\cup\{\infty\}$	the extended natural numbers
$\mathbb{R}_0^+$	$=[0,\infty) = \{x\in\mathbb{R} : x\geqslant 0\}$	non-negative real numbers
$\mathbb{R}_+$	$=(0,\infty) = \{x\in\mathbb{R} : x> 0\}$	positive real numbers
t	$\in\mathbb{R}_0^+$	time
$\mathcal{M}$		*transport space*
d	$:\mathcal{M}\times\mathcal{M}\to\mathbb{R}_0^+$	*transport space hemimetric*

Continued on next page

Table B.7 – continued from previous page

Symbol	Value/Domain	Meaning
γ	$: [0, d(x,y)] \to \mathcal{M}$	*geodesic directed segment* between $x \in \mathcal{M}$ and $y \in \mathcal{M}$
V		*set of vertices of a digraph*
ν	$\in V$	vertex in a *digraph*
E	$\subset V \times V$	*set of arcs of a digraph*
e	$\in E$	arc in a *digraph*
ω	$: E \to \mathbb{R}_0^+$	*weight function of a digraph*
G	$= (V, E, \omega)$	*network*
$d_{\nu\nu'}$	$\in \mathbb{R}_0^+$	length of a shortest path from vertex ν to vertex ν' in a *network*
d_G	$: V \times V \to \mathbb{R}_0^+$	*network hemimetric of shortest paths*
$\tilde{\nu}$	$\in \mathcal{M}$	scheduled transporter destination
σ	$\in \mathbb{R}_0^+$	remaining travel time to scheduled destination
$\nu(\nu, \tilde{\nu}, \sigma)$	$\in \mathcal{M}$	*intermediary position function*
$\theta(\nu, \tilde{\nu}, \sigma)$	$\in [0, \sigma]$	*time-to intermediary position function*
$\mathcal{L}$		*set of loads*
l	$\in \mathbb{N}$	*load index*
$\bigcirc$	$\in \mathcal{M}$	*origin of a load*
$\square$	$\in \mathcal{M}$	*destination of a load*
t_l	$\in \mathbb{R}_0^+$	*arrival epoch* of load l
t_l^p	$\geqslant t_l$	*pick-up epoch* of load l
t_l^d	$\geqslant t_l^p$	*delivery epoch* of load l
Σ		*set of load sources*
σ	$\in \Sigma$	*load source of a load*
$(\mathcal{T}, \mathcal{Y})$	$= \{ (T_n, Y_n), n \in \mathbb{N} \}$	*load arrival process*
T_n	$\mathbb{R}_+$	random variable of the arrival epoch of the n-th load
Y_n	$\Sigma \times \mathcal{M} \times \mathcal{M} \cup \{ \nabla \}$	random variable of the mark of the n-th load
∇		the irrelevant mark
N	$\in \mathbb{N}$	*number of transporters* in a D3T model
i	$\in \{ 1, \ldots, N \}$	*transporter index*
C_i	$\in \overline{\mathbb{N}}$	*capacity* of transporter i
ν_i^0	$\in \mathcal{M}$	*initial position* of transporter i
ν	$\in \mathcal{M}$	*transporter position*
$\mathbb{L}$	$\in \mathcal{F}(\mathcal{L})$	*transporter payload*
$w(t)$	$\in \mathbb{R}_0^+$	*transporter waiting time* to arrive at next position $\nu(t)$
$\mathbb{P}$	$\in \mathcal{F}(\mathcal{L})$	*pick-up set in a transporter job*
$\mathbb{D}$	$\in \mathcal{F}(\mathcal{L})$	*delivery set in a transporter job*
j	$= (\mathbb{P}, \nu, \mathbb{D})$	*transporter job*
$\tilde{\mathbb{P}}$	$\in \mathcal{F}(\mathcal{L})$	*set of loads scheduled for pick-up*
$\tilde{\mathbb{D}}$	$\in \mathcal{F}(\mathcal{L})$	*set of loads scheduled for delivery*
$\tilde{j}$	$= (\tilde{\mathbb{P}}, \tilde{\nu}, \tilde{\mathbb{D}})$	scheduled transporter job
Q	$= (\tilde{j}_n)_n$	*transporter queue*
$\tau_P(\nu, \mathbb{P})$	$\in \mathbb{R}_0^+$	*transporter pick-up period function*
$\tau_D(\nu, \mathbb{D})$	$\in \mathbb{R}_0^+$	*transporter delivery period function*
c	$\in \{ 0, 1 \}$	*Boolean transporter cancel status*

Continued on next page

Table B.7 – continued from previous page

Symbol	Value/Domain	Meaning
M	$\in \{ \mathbf{I}, *, \mathbf{P}, \mathbf{T}, \mathbf{D}, \dagger \}$	*transporter mode*
$\mathbf{I}$		*transporter idle mode*
$*$		*transporter job start pseudo-mode*
$\mathbf{P}$		*transporter pick-up mode*
$\mathbf{T}$		*transporter travel mode*
$\mathbf{D}$		*transporter delivery mode*
$\dagger$		*transporter job end pseudo-mode*
$\mathfrak{D}$		*dispatching policy*
$\mathbf{M}$	$= \Big((\mathcal{M}, d), \Sigma, (\mathcal{T}, \mathcal{Y}), N, \{ C_i \}_{i=1}^{N}, \{ v_i^0 \}_{i=1}^{N}, \tau_P, \tau_D, \mathbf{D}_{\mathfrak{D}}, \mathfrak{O} \Big)$	*D3T model*
$\mathbf{D}_{\mathfrak{D}}$	$\in \hat{D}$	dispatcher model with dispatching policy $\mathfrak{D}$
$\hat{D}$		dispatcher class
$\mathfrak{O}$	$= \{ \mathbf{O}_j \}_j$	set of observer models
$\mathbf{O}$	$\in \hat{O}$	observer model
$\hat{O}$		observer class

C SCIENTIFIC COMPUTING CODE

C.1 COMPUTATIONAL ENVIRONMENT

Listing C.1: environment.yml, available online at `https://gitlab.gwdg.de/asorge/diss17/blob/pub/environment.yml`

```
name: diss17
channels:
- conda-forge
- defaults
dependencies:
- cairo=1.14.6=4
- fontconfig=2.12.1=4
- freetype=2.7=1
- gettext=0.19.7=1
- glib=2.51.4=0
- icu=58.1=1
- jupyter_contrib_core=0.3.0=py36_1
- jupyter_contrib_nbextensions=0.2.6=py36_0
- jupyter_highlight_selected_word=0.0.11=py36_0
- jupyter_latex_envs=1.3.8.2=py36_1
- jupyter_nbextensions_configurator=0.2.4=py36_0
- libpng=1.6.28=0
- libtiff=4.0.6=7
- matplotlib=2.0.1=np112py36_0
- openjpeg=2.1.2=2
- pandoc=1.19.2=0
- pixman=0.34.0=0
- poppler=0.52.0=2
- poppler-data=0.4.7=0
- psutil=5.2.1=py36_0
- qt=5.6.2=2
- appdirs=1.4.0=py36_0
- bleach=1.5.0=py36_0
- cloudpickle=0.2.2=py36_0
- curl=7.52.1=0
- cycler=0.10.0=py36_0
- dbus=1.10.10=0
- decorator=4.0.11=py36_0
- entrypoints=0.2.2=py36_1
- expat=2.1.0=0
- future=0.16.0=py36_1
- git=2.11.1=0
- gitdb2=2.0.0=py36_0
- gitpython=2.1.3=py36_0
- gst-plugins-base=1.8.0=0
- gstreamer=1.8.0=0
- h5py=2.7.0=np112py36_0
- hdf5=1.8.17=1
- html5lib=0.999=py36_0
- ipykernel=4.6.0=py36_0
```

```
  - ipyparallel=6.0.2=py36_0
  - ipython=5.3.0=py36_0
  - ipython_genutils=0.2.0=py36_0
  - ipywidgets=6.0.0=py36_0
  - jinja2=2.9.6=py36_0
  - jpeg=9b=0
  - jsonschema=2.5.1=py36_0
  - jupyter=1.0.0=py36_3
  - jupyter_client=5.0.1=py36_0
  - jupyter_console=5.1.0=py36_0
  - jupyter_core=4.3.0=py36_0
  - libffi=3.2.1=1
  - libgcc=5.2.0=0
  - libgfortran=3.0.0=1
  - libiconv=1.14=0
  - libsodium=1.0.10=0
  - libxcb=1.12=1
  - libxml2=2.9.4=0
  - markupsafe=0.23=py36_2
  - mistune=0.7.4=py36_0
  - mkl=2017.0.1=0
  - mpmath=0.19=py36_1
  - nbconvert=5.1.1=py36_0
  - nbformat=4.3.0=py36_0
  - networkx=1.11=py36_0
  - notebook=5.0.0=py36_0
  - numpy=1.12.1=py36_0
  - openssl=1.0.2k=1
  - pandas=0.19.2=np112py36_1
  - pandocfilters=1.4.1=py36_0
  - path.py=10.1=py36_0
  - pcre=8.39=1
  - pexpect=4.2.1=py36_0
  - pickleshare=0.7.4=py36_0
  - pip=9.0.1=py36_1
  - prompt_toolkit=1.0.14=py36_0
  - ptyprocess=0.5.1=py36_0
  - pygments=2.2.0=py36_0
  - pyparsing=2.1.4=py36_0
  - pyqt=5.6.0=py36_2
  - python=3.6.1=0
  - python-dateutil=2.6.0=py36_0
  - pytz=2017.2=py36_0
  - pyyaml=3.12=py36_0
  - pyzmq=16.0.2=py36_0
  - qtconsole=4.3.0=py36_0
  - readline=6.2=2
  - scipy=0.19.0=np112py36_0
  - seaborn=0.7.1=py36_0
  - setuptools=27.2.0=py36_0
  - simplegeneric=0.8.1=py36_1
  - sip=4.18=py36_0
  - six=1.10.0=py36_0
  - smmap2=2.0.1=py36_0
  - sqlite=3.13.0=0
  - sympy=1.0=py36_0
  - terminado=0.6=py36_0
  - testpath=0.3=py36_0
  - tk=8.5.18=0
  - tornado=4.4.2=py36_0
```

```
- traitlets=4.3.2=py36_0
- wcwidth=0.1.7=py36_0
- wheel=0.29.0=py36_0
- widgetsnbextension=2.0.0=py36_0
- xz=5.2.2=1
- yaml=0.1.6=0
- zeromq=4.1.5=0
- zlib=1.2.8=3
- pip:
  - devs==0.1.8
  - doit==0.30.3
  - fssa==0.7.6
  - git+ssh://git@gitlab.gwdg.de/pycnic/pytemper@0.3.2#egg=pytemper
  - pyinotify==0.9.6
  - scikits.bootstrap==0.3.2
  - yapf==0.16.1
  - git+ssh://git@gitlab.gwdg.de/d3t/pyd3t@v0.2.2#egg=d3t
```

C.2 D3T PYTHON PACKAGE

pyd3t version: 0.2.2

Listing C.2: d3t/loadsource.py

```python
'''

   Copyright 2014 The pyd3t Developers

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.

'''

import itertools

import devs
from d3t import RequestEvent
from d3t.components import LoadSourceClass


class LoadSourceModel(LoadSourceClass):

    MODEL_INPUT_CLASS = set()
    MODEL_OUTPUT_CLASS = set(['request'])

    new_load = itertools.count()
    def __init__(self, generator):
        """
        generator must yield tuples of length 3
        [0]: inter-arrival time
        [1]: origin
        [2]: destination
        """
        self._preloads = []
        self._next_preload = None
        self._generator = generator
        self.internal_transition()

    def output(self):
        ret = []
        for preload in self._preloads:
            load = next(LoadSourceModel.new_load)
            ret.append(
                RequestEvent(load, preload[1], preload[2])
            )
        return ret

    def time_advance(self):
        if self._preloads:
            return self._preloads[0][0]
        else:
            return devs.infinity

    def internal_transition(self):
        self._preloads = []
        if self._next_preload:
            self._preloads.append(self._next_preload)
            self._next_preload = None

        while True:
            try:
                preload = next(self._generator)
            except StopIteration:
                break
            if preload[0] > 0.0:
                if self._preloads:
                    self._next_preload = preload
                else:
                    self._preloads.append(preload)
                break
            else:
                self._preloads.append(preload)
                continue

    def external_transition(self, elapsed_time, input_events):
        raise RuntimeError(
            "Load source external transition should not be called"
        )

    def confluent_transition(self, input_events):
        self.internal_transition()
        self.external_transition(0.0, input_events)
```

Listing C.3: d3t/transporter.py

```
# -*- coding: utf-8 -*-

'''

   Copyright 2014-2015 The pyd3t Developers

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.

'''

import itertools
import logging
from collections import deque

import d3t
import devs
from d3t.components import TransporterClass


class TransporterModel(TransporterClass):

    MODEL_OUTPUT_CLASS = set([
        'init',
        'jobstart',
        'pickup',
        'departure',
        'arrival',
        'delivery',
        'jobend',
        'emptyqueue',
    ])
    MODEL_INPUT_CLASS = set(['submit', 'cancel'])

    HAS_PICKUP_PHASE = False
    HAS_DELIVERY_PHASE = False

    new_job_id = itertools.count()

    def __init__(
        self, space, id, position,
        pickup_period_function=None,
        delivery_period_function=None,
    ):
        """
        Construct new Transporter instance.

        Parameters
        ----------
        space
```

```
            Transport space

        """
        self._logger = logging.getLogger(
            'd3t.' + self.__class__.__name__ + '.{}'.format(id)
        )
        self._logger.debug('Set up logging.')
        self._space = space
        self._id = id
        self._position = position
        self._ita = 0.0
        self.HAS_INITIAL_PHASE = (
            'init' in self.OUTPUT_PORTS
            or 'emptyqueue' in self.OUTPUT_PORTS
        )
        self._pickup_period_function = pickup_period_function
        self._delivery_period_function = delivery_period_function
        self.HAS_PICKUP_PHASE = (pickup_period_function is not None)
        self.HAS_DELIVERY_PHASE = (delivery_period_function is not None)
        self._initial = self.HAS_INITIAL_PHASE
        self._idle = not self.HAS_INITIAL_PHASE
        self._pickup = False
        self._travel = False
        self._delivery = False
        self._cancel = False
        self._payload = set()
        self._scheduled_pickup_set = set()
        self._scheduled_destination = position
        self._scheduled_delivery_set = set()
        self._queue = deque()
        self._current_job_id = None
        self._next_job_id = next(TransporterModel.new_job_id)

    def _is_delivery_imminent(self):
        ret = bool(
            self._travel and self._scheduled_delivery_set and not self._cancel
        )
        self._logger.debug('Delivery imminent: {}'.format(ret))
        return ret

    def _is_delivery(self):
        ret = (
            self._delivery if self.HAS_DELIVERY_PHASE
            else self._is_delivery_imminent()
        )
        self._logger.debug('Delivery: {}'.format(ret))
        return ret

    def _is_job_end(self):
        ret = (
            (self._pickup and self._cancel)
            or
            (self._travel and not self._is_delivery_imminent())
            or
            self._is_delivery()
        ) if self.HAS_DELIVERY_PHASE else self._travel
        self._logger.debug('Job end: {}'.format(ret))
        return ret
    def _is_job_start(self):
        ret = bool(
            self._queue and (self._is_job_end() or self._idle)
        )
        self._logger.debug('Job start: {}'.format(ret))
        return ret

    def _next_pickup_set(self):
        ret = (
            self._queue[0][0] if self._is_job_start()
            else self._scheduled_pickup_set
        )
        self._logger.debug('Next pick-up set: {}'.format(ret))
        return ret

    def _is_pickup_imminent(self):
        ret = bool(
            self._next_pickup_set() if self._is_job_start() else False
        )
        self._logger.debug('Pick-up imminent: {}'.format(ret))
        return ret

    def _is_pickup(self):
        ret = (
            self._pickup if self.HAS_PICKUP_PHASE
            else self._is_pickup_imminent()
        )
        self._logger.debug('Pick-up: {}'.format(ret))
        return ret

    def _is_travel_imminent(self):
        ret = (
            (self._is_job_start() and not self._is_pickup_imminent())
            or
            (self._pickup and not self._cancel)
        ) if self.HAS_PICKUP_PHASE else self._is_job_start()
        self._logger.debug('Pick-up imminent: {}'.format(ret))
        return ret

    def internal_transition(self):

        self._logger.debug('Internal transition')

        if self.HAS_INITIAL_PHASE:
            if self._initial:
                self._initial = False
                self._idle = True
                self._logger.info('INITIAL -> IDLE')
                return

        delivery_imminent = self._is_delivery_imminent()
        delivery = self._is_delivery()
        job_end = self._is_job_end()
        job_start = self._is_job_start()
        next_pickup_set = self._next_pickup_set()
        pickup_imminent = self._is_pickup_imminent()
        pickup = self._is_pickup()
        travel_imminent = self._is_travel_imminent()

        # FROM transition
        if pickup:
            self._payload |= next_pickup_set
            self._logger.info('Pick-up loads {}, new payload: {}'.format(
                next_pickup_set, self._payload
            ))

        if self._travel:
            self._position = self._scheduled_destination
            self._logger.info('Arrive at position {}'.format(self._position))

        if delivery:
            self._payload -= self._scheduled_delivery_set
            self._logger.info('Deliver loads {}, new payload: {}'.format(
                self._scheduled_delivery_set, self._payload
            ))

        # NEXTJOB transition
        if job_start:
            next_job = self._queue.popleft()
            (
                self._scheduled_pickup_set,
                self._scheduled_destination,
                self._scheduled_delivery_set,
            ) = next_job
            self._current_job_id = self._next_job_id
            self._next_job_id = next(TransporterModel.new_job_id)
            self._logger.info(
                'Start new job {}: {}'.format(self._current_job_id, next_job)
            )

        # TO transition
        if self.HAS_PICKUP_PHASE and pickup_imminent:
            self._ita = self._pickup_period_function(
                self._position,
                self._scheduled_pickup_set
            )
            self._pickup = True
            self._idle = self._travel = self._delivery = False
            self._cancel = False
            return

        if travel_imminent:
            self._ita = self._space.distance(
                self._position, self._scheduled_destination
            )
            self._travel = True
            self._idle = self._pickup = self._delivery = False
            self._cancel = False
            self._logger.info('-> TRAVEL to destination {}, ETA {}'.format(
                self._scheduled_destination,
                self._ita
            ))
            return

        if self.HAS_DELIVERY_PHASE and delivery_imminent:
            self._ita = self._delivery_period_function(
                self._position,
                self._scheduled_delivery_set
            )
            self._delivery = True
            self._travel = False
            return

        if job_end and not self._queue:
            self._ita = devs.infinity
            self._idle = True
            self._pickup = self._travel = self._delivery = False
            self._cancel = False
            self._logger.info('-> IDLE')
            return

        raise RuntimeError("Transporter delta_int error")

    def external_transition(self, elapsed_time, input_events):

        self._ita = max(self._ita - elapsed_time, 0.0)

        if 'cancel' in self.INPUT_PORTS:
            # look for cancel event
            for event in input_events:
                if event.type == 'cancel':
                    # received cancel event
                    self._cancel = True
                    self._queue.clear()
                    if self._travel:
                        self._scheduled_destination, self._ita = (
                            self._space.intermediate_point(
                                self._position,
                                self._scheduled_destination,
                                self._ita
                            )
                        )

        # look for exactly one submit event
        for event in input_events:
            if event.type == 'submit':
                jobs = event[1]
                if event[2]:  # replace?
                    self._queue.clear()
                self._queue.extend(jobs)

    def confluent_transition(self, input_events):
        self.internal_transition()
        self.external_transition(0.0, input_events)

    def output(self):

        ret = list()

        current_position = (
            self._scheduled_destination if self._travel else self._position
        )

        next_destination = (
            self._queue[0][1]
            if self._is_job_start() and self._is_travel_imminent()
            else self._scheduled_destination
        )

        current_payload = set()
        if {'jobstart', 'jobend'} & self.OUTPUT_PORTS:
            if (
                ('jobstart' in self.OUTPUT_PORTS and self._is_job_start())
                or
                ('jobend' in self.OUTPUT_PORTS and self._is_job_end())
            ):
                current_payload |= self._payload

                if self._is_delivery():
                    current_payload -= self._scheduled_delivery_set

                if self.HAS_PICKUP_PHASE and self._pickup:
                    current_payload |= self._scheduled_pickup_set

        if 'init' in self.OUTPUT_PORTS and self._initial:
            ret.append(d3t.InitEvent(self._id, current_position))

        if 'jobstart' in self.OUTPUT_PORTS and self._is_job_start():
            ret.append(d3t.JobstartEvent(self._id, self._next_job_id))

        if 'pickup' in self.OUTPUT_PORTS and self._is_pickup():
            job_id = (
                self._next_job_id
                if self._is_job_start()
                else self._current_job_id
            )
            ret.append(d3t.PickupEvent(
                self._id, job_id, self._next_pickup_set()
            ))

        if 'departure' in self.OUTPUT_PORTS and self._is_travel_imminent():
            job_id = (
                self._next_job_id
                if self._is_job_start()
                else self._current_job_id
            )
            ret.append(d3t.DepartureEvent(
                self._id, job_id, next_destination, self._space.distance(
                    current_position, next_destination
                )
            ))

        if 'arrival' in self.OUTPUT_PORTS and self._travel:
            ret.append(d3t.ArrivalEvent(
                self._id, self._current_job_id, self._scheduled_destination
            ))

        if 'delivery' in self.OUTPUT_PORTS and self._is_delivery():
            ret.append(d3t.DeliveryEvent(
                self._id, self._current_job_id, self._scheduled_delivery_set
            ))

        if 'jobend' in self.OUTPUT_PORTS and self._is_job_end():
            ret.append(d3t.JobendEvent(self._id, self._current_job_id))

        if 'emptyqueue' in self.OUTPUT_PORTS:
            if (
                (self._is_job_end() and not self._queue)
                or self._initial
            ):
                ret.append(d3t.EmptyqueueEvent(self._id))

        self._logger.info('Output events {}'.format(ret))
        return ret

    def time_advance(self):
        if self._idle:
            return 0.0 if self._queue else devs.infinity
        if self.HAS_INITIAL_PHASE and self._initial:
            return 0.0
        return self._ita
```

Listing C.4: d3t/dispatchers/myopic_taxi_fcfs_nearest_transporter_dispatcher.py

```
# -*- coding: utf-8 -*-

#   Copyright 2015 The pyd3t Developers
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.

import copy
import logging
from collections import deque, namedtuple

import d3t
import devs
from d3t.components import DispatcherClass
from d3t.events import events_dict, Job


class MyopicTaxiFCFSNearestTransporterDispatcherModel(DispatcherClass):

    MODEL_OUTPUT_CLASS = set([
        'submit', 'assign', 'busy', 'idle'
    ])
    MODEL_INPUT_CLASS = set([
        'request', 'init', 'arrival', 'emptyqueue'
    ])

    Request = namedtuple(
        typename='Request',
        field_names=['load', 'origin', 'destination'],
    )

    Assignment = namedtuple(
        typename='Assignment',
        field_names=['load', 'transporter', 'origin', 'destination'],
    )
```

```python

    def __init__(self, space):
        self._space = space
        self._positions = dict()
        self._pending_requests = deque()
        self._idle_transporters = set()
        self._new_idle_transporters = set()
        self._new_transporters = set()
        self._assignments = list()

        self._logger = logging.getLogger('d3t.' + self.__class__.__name__)
        self._logger.debug('Set up logging.')

    def __str__(self):
        return (
            "Myopic Taxi FCFS Nearest-Transporter (MTFN)"
        )

    def _assigned_transporters(self):
        """
        Return the set of transporters in self._assignments
        """
        return {
            assignment.transporter
            for assignment in self._assignments
        }

    def time_advance(self):
        return (
            0.0 if self._new_idle_transporters or self._assignments
            else devs.infinity
        )

    def output(self):
        ret = []

        if 'submit' in self.OUTPUT_PORTS:
            ret.extend(
                d3t.SubmitEvent(
                    transporter=assignment.transporter,
                    jobs=[
                        Job(
                            pickup=set(),
                            destination=assignment.origin,
                            deliver=set()
                        ),
                        Job(
                            pickup={assignment.load},
                            destination=assignment.destination,
                            deliver={assignment.load},
                        )
                    ],
                    replace=False,
                )
                for assignment in self._assignments
            )

        if 'assign' in self.OUTPUT_PORTS:
            ret.extend(
                d3t.AssignEvent(
                    load=assignment.load,
                    transporter=assignment.transporter,
                )
                for assignment in self._assignments
            )

        if 'busy' in self.OUTPUT_PORTS:
            transporters = (
                self._assigned_transporters() & (
                    (
                        self._idle_transporters
                        - self._new_idle_transporters
                    )
                    |
                    self._new_transporters
                )
            )
            ret.extend(
                d3t.BusyEvent(transporter=transporter)
                for transporter in transporters
            )

        if 'idle' in self.OUTPUT_PORTS:
            transporters = (
                self._new_idle_transporters
                - self._assigned_transporters()
            )
            ret.extend(
                d3t.IdleEvent(transporter) for transporter in transporters
            )

        self._logger.info('Output events {}'.format(ret))
        return ret

    def _internal_transition_idle_transporters(self):
        self._logger.info('Clear idle transporters')

        self._idle_transporters -= self._assigned_transporters()

        self._logger.info('Idle transporters: {}'.format(
            self._idle_transporters
        ))

        self._new_idle_transporters.clear()
        self._new_transporters.clear()

    def _internal_transition_assignments(self):
        self._logger.info('Clear assignments')
        del self._assignments[:]

    def internal_transition(self):

        # idle transporters
        self._internal_transition_idle_transporters()

        # assignments
        self._internal_transition_assignments()

    def _external_transition_init(self, input_events):
        self._new_transporters = {event.transporter for event in input_events}
        self._positions.update({
            event.transporter: event.position for event in input_events
        })
        self._logger.info('Add transporters: {}'.format(
            self._new_transporters
        ))

    def _external_transition_arrival(self, input_events):

        self._positions.update({
            event.transporter: event.position for event in input_events
        })

    def _external_transition_request(self, input_events):

        for event in input_events:
            self._pending_requests.append(
                self.Request(
                    load=event.load,
                    origin=event.origin,
                    destination=event.destination,
                )
            )

    def _external_transition_emptyqueue(self, input_events):
        self._new_idle_transporters = {
            event.transporter for event in input_events
        }
        self._idle_transporters |= self._new_idle_transporters

        self._logger.info('New idle transporters: {}'.format(
            self._new_idle_transporters
        ))

    def _match_transition(self):

        # only execute this function if there are idle transporters AND pending
        # requests to match
        if not self._idle_transporters or not self._pending_requests:
            return

        # make a copy of idle transporters
        remaining_idle_transporters = copy.copy(self._idle_transporters)

        while remaining_idle_transporters and self._pending_requests:

            # match next request
            request = self._pending_requests.popleft()

            # function to return the distance to the load origin
            def distance_to_load_origin(transporter):
                return self._space.distance(
                    self._positions[transporter], request.origin
                )

            # implement the argmin
            nearest_transporter = min(
                remaining_idle_transporters,
                key=distance_to_load_origin
            )
            # match request with nearest transporter
            self._assignments.append(
                self.Assignment(
                    load=request.load,
                    transporter=nearest_transporter,
                    origin=request.origin,
                    destination=request.destination,
                )
            )
            remaining_idle_transporters.remove(nearest_transporter)

    def external_transition(self, elapsed_time, input_events):

        events = events_dict(input_events)

        # init transition
        self._external_transition_init(events['init'])

        # arrival transition
        self._external_transition_arrival(events['arrival'])

        # request transition
        self._external_transition_request(events['request'])

        # emptyqueue transition
        self._external_transition_emptyqueue(events['emptyqueue'])

        # match transition
        self._match_transition()

    def confluent_transition(self, input_events):
        self.internal_transition()
        self.external_transition(0.0, input_events)
```

C.3 DEVS PYTHON PACKAGE

pydevs version: 0.1.8

Listing C.5: devs/devs.pyx, available online at `https://github.com/andsor/pydevs/blob/v0.1.8/devs/devs.pyx`

```
'''

   Copyright 2014 The pydevs Developers

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.

'''

from cpython.ref cimport PyObject, Py_INCREF, Py_XINCREF, Py_CLEAR, Py_DECREF
cimport cython.operator as co
cimport cadevs
import logging
import sys
import warnings

logger = logging.getLogger(__name__)

ctypedef cadevs.PythonObject PythonObject
ctypedef cadevs.Time Time
ctypedef cadevs.Port Port
ctypedef cadevs.CPortValue CPortValue
ctypedef cadevs.IOBag CIOBag
ctypedef cadevs.IOBagIterator CIOBagIterator
ctypedef cadevs.CDevs CDevs
ctypedef cadevs.Components CComponents
ctypedef cadevs.ComponentsIterator CComponentsIterator

infinity = sys.float_info.max


cdef class IOBag:
    """
    Python extension base type that wraps an existing C++ I/O bag

    For constant bags, only the internal pointer _thisconstptr is used.
    For non-const bags, both internal pointers _thisconstptr and _thisptr are
    used.
    """
    cdef CIOBag* _thisptr
    cdef const CIOBag* _thisconstptr
    cdef bint _is_const

    cpdef unsigned int size(self):
        return self._thisconstptr.size()

    cpdef bint empty(self):
        return self._thisconstptr.empty()

    def __iter__(self):
        """
        Generator to iterate over elements in bag

        http://docs.python.org/3/library/stdtypes.html#generator-types

        Return (port, value) tuples upon each iteration
        """

        # get first and last element
        cdef CIOBagIterator it = self._thisconstptr.begin()
        cdef CIOBagIterator end = self._thisconstptr.end()

        cdef const CPortValue* pv

        while it != end:
            pv = &co.dereference(it)
            yield pv.port, <object>(pv.value)
            co.preincrement(it)


cdef object CreateOutputBag(CIOBag* bag):
    """
    Create a new Python object output bag wrapped around an existing C++ I/O
    bag
    http://stackoverflow.com/a/12205374/2366781
    """
    output_bag = OutputBag()
    output_bag._thisptr = bag
    output_bag._thisconstptr = bag
    output_bag._is_const = False
    return output_bag


cdef class OutputBag(IOBag):
    """
    Python extension type that wraps an existing C++ I/O bag

    To construct an instance in Cython, use the CreateOutputBag factory
    function.

    When inserting port/values, increase the reference counter for Python
    objects.
    """

    cpdef insert(self, int port, object value):
        pyobj = <PythonObject>value
        self._thisptr.insert(CPortValue(port, pyobj))
        Py_XINCREF(pyobj)


cdef object CreateInputBag(const CIOBag* bag):
    """
    Create a new Python object input bag wrapped around an existing C++ I/O bag
    http://stackoverflow.com/a/12205374/2366781
    """
    input_bag = InputBag()
    input_bag._thisconstptr = bag
    input_bag._is_const = True
    return input_bag


cdef class InputBag(IOBag):
    """
    Python extension type that wraps an existing C++ I/O bag
    """
    pass


cdef class AtomicBase:
    """
    Python extension type, base type for DEVS Atomic Model

    Python modules subclass this type and overwrite the methods

    How does it work?
```

```
-----------------

When initialized, the constructor (__init__) creates a new instance
of the underlying C++ wrapper class Atomic (defined in the C++ header
file).
The C++ wrapper class Atomic inherits from adevs::Atomic and implements
all the virtual functions.
The C++ wrapper instance receives the function pointers to the cy_*
helper functions defined here, as well as a pointer to the Python extension
type instance.
Whenever adevs calls one of the virtual functions of the C++ wrapper
instance, the C++ wrapper instance routes it via the function pointer to
the corresponding cy_* helper function.
The cy_* helper function calls the corresponding method of the instance of
the Python extension type.

http://stackoverflow.com/a/12700121/2366781
https://bitbucket.org/binet/cy-cxxfwk/src


Reference counting
------------------

When initialized, the constructor (__init__) creates a new instance of the
underlying C++ wrapper class Atomic (defined in the C++ header file).
Upon adding the model to a Digraph, the Digraph increases the reference
count to this Python object, and decreases the reference count upon
destruction.
Note that the adevs C++ Digraph instance assumes ownership of the C++
wrapper instances.
The C++ Digraph instance deletes all C++ wrapper instances upon destruction.
So the Python object might still exist even though the C++ wrapper
instance is long gone.
When adevs deletes the C++ wrapper instance, the Python object is not
deleted, when it is still referenced in the Python scope, but we can live
with that.


Input/output
------------
The port type is integer.
The value type is a generic Python object.
This Python wrapper class abstracts away the underlying adevs C++ PortValue
type.

adevs creates (copies) the C++ PortValue instance.
https://github.com/smiz/adevs/blob/aae196ba660259ac32fc254bad810f4b4185d52f/
    include/adevs_digraph.h#L194
https://github.com/smiz/adevs/blob/aae196ba660259ac32fc254bad810f4b4185d52f/
    include/adevs_bag.h#L156

The only interface we need is to iterate over input (InputBag) in delta_ext
and delta_conf, and to add output events (OutputBag) in output_func.
Adding output events, the instance of this Python wrapper class increases
the reference counter of the value Python object.
The C++ wrapper class decreases the reference counter upon adevs' call to
the gc_output garbage collection function.

We deliberately break the adevs interface for the output_func method.
In adevs, a reference to a Bag is supplied to the method returning void.
```

```
    Here, we choose the Pythonic way and take the return value of the method as
    the output bag.
    This is converted automatically by the cy_output_func helper function.
    output_func can either return
        None (no output),
        a tuple (of length 2: port, value),
        or an iterable (of tuples of length 2: port, value).
    For example, output_func can be implemented as a generator expression.

    Similarly, the cy_delta_ext and cy_delta_conf helper functions convert the
    input bag to a Python list of port, value tuples.
    """

    cdef cadevs.Atomic* base_ptr_
    cdef object _logger

    def __cinit__(self, *args, **kwargs):
        logger.debug('Initialize AtomicBase (__cinit__)...')
        self.base_ptr_ = new cadevs.Atomic(
            <PyObject*>self,
            <cadevs.DeltaIntFunc>cy_delta_int,
            <cadevs.DeltaExtFunc>cy_delta_ext,
            <cadevs.DeltaConfFunc>cy_delta_conf,
            <cadevs.OutputFunc>cy_output_func,
            <cadevs.TaFunc>cy_ta,
        )
        logger.debug('Initialized AtomicBase (__cinit__).')
        logger.debug('Set up logging for new AtomicBase instance...')
        self._logger = logging.getLogger(__name__ + '.AtomicBase')
        self._logger.debug('Set up logging.')

    def __dealloc__(self):
        if self.base_ptr_ is NULL:
            logger.debug('AtomicBase: Internal pointer already cleared.')
        else:
            logger.debug('AtomicBase: Deallocate internal pointer...')
            del self.base_ptr_
            logger.debug('AtomicBase: Deallocated internal pointer.')

    def _reset_base_ptr(self):
        self._logger.debug('Reset internal pointer')
        self.base_ptr_ = NULL

    def delta_int(self):
        warn_msg = 'delta_int not implemented'
        self._logger.warning(warn_msg)
        warnings.warn(warn_msg)

    def delta_ext(self, e, xb):
        warn_msg = 'delta_ext not implemented'
        self._logger.warning(warn_msg)
        warnings.warn(warn_msg)

    def delta_conf(self, xb):
        warn_msg = 'delta_conf not implemented'
        self._logger.warning(warn_msg)
        warnings.warn(warn_msg)

    def output_func(self):
        warn_msg = 'output_func not implemented, return None'
```

```
            self._logger.warning(warn_msg)
            warnings.warn(warn_msg)
            return None

        def ta(self):
            warn_msg = 'ta not implemented, return devs.infinity'
            self._logger.warning(warn_msg)
            warnings.warn(warn_msg)
            return infinity


cdef void cy_delta_int(PyObject* object) except *:
    logger.debug('Cython delta_int helper function')
    cdef AtomicBase atomic_base = <AtomicBase>object
    atomic_base.delta_int()


cdef void cy_delta_ext(
    PyObject* object, cadevs.Time e, const cadevs.IOBag& xb
) except *:
    logger.debug('Cython delta_ext helper function')
    cdef AtomicBase atomic_base = <AtomicBase>object

    # wrap the C++ Bag in a Python Wrapper Bag class
    cdef InputBag input_bag = CreateInputBag(&xb)

    atomic_base.delta_ext(e, list(input_bag))


cdef void cy_delta_conf(
    PyObject* object, const cadevs.IOBag& xb
) except *:
    logger.debug('Cython delta_conf helper function')
    cdef AtomicBase atomic_base = <AtomicBase>object

    # wrap the C++ Bag in a Python Wrapper Bag class
    cdef InputBag input_bag = CreateInputBag(&xb)

    atomic_base.delta_conf(list(input_bag))


cdef void cy_output_func(
    PyObject* object, cadevs.IOBag& yb
) except *:
    logger.debug('Cython output_func helper function')
    cdef AtomicBase atomic_base = <AtomicBase>object

    # wrap the C++ Bag in a Python Wrapper Bag class
    cdef OutputBag output_bag = CreateOutputBag(&yb)

    output = atomic_base.output_func()

    if output is None:
        logger.debug('output_func returns None')
        return

    if type(output) is tuple:
        logger.debug('output_func returns tuple')
        if len(output) != 2:
            err_msg = (
                'output_func needs to return tuple of length 2, got length {}'
            ).format(len(output))
            logger.error(err_msg)
            raise ValueError(err_msg)
        output_bag.insert(output[0], output[1])
        return

    try:
        iterator = iter(output)
    except TypeError:
        raise ValueError

    for port, value in output:
        output_bag.insert(port, value)


cdef Time cy_ta(
    PyObject* object
) except *:
    logger.debug('Cython ta helper function')
    cdef AtomicBase atomic_base = <AtomicBase>object

    return atomic_base.ta()


cdef class Digraph:
    """
    Python extension type that wraps the C++ wrapper class for the adevs
    Digraph class

    Design decision
    ---------------
    For now, we only provide Atomic models.
    I.e. nested network models are not supported yet.

    Memory management
    -----------------
    An instance of the C++ Digraph class takes ownership of added components,
    i.e. deletes the components at the end of its lifetime.
    This is why we increase the reference count to the Python object as soon as
    we add it to the Digraph.
    Upon deletion of the Digraph, the reference count is decreased.
    https://github.com/smiz/adevs/blob/aae196ba660259ac32fc254bad810f4b4185d52f/
        include/adevs_digraph.h#L205
    """
    cdef cadevs.Digraph* _thisptr
    cdef object logger

    def __cinit__(self):
        logger.debug('Initialize Digraph...')
        self._thisptr = new cadevs.Digraph()
        logger.debug('Initialized Digraph.')

    def __init__(self):
        logger.debug('Set up logging for new Digraph instance...')
        self.logger = logging.getLogger(__name__ + '.Digraph')
        self.logger.debug('Set up logging.')

    def __dealloc__(self):
        self.logger.debug('Temporarily store the Python objects')
        components = list(self)
        self.logger.debug('Deallocate internal pointer...')
        # this deletes all C++ Atomic models (and in turn, the references to
        # the Python objects)
        del self._thisptr
        self.logger.debug('Deallocated internal pointer.')
        self.logger.debug('Decrease reference counts of all Python objects')
        for component in components:
            Py_DECREF(component)
            component._reset_base_ptr()

    cpdef add(self, AtomicBase model):
        self.logger.debug('Add model...')
        self.logger.debug('Increase reference counter to Python object')
        Py_INCREF(model)
        self._thisptr.add(model.base_ptr_)
        self.logger.debug('Added model.')

    cpdef couple(
        self,
        AtomicBase source, Port source_port,
        AtomicBase destination, Port destination_port,
    ):
        self._thisptr.couple(
            source.base_ptr_, source_port,
            destination.base_ptr_, destination_port,
        )

    def __iter__(self):
        """
        Generator to iterate over components of the digraph

        http://docs.python.org/3/library/stdtypes.html#generator-types

        Return AtomicBase Python objects upon each iteration
        """

        self.logger.debug("Start iteration")
        cdef CComponents components
        self._thisptr.getComponents(components)

        # get first and last element
        cdef CComponentsIterator it = components.begin()
        cdef CComponentsIterator end = components.end()

        cdef cadevs.Atomic* component
        cdef PyObject* c_python_object
        cdef object python_object

        while it != end:
            self.logger.debug("Retrieve next component")
            component = <cadevs.Atomic*>(co.dereference(it))
            self.logger.debug("Get C Python object")
            c_python_object = <PyObject*>(component.getPythonObject())
            self.logger.debug("Cast to Python object")
            python_object = <object>c_python_object
            self.logger.debug("Yield Python object")
            yield python_object
            self.logger.debug("Increment iterator")
            co.preincrement(it)
        self.logger.debug("Stop iteration")


cdef class Simulator:
    """
    Python extension type that wraps the adevs C++ Simulator class

    Memory management
    -----------------
    Note that the adevc C++ Simulator class does not assume ownership of the
    model.
    Hence, when using a Python wrapper Simulator instance, we need to keep
    the Python wrapper Digraph or AtomicBase-subclassed instance in scope as
    well.
    When the model Python instance goes out of scope, the internal C++ pointer
    gets deleted, rendering the Simulator defunct.
    """
    cdef cadevs.Simulator* _thisptr
    cdef object logger
    cdef object sim_logger

    def __cinit__(self):
        pass

    def __init__(self, object model):
        logger.debug('Initialize Simulator...')

        if isinstance(model, AtomicBase):
            if type(model) is AtomicBase:
                error_msg = (
                    'Model is AtomicBase instance, use a subclass instead'
                )
                logger.error(error_msg)
                raise TypeError(error_msg)
            logger.debug('Initialize Simulator with atomic model')
            self._thisptr = new cadevs.Simulator((<AtomicBase>model).base_ptr_)
            logger.info('Initialized Simulator with atomic model')
        elif isinstance(model, Digraph):
            logger.debug('Initialize Simulator with digraph')
            self._thisptr = new cadevs.Simulator((<Digraph>model)._thisptr)
            logger.info('Initialized Simulator with digraph')
        else:
            raise TypeError

        self.logger = logging.getLogger(__name__ + '.Simulator')
        self.logger.debug('Set up logging.')

    def __dealloc__(self):
        self.logger.debug('Deallocate internal pointer...')
        del self._thisptr
        self.logger.debug('Deallocated internal pointer.')

    def next_event_time(self):
        self.logger.debug('Compute time of next event')
        return self._thisptr.nextEventTime()

    def execute_next_event(self):
        self.logger.info('Execute next event')
        self._thisptr.executeNextEvent()
    def execute_until(self, Time t_end):
        self.logger.info('Execute until time {}'.format(t_end))
        self._thisptr.executeUntil(t_end)


logger.debug('devs imported.')
```

C.4 PYTEMPER PYTHON PACKAGE

pytemper version: 0.3.2

Listing C.6: pytemper/__init__.py

```
# encoding: utf-8

r"""
Finite-time analysis of the recurrence--transience transition

Implements the algorithmic finite-time analysis of the recurrence--transience
transition within the "temporal percolation" paradigm.

This package provides these high-level functions from the :mod:`temper.temper`
module:

.. autosummary::

   temper.analyze_temper

See Also
--------

temper.temper : low-level functions

"""

from __future__ import absolute_import

from .singlerun import statistics as single_run_statistics
from .singlerun import epochs2periods, periods2epochs
from .temper import merge_runs
from .temper import analyze_temper
from .stats import statistics

import pkg_resources

try:
    __version__ = pkg_resources.get_distribution(__name__).version
except:
    __version__ = 'unknown'
```

Listing C.7: pytemper/temper.py

```
# coding: utf-8

"""
Low-level routines for analyzing the recurrence--transience transition

See Also
--------

temper : The high-level module

"""


# Python 2/3 compatibility
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

from builtins import dict

import collections
import warnings

import numpy as np
import scipy
import scipy.stats


SPANNING_CLUSTER_THRESHOLD_FRACTION = 1. - np.exp(-1.)
alpha_1sigma = 2 * scipy.stats.norm.cdf(-1.0)

Histograms = collections.namedtuple(
    typename='Histograms',
    field_names=['counts', 'edges'],
)


def check_ts(ts):
    """
    Helper function to validate `ts`

    Check that all times are finite and not negative
    """
    ts = np.asanyarray(ts)
    if ts.ndim > 1:
        raise ValueError
    if not np.all(np.isfinite(ts)):
        raise ValueError
    if not np.all(ts > 0.0):
        raise ValueError
    if not ts.size:
        raise ValueError
    return ts


def check_omegas(omegas):
    """
    Helper function to validate `omegas`

    Check that all omegas are finite and not negative
    """
    omegas = np.asanyarray(omegas)
    if omegas.ndim > 1:
        raise ValueError
    if not np.all(np.isfinite(omegas)):
        raise ValueError
    if np.any(omegas < 0):
        raise ValueError
    if not omegas.size:
        raise ValueError
    return omegas


def merge_runs(run_statistics):
    """
    Merge the statistics of several runs

    Merges also the return period distributions if present.

    Parameters
    ----------
    run_statistics: list
        List of outputs of :py:func:`single_run_statistics`

    Returns
    -------
    ret : dict
        Result dictionary with ``ret.keys() == ['has_spanning_cluster',
        'max_return_period', 'number_of_returns', 'moments']``. If the `hist`
        option was set to ``True``, there is another key
        ``return_period_histogram``.
        The values are ndarrays. The first axis corresponds to the run.
        The further axes are the outputs of the
        :py:func:`single_run_statistics` function.

    See also
    --------
    single_run_statistics

    """

    warnings.warn(
        "Use of merge_runs is unpythonic and deprecated. Use np.stack instead.",
        DeprecationWarning)
    return np.stack(run_statistics)


def _get_single_run_statistics(
    rho, omegas, ts, t_max,
    return_epoch_generator, generator_options, dist=False
):
    """
    Compute single-run statistics for several realizations

    Helper function for :func:`analyze_temper`

    See Also
    --------

    analyze_temper
    """

    for omega in omegas:

        # initialize generator
        return_epochs_gen = return_epoch_generator(
            rho=rho, omega=omega, t_max=t_max, **generator_options
        )
        return_epochs = list(return_epochs_gen)
        yield single_run_statistics(
            ts, return_epochs, dist=dist
        )


def _get_merged_single_run_statistics(
    rho, omegas, ts, t_max,
    return_epoch_generator, generator_options, dist=False
):
    """
    Merge single-run statistics

    Helper function for :func:`analyze_temper`

    See Also
    --------

    _get_single_run_statistics

    analyze_temper

    """

    single_run_statistics = list(_get_single_run_statistics(
        rho=rho, ts=ts, t_max=t_max, omegas=omegas, dist=dist,
        return_epoch_generator=return_epoch_generator,
        generator_options=generator_options,
    ))

    return merge_single_run_statistics(
        omegas=omegas,
        ts=ts,
        single_run_statistics=single_run_statistics,
    )


def _get_single_rho_statistics(
    rho, omegas, ts, t_max,
    return_epoch_generator, generator_options, dist=False
):
    """
    Compute temper statistics for a single control parametr value

    Helper function for :func:`analyze_temper`

    See Also
    --------

    _get_merged_single_run_statistics

    analyze_temper
```

```python
    """

    merged_single_run_statistics = _get_merged_single_run_statistics(
        rho=rho, t_max=t_max, omegas=omegas, ts=ts, dist=dist,
        return_epoch_generator=return_epoch_generator,
        generator_options=generator_options,
    )

    return single_rho_statistics(
        return_period_dist_bootstrap_samples=1,
        ts=ts,
        merged_single_run_statistics=merged_single_run_statistics,
    )


def analyze_temper(
    rhos,
    omegas,
    ts,
    return_epoch_generator,
    generator_options=None,
):
    """
    Analyze temporal percolation statistics

    Deliver statistics for the temporal percolation transition for a given
    dynamical system

    Parameters
    ----------

    rhos : 1-D array_like
        control parameters

    omegas : 1-D array_like
        seeds for the random number generator (realizations)

    ts : 1-D array_like
        finite simulation times

    return_epoch_generator
        A generator function with signature ``rho, omega, t_max, **kwargs``.

    generator_options : dict, optional
        Options to pass as keyword arguments to the `return_epoch_generator`.
        Default is ``None``.
    """

    t_max = np.max(ts)

    if generator_options is None:
        generator_options = dict()

    single_rho_statistics = [
        _get_single_rho_statistics(
            rho=rho, t_max=t_max, omegas=omegas, ts=ts, dist=False,
            return_epoch_generator=return_epoch_generator,
            generator_options=generator_options,
        )
```

```
            for rho in rhos
        ]

        return merge_single_rho_statistics(
            single_rho_statistics
        )


    def merge_single_rho_statistics(
        single_rho_statistics
    ):
        """
        Merge the single-parameter statistics for all parameter values

        Does not merge the return period distributions.

        Parameters
        ----------
        single_rho_statistics: list
            List of outputs of :py:func:`single_rho_statistics`

        Returns
        -------
        ret : dict
            Result dictionary with ``ret.keys() == ['percolation_prob',
            'percolation_prob_ci', 'percolation_strength',
            'percolation_strength_ci', 'no_return_prob_ubound',
            'no_return_prob_ubound_ci', 'moments', 'moments_ci']``.
            The values are ndarrays. The first axis corresponds to the parameter.
            The further axes are the outputs of the
            :py:func:`single_rho_statistics` function.

        See also
        --------
        single_rho_statistics

        """

        stats = [
            ('percolation_prob', 1),
            ('percolation_prob_ci', 2),
            ('percolation_strength', 1),
            ('percolation_strength_ci', 2),
            ('no_return_prob_ubound', 1),
            ('no_return_prob_ubound_ci', 2),
            ('moments', 2),
            ('moments_ci', 3),
        ]

        ret = dict()

        for stat, dim in stats:
            if dim == 1:
                ret[stat] = np.vstack((
                    single_rho[stat] for single_rho in single_rho_statistics
                ))
            elif dim == 2:
                ret[stat] = np.rollaxis(
                    np.dstack((
                        single_rho[stat] for single_rho in single_rho_statistics
                    )),
                    2
                )
            elif dim == 3:
                ret[stat] = np.concatenate([
                    single_rho[stat] for single_rho in single_rho_statistics
                ]).reshape([-1] + list(single_rho_statistics[0][stat].shape))
```

Listing C.8: pytemper/singlerun.py

```
# coding: utf-8

"""
Low-level routines for analyzing the recurrence--transience transition in a
single run

See Also
--------

temper : The high-level module

"""


# Python 2/3 compatibility
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

from builtins import dict

import numpy as np

from .temper import SPANNING_CLUSTER_THRESHOLD_FRACTION, check_ts

RUN_DTYPE = [
    (np.str(field[0]),) + field[1:]
    for field in [
        ('last_return_period', np.float),
        ('has_spanning_cluster', np.bool),
        ('max_finite_return_period', np.float),
        ('max_return_period', np.float),
        ('number_of_returns', np.uint),
        ('moments', np.float, 4),
    ]
]


def check_return_periods(return_periods):
    """
    Helper function to validate `return_periods`

    Check that all return periods are finite and positive
    """
    return_periods = np.asanyarray(return_periods)
    if return_periods.ndim > 1:
        raise ValueError
    if not np.all(np.isfinite(return_periods)):
        raise ValueError
```

```python
    if not np.all(return_periods > 0.0):
        raise ValueError
    return return_periods


def check_return_epochs(return_epochs):
    """
    Helper function to validate `return_epochs`

    Check that all return epochs are finite, ordered unique, and positive
    """
    return_epochs = np.asanyarray(return_epochs)
    if return_epochs.ndim > 1:
        raise ValueError
    if not np.all(np.isfinite(return_epochs)):
        raise ValueError
    if not np.all(return_epochs > 0.0):
        raise ValueError
    try:
        np.testing.assert_equal(np.unique(return_epochs), return_epochs)
    except:
        raise ValueError
    return return_epochs


def epochs2periods(return_epochs):
    """
    Convert a single-run return epoch sequence to a sequence of return periods

    The return periods are counted from the initial time instant `0`, no matter
    whether that is a return epoch or not!

    Parameters
    ----------
    return_epochs: 1-D ndarray
        an increasing sequence of return epochs

    Returns
    -------
    1-D ndarray
        sequence of return periods
    """

    if return_epochs.size:
        ret = np.ediff1d(
            return_epochs,
            to_begin=return_epochs[0] if return_epochs[0] > 0.0 else None
        )
    else:
        ret = return_epochs

    return ret


def periods2epochs(return_periods):
    """
    Convert a single-run return period sequence to a sequence of return epochs

    Parameters
    ----------
    return_periods: 1-D ndarray
        an sequence of return periods

    Returns
    -------
    1-D ndarray
        sequence of return epochs
    """

    return np.cumsum(return_periods)


def last_return_epoch_index(ts, return_epochs):
    """
    Determine the indices of the last return epochs of a single run

    Returns an array of indices of the last return epochs of the same shape as
    `ts`.
    The index is ``-1`` if there is no such return epoch within the finite time
    `t`.

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    return_epochs: 1-D ndarray
        an increasing sequence of return epochs

    Returns
    -------
    1-D ndarray
        indices of the last return epochs for each `t` in `ts`
    """
    return np.searchsorted(return_epochs, ts, side='right') - 1


def last_return_epoch(ts, return_epochs, last_return_index):
    """
    Determine the last return epochs of a single run

    Returns an array of last return epochs of the same shape as `ts`.
    The return epoch is ``-1`` if there is no such return epoch within the
    finite time `t`.

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    return_epochs: 1-D ndarray
        an increasing sequence of return epochs

    last_return_index: 1-D ndarray
        the output of :py:func:`last_return_epoch_index`

    Returns
    -------
    1-D ndarray
        last return epochs for each `t` in `ts` with the same type as
        `return_epochs` if they are floats, otherwise (signed) integers
    """

    # Return the same data type as return epochs if they are floats
    # If they are not floats, they could be unsigned integers
    # However, we need the value of -1, so in that case we choose the default
    # (signed) integer type
    ret_dtype = (
        return_epochs.dtype
        if np.issubdtype(return_epochs.dtype, float)
        else np.int_
    )
    ret = - np.ones_like(ts, dtype=ret_dtype)

    # determine the indices of the finite times t that have a return epoch
    # (and hence, have a last return epoch)
    with_return_epoch_indices = (last_return_index >= 0).nonzero()[0]

    # set the last return epochs for all finite times t that have a return
    # epoch
    if with_return_epoch_indices.size:
        ret[with_return_epoch_indices] = return_epochs[
            last_return_index[with_return_epoch_indices]
        ]

    return ret


def last_return_period(ts, last_return_epoch):
    """
    Determine the (truncated) last return periods of a single run

    Returns an array of last return periods of the same shape as `ts`.
    If there is no return epoch, the whole finite time `t` is counted as the
    "last" return period.
    The return period is counted from the initial time instant `0`, no matter
    whether it was a return epoch or not!
    The last return period is `0` if and only if the finite time `t` is a
    return epoch.

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    last_return_epoch: 1-D ndarray
        the output of :py:func:`last_return_epoch`

    Returns
    -------
    1-D ndarray
        last return periods for each `t` in `ts`
    """

    return np.where(
        last_return_epoch >= 0,
        ts - last_return_epoch,
        ts
    )
```

```python
def number_of_returns(last_return_index):
    """
    The number of returns in a single run, excluding the initial "return" at
    time instant `0` (if present at all)

    Parameters
    ----------
    last_return_index: 1-D ndarray
        the output of :py:func:`last_return_epoch_index`

    Returns
    -------
    1-D ndarray
        number of returns for each `t` in `ts`
    """

    return last_return_index + 1


def t_return_periods(
    ts, return_periods, number_of_returns
):
    """
    Determine the return periods for a finite simulation time from a single run

    Generator that for every finite simulation time `t` yields a sequence of
    return periods up to the finite simulation time `t`, excluding the last
    (truncated) return period

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    return_periods: 1-D ndarray
        a sequence of return periods, output of
        :py:func:`epochs2periods`

    number_of_returns: 1-D ndarray
        the number of returns in a single run, output of
        :py:func:`number_of_returns`

    Yields
    ------
    1-D ndarray
        The sequence of return periods up to the finite simulation time `t`,
        excluding the last (truncated) return period
    """

    for t_index, t in enumerate(ts):
        yield return_periods[:number_of_returns[t_index]]


def return_period_histogram(
    ts,
    ts_return_periods,
    numbins=10,
    logbins=False,
    cumfreq=False,
    min_period=1.0,
    max_period=None,
):
    """
    The single-run histogram of return periods

    The return period distribution only includes finite return periods.
    That is, the last (truncated) return periods are excluded.

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    ts_return_periods : iterable
        output of :py:func:`t_return_periods`

    numbins : int, optional
        The number of bins to use for the histogram. Default is 10.

    logbins : boolean, optional
        If `True`, bin edges are on a logarithmic scale. Default is `True`.

    cumfreq : boolean, optional
        if `True`, cumulative frequencies are returned. Default is `True`.

    min_period : int or float, optional
        Lower value of the range of the histogram. Default is `1.0`.

    max_period : int or float, optional
        Upper value of the range of the histogram. Default is `None`.
        If `None`, the upper value of the range is `np.sqrt(ts.max())`.

    Returns
    -------
    hists : 2-D ndarray of ints
        `hists[t_index]` is the histogram for the `t_index`-th finite
        simulation time from `ts`.

    bin_edges : 1-D ndarray of floats
        The bin edges
    """

    if not ts.size:
        raise ValueError

    if max_period is None:
        max_period = np.sqrt(ts.max())

    if min_period >= max_period:
        raise ValueError

    hist_range = (
        min_period, max_period
    )
    if logbins:
        hist_range = np.log(hist_range)

    ret_dtype = [
        (np.str(field[0]), ) + field[1:]
        for field in [
            ('histogram_counts', np.int, (numbins, )),
            ('histogram_edges', np.float, (numbins + 1, )),
        ]
    ]
    ret = np.empty(ts.size, dtype=ret_dtype)

    # get return periods for finite time t
    for t_index, t_return_periods in enumerate(ts_return_periods):
        my_t_return_periods = (
            np.log(t_return_periods) if logbins else t_return_periods
        )
        ret[t_index]['histogram_counts'], ret[t_index]['histogram_edges'] = (
            np.histogram(
                my_t_return_periods,
                bins=numbins,
                range=hist_range,
            )
        )
        if cumfreq:
            ret[t_index]['histogram_counts'] = (
                ret[t_index]['histogram_counts'].cumsum()
            )
        if logbins:
            ret[t_index]['histogram_edges'] = (
                np.exp(ret[t_index]['histogram_edges'])
            )

    return ret


def has_spanning_cluster(
    ts,
    last_return_period,
    threshold_fraction=SPANNING_CLUSTER_THRESHOLD_FRACTION,
):
    """
    Determine whether a sequence of return epochs has a spanning cluster

    In percolation theory, a *spanning cluster* is a cluster extending to the
    system size. The temporal percolation setting is a one-ended 1-d infinite
    chain. A "spanning cluster" is the (possibly infinite) last return period,
    if it extends beyond a threshold fraction of the whole (finite) simulation
    time.

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    last_return_period: 1-D ndarray
        the output of :py:func:`last_return_period`

    threshold_fraction: float, optional
        a fraction between 0 and 1 that the last return period needs to extend
        to of the finite simulation time `t` in order to qualify as a spanning
        cluster

    Returns
    -------
    1-D ndarray of bools
        A boolean array, which indicates whether the sequence of return epochs
        exhibits a spanning cluster or not, for each `t` in `ts`
    """

    return last_return_period >= ts * threshold_fraction


def max_finite_return_period(ts_return_periods):
    """
    Determine the maximum finite return period in a sequence of return periods

    This excludes the last return period, which is truncated at the finite
    simulation time.

    Parameters
    ----------
    ts_return_periods : iterable
        output of :py:func:`t_return_periods`

    Returns
    -------
    1-D ndarray
        maximum return periods with the same dtype as `ts_return_period[0]`
    """

    return np.fromiter(
        (
            t_return_periods.max() if t_return_periods.size else 0
            for t_return_periods in ts_return_periods
        ),
        dtype=ts_return_periods[0].dtype,
    )


def max_return_period(last_return_period, max_finite_return_period):
    """
    Determine the maximum return period in a sequence of return epochs

    This includes the last return period, which is truncated at the finite
    simulation time.

    Parameters
    ----------
    last_return_period: 1-D ndarray
        the output of :py:func:`last_return_period`

    max_finite_return_period : 1-D ndarray
        output of :py:func:`max_finite_return_period`

    Returns
    -------
    1-D ndarray
        maximum return periods with the same dtype as
        `max_finite_return_periods`
    """

    return np.maximum(last_return_period, max_finite_return_period)
```

```
def moments(ts, ts_return_periods):
    """
    The first 4 empirical moments of the single-run return period distribution

    Precalculates the empirical moments up to the 4th moment (for Binder
    cumulant) for each `t` in `ts`

    Parameters
    ----------
    ts : 1-D ndarray
        finite simulation times

    ts_return_periods : iterable
        output of :py:func:`t_return_periods`

    Returns
    -------
    2-D ndarray
        The first four empirical moments. The array has shape
        ``(len(ts_return_periods), 4)`` and type ``np.float64``.
    """

    ks = np.arange(1, 5)
    ret = np.zeros((ts.size, ks.size), dtype=np.float64)
    for t_index, t_return_periods in enumerate(ts_return_periods):
        ret[t_index] = np.power(
            np.tile(t_return_periods, (ks.size, 1)).T.astype(np.float64),
            ks
        ).sum(axis=0)

    return ret


def statistics(ts, return_periods, hist=False, hist_options=None):
    """
    Compute the statistics of a single run from the return periods

    Parameters
    ----------
    ts : 1-D array_like
        finite simulation times

    return_periods: 1-D array_like
        a sequence of return periods

    hist : bool, optional
        Set to ``True`` to compute the return period distribution

    hist_options : dict, optional
        Keyword parameters to pass through to
        :py:func:`return_period_histogram`

    Returns
    -------
    ret : dict
        Result dictionary with ``ret.keys() == ['last_return_period',
        'has_spanning_cluster', 'max_finite_return_period',
        'max_return_period', 'number_of_returns', 'moments']``.
        If the `hist` option was set to ``True``, there is another key
        ``return_period_histogram``.
        The values are the outputs of the respective functions.

    See also
    --------
    last_return_period

    has_spanning_cluster

    max_finite_return_period

    max_return_period

    number_of_returns

    moments

    return_period_histogram

    """

    # convert finite times to ndarray and validate
    ts = check_ts(ts)

    # convert return periods to ndarray and validate
    return_periods = check_return_periods(return_periods)

    # auxiliary statistics
    return_epochs = periods2epochs(
        return_periods=return_periods,
    )
    my_last_return_index = last_return_epoch_index(
        ts=ts, return_epochs=return_epochs,
    )
    my_number_of_returns = number_of_returns(
        last_return_index=my_last_return_index,
    )
    my_ts_return_periods = list(
        t_return_periods(
            ts=ts,
            return_periods=return_periods,
            number_of_returns=my_number_of_returns,
        )
    )
    my_last_return_epoch = last_return_epoch(
        ts=ts,
        return_epochs=return_epochs,
        last_return_index=my_last_return_index,
    )
    my_last_return_period = last_return_period(
        ts=ts,
        last_return_epoch=my_last_return_epoch,
    )
    my_max_finite_return_period = max_finite_return_period(
        ts_return_periods=my_ts_return_periods,
    )

    ret_dtype = RUN_DTYPE

    if hist:
        if hist_options is None:
            hist_options = dict()

        my_histogram = return_period_histogram(
            ts=ts, ts_return_periods=my_ts_return_periods,
            **hist_options
        )

        ret_dtype = list(ret_dtype)
        ret_dtype += my_histogram.dtype.descr

    ret = np.empty(ts.size, dtype=ret_dtype)

    ret['last_return_period'] = my_last_return_period
    ret['has_spanning_cluster'] = has_spanning_cluster(
        ts=ts,
        last_return_period=my_last_return_period,
    )
    ret['max_finite_return_period'] = my_max_finite_return_period
    ret['max_return_period'] = max_return_period(
        last_return_period=my_last_return_period,
        max_finite_return_period=my_max_finite_return_period,
    )
    ret['number_of_returns'] = my_number_of_returns
    ret['moments'] = moments(
        ts=ts,
        ts_return_periods=my_ts_return_periods,
    )

    if hist:
        for field in my_histogram.dtype.names:
            ret[field] = my_histogram[field]

    return ret
```

Listing C.9: pytemper/stats.py

```
# coding: utf-8

"""
Low-level routines for statistics over multiple runs


See Also
--------

temper : The high-level module

"""


# Python 2/3 compatibility
from __future__ import absolute_import, division, unicode_literals

from builtins import dict

import itertools

import numpy as np
```

```python
import scipy.stats
import scikits.bootstrap as boot
from numpy import ma

ALPHA_1SIGMA = 2 * scipy.stats.norm.cdf(-1.0)

STATS_RETURN_DTYPE = [
    (np.str(field[0]), ) + field[1:]
    for field in [
        ('mean', np.float),
        ('ci', np.float, 2),
    ]
]

STATISTICS_RETURN_DTYPE = [
    (np.str(field[0]), ) + field[1:]
    for field in [
        ('percolation_probability_mean', np.float),
        ('percolation_probability_ci', np.float, 2),
        ('percolation_strength_mean', np.float),
        ('percolation_strength_ci', np.float, 2),
        ('infinite_period_mean', np.float),
        ('infinite_period_ci', np.float, 2),
        ('max_finite_period_mean', np.float),
        ('max_finite_period_ci', np.float, 2),
        ('return_number_mean', np.float),
        ('return_number_ci', np.float, 2),
        ('no_return_prob_ubound_mean', np.float),
        ('no_return_prob_ubound_ci', np.float, 2),
        ('moments_mean', np.float, (4, )),
        ('moments_ci', np.float, (4, 2)),
        ('combined_moments_mean', np.float, (4, )),
        ('combined_moments_ci', np.float, (4, 2)),
    ]
]


def _normal_ci(quantity, normalize_by=None, alpha=ALPHA_1SIGMA):
    """
    Compute the mean and its normal confidence interval

    Parameters
    ----------
    quantity: 2-D ndarray
        from a field of the output of :py:func:`merge_runs`

    normalize_by
        division factors for normalization of the quantity

    alpha: float, optional
        Significance level. ``1 - alpha`` is the confidence level

    Returns
    -------
    ret[0]: 1-D ndarray of float
        The estimated mean for each `t` in `ts`

    ret[1]: 2-D ndarray of float
        The lower and upper bounds of the confidence interval for each `t` in
        `ts`
    See also
    --------
    merge_runs

    single_run_statistics
    """

    ret = np.empty(quantity.shape[1:], dtype=STATS_RETURN_DTYPE)

    ret['mean'] = quantity.mean(axis=0)
    if normalize_by is not None:
        ret['mean'] /= normalize_by

    n = quantity.shape[0]
    df = n - 1
    s = quantity.std(axis=0, ddof=1)
    if normalize_by is not None:
        s /= normalize_by

    ci = np.array(scipy.stats.t.interval(
        1. - alpha,
        df=df,
        loc=ret['mean'],
        scale=s / np.sqrt(n)
    ))

    ret['ci'] = ci.T if quantity.ndim == 2 else np.rollaxis(
        ci, axis=0, start=3
    )

    return ret


def percolation_probability(has_spanning_cluster, alpha=ALPHA_1SIGMA):
    r'''
    Compute the percolation probability and its confidence interval

    The percolation probability is the Binomial proportion of spanning
    clusters.
    Employs the Bayesian credible interval as confidence interval.

    Parameters
    ----------
    has_spanning_cluster: 2-D ndarray of bool
        from the ``has_spanning_cluster`` field of the output of
        :py:func:`merge_runs`

    alpha: float, optional
        Significance level. ``1 - alpha`` is the confidence level

    Returns
    -------
    ret[0] : 1-D ndarray of float
        The estimated percolation probability for each `t` in `ts`

    ret[1] : 2-D ndarray of float
        The lower and upper bounds of the confidence interval for each `t` in
        `ts`
    See also
    --------
    merge_runs

    single_run_statistics

    pytemper.singlerun.has_spanning_cluster

    Notes
    -----
    Given :math:`k` runs with spanning clusters, out of a total of :math:`n`
    runs, we adopt the Bayesian posterior mean :math:`\bar{p} =
    \frac{k+1}{n+2}` as point estimator for the percolation probability.
    See the :ref:`ci-proportions` Section in the manual.

    The :math:`1-\alpha` credible interval is given by the
    :math:`\frac{\alpha}{2}, 1-\frac{\alpha}{2}` quantiles of the Beta
    distribution :math:`B(k+1, n-k+1)`.
    See the :ref:`ci-proportions` Section in the manual.

    References
    ----------
    .. [1] E. Cameron, Publications of the Astronomical Society of Australia
       28, 128 (2011), `doi:10.1071/as10046
       <http://dx.doi.org/10.1071/as10046>`_
    .. [2] L. Wasserman, All of Statistics (Springer New York, 2004),
       `doi:10.1007/978-0-387-21736-9
       <http://dx.doi.org/10.1007/978-0-387-21736-9>`_
    '''

    number_of_ts = has_spanning_cluster.shape[1]
    ret = np.empty(number_of_ts, dtype=STATS_RETURN_DTYPE)

    n = float(has_spanning_cluster.shape[0])
    k = has_spanning_cluster.sum(axis=0, dtype=np.float)

    a = k + 1.
    b = n - k + 1.

    p_lower = scipy.stats.distributions.beta.ppf(alpha / 2., a, b)
    p_upper = scipy.stats.distributions.beta.ppf(1. - alpha / 2., a, b)

    ret['mean'] = (k + 1.) / (n + 2.)
    ret['ci'] = np.vstack((p_lower, p_upper)).T

    return ret


def percolation_strength(ts, max_return_period, alpha=ALPHA_1SIGMA):
    """
    Compute the percolation strength and its confidence interval

    The percolation strength is the average relative size of the maximum return
    period.
    Note this does not need to coincide with a spanning cluster.
    Employs the normal confidence interval.

    Parameters
    ----------
    ts : 1-D array_like
```

```
            finite simulation times

        max_return_period: 2-D array_like
            from the ``max_return_period`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0]: 1-D ndarray of float
            The estimated percolation strength for each `t` in `ts`

        ret[1]: 2-D ndarray of float
            The lower and upper bounds of the confidence interval for each `t` in
            `ts`

        See also
        --------
        merge_runs

        single_run_statistics

        pytemper.singlerun.max_return_period

        """

        return _normal_ci(
            quantity=max_return_period,
            normalize_by=ts,
            alpha=alpha,
        )


    def infinite_period(ts, last_return_period, alpha=ALPHA_1SIGMA):
        """
        Compute the infinite return period and its confidence interval

        The infinite period is the average relative size of the last return period.
        Employs the normal confidence interval.

        Parameters
        ----------
        ts : 1-D array_like
            finite simulation times

        last_return_period: 2-D array_like
            from the ``last_return_period`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0]: 1-D ndarray of float
            The estimated infinite period for each `t` in `ts`

        ret[1]: 2-D ndarray of float
```

```
            The lower and upper bounds of the confidence interval for each `t` in
            `ts`

        See also
        --------
        merge_runs

        single_run_statistics

        pytemper.singlerun.last_return_period

        """

        return _normal_ci(
            quantity=last_return_period,
            normalize_by=ts,
            alpha=alpha,
        )


    def max_finite_period(ts, max_finite_return_period, alpha=ALPHA_1SIGMA):
        """
        Compute the maximum finite period and its confidence interval

        The maximum finite period is the average relative size of the maximum
        finite return period (excluding the last return period).
        Employs the normal confidence interval.

        Parameters
        ----------
        ts : 1-D array_like
            finite simulation times

        max_finite_return_period: 2-D array_like
            from the ``max_finite_return_period`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0]: 1-D ndarray of float
            The estimated maximum finite period for each `t` in `ts`

        ret[1]: 2-D ndarray of float
            The lower and upper bounds of the confidence interval for each `t` in
            `ts`

        See also
        --------
        merge_runs

        single_run_statistics

        pytemper.singlerun.max_finite_return_period

        """

        return _normal_ci(
            quantity=max_finite_return_period,
            normalize_by=ts,
            alpha=alpha,
        )


    def return_number(ts, number_of_returns, alpha=ALPHA_1SIGMA):
        """
        Compute the normalized number of returns and its confidence interval

        This is the average relative number of returns per time unit.
        Employs the normal confidence interval.

        Parameters
        ----------
        ts : 1-D array_like
            finite simulation times

        number_of_returns: 2-D array_like
            from the ``number_of_returns`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0]: 1-D ndarray of float
            The estimated return number for each `t` in `ts`

        ret[1]: 2-D ndarray of float
            The lower and upper bounds of the confidence interval for each `t` in
            `ts`

        See also
        --------
        merge_runs

        single_run_statistics

        pytemper.singlerun.number_of_returns

        """

        return _normal_ci(
            quantity=number_of_returns,
            normalize_by=ts,
            alpha=alpha
        )


    def no_return_prob_ubound(number_of_returns, alpha=ALPHA_1SIGMA):
        r'''
        Compute the upper bound of the no-return probability and its confidence
        interval

        The upper bound of the probability of no return is the
        maximum-likelihood-estimate of the parameter of a geometric distribution.
        This is an upper bound as this assumes that every run terminates with a
        fail to return, even though the return probability might well be 1, and
        hence, the "no-return" parameter be 0.

        Use Bayesian inference with a Beta(0,0) conjugate prior for the confidence
        interval.

        Parameters
        ----------
        number_of_returns : 2-D array_like
            from the ``number_of_returns`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0] : 1-D ndarray of float
            The estimated upper bound of the no-return probability for each `t` in
            `ts`

        ret[1] : 2-D ndarray of float
            The lower and upper bounds of the confidence interval for each `t` in
            `ts`

        See also
        --------
        merge_runs

        single_run_statistics

        number_of_returns

        Notes
        -----

        Determine the parameter as the maximum likelihood estimate

        .. math::

           \hat{p} = \frac{n}{\sum_{i=1}^n k_i + n}.

        This is also the posterior mean for the conjugate prior Beta(0,0)
        distribution in Bayesian inference.

        Determine the Bayesian credible interval according to the Beta(a,b)
        posterior with

        .. math::
           a = n, b = \sum_{i=1}^n k_i.

        References
        ----------
        .. [1] http://en.wikipedia.org/wiki/Geometric_distribution#
            Parameter_estimation

        .. [2] http://en.wikipedia.org/wiki/Beta_distribution#Bayesian_inference

        '''

        number_of_ts = number_of_returns.shape[1]
        ret = np.empty(number_of_ts, dtype=STATS_RETURN_DTYPE)

        a = float(number_of_returns.shape[0])
        b = number_of_returns.sum(axis=0)

        ret['mean'] = a / (a + b)
        ret['ci'] = np.array(scipy.stats.beta.interval(
            1. - alpha, a=a, b=b,
        )).T

        return ret


    def moments(number_of_returns, moments, alpha=ALPHA_1SIGMA):
        """
        Empirical moments of the return period distribution and their confidence
        intervals

        The computation of the moments excludes the respective (truncated, and
        possibly "spanning") last return period.
        Employs the normal confidence interval.

        Parameters
        ----------
        number_of_returns : 2-D array_like
            from the ``number_of_returns`` field of the output of
            :py:func:`merge_runs`

        moments : 3-D array_like
            from the ``moments`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0] : 2-D ndarray of float
            The average moment for each `t` in `ts`. ``ret[0][k - 1][i]`` is the
            empirical k-th raw moments of the i-th simulation time. The shape of
            the array is ``(4, ts.size)``.

        ret[1] : 3-D ndarray of float
            ``ret[1][k - 1]`` are the lower and upper bounds of the confidence
            interval for the k-th empirical raw moment for each `t` in `ts`.
            The array `ret[1]` has shape ``(4, 2, ts.size)``.

        See also
        --------
        merge_single

        single_run_statistics

        singlerun.moments
        """

        ret = np.empty(moments.shape[1:], dtype=STATS_RETURN_DTYPE)

        # moments: (omega, t, moment)
        # number_of_returns: (omega, t)
        # normalized_moments: (omega, t, moment)
        normalized_moments = np.rollaxis(
            ma.masked_invalid(
                np.rollaxis(moments, axis=2) / number_of_returns,
            ).filled(0.0),
            axis=0,
            start=3,
        )

        ret['mean'] = np.mean(normalized_moments, axis=0)

        n = moments.shape[0]
        df = n - 1
        dtype = moments.dtype.type

        # s: (moment, t)
        s = normalized_moments.astype(np.float64).std(
            axis=0, ddof=1
        ).astype(dtype)

        ret['ci'] = np.rollaxis(
            np.array(scipy.stats.t.interval(
                1. - alpha,
                df=df,
                loc=ret['mean'],
                scale=s / np.sqrt(n)
            )).T,
            1
        )

        return ret


    def combined_moments(number_of_returns, moments, alpha=ALPHA_1SIGMA):
        """
        Empirical moments of the return period distribution and their confidence
        intervals, computed by summing over all runs first and then averaging

        The computation of the moments excludes the respective (truncated, and
        possibly "spanning") last return period.
        Employs a poor man's confidence interval based on the ranks of the given
        sample of moments.

        TODO
        Employ bootstrapping

        Parameters
        ----------
        number_of_returns : 2-D array_like
            from the ``number_of_returns`` field of the output of
            :py:func:`merge_runs`

        moments : 3-D array_like
            from the ``moments`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret[0] : 2-D ndarray of float
            The average moment for each `t` in `ts`. ``ret[0][k - 1][i]`` is the
            empirical k-th raw moments of the i-th simulation time. The shape of
            the array is ``(4, ts.size)``.

        ret[1] : 3-D ndarray of float
            ``ret[1][k - 1]`` are the lower and upper bounds of the confidence
            interval for the k-th empirical raw moment for each `t` in `ts`.
            The array `ret[1]` has shape ``(4, 2, ts.size)``.

        See also
        --------
        moments

        single_run_statistics

        singlerun.moments
        """

        ret = np.empty(moments.shape[1:], dtype=STATS_RETURN_DTYPE)

        # moments: (omega, t, moment)
        # number_of_returns: (omega, t)
        # ret['mean']: (t, moment)
        ret['mean'] = ma.masked_invalid(
            moments.sum(axis=0).T / number_of_returns.sum(axis=0)
        ).filled(0.0).T

        def combinedmean(mymoments, mynumber_of_returns, weights):
            sum_returns = np.sum(np.multiply(mynumber_of_returns, weights))
            return (
                np.sum(np.multiply(mymoments, weights)) / sum_returns if sum_returns
                else 0.0
            )

        # ret['ci']: (t, moment, 2)
        n_ts = moments.shape[1]
        n_moments = moments.shape[2]
        ret['ci'] = np.zeros((n_ts, n_moments, 2))
        for t_index, moment_index in itertools.product(range(n_ts), range(n_moments)
         ):
            ret['ci'][t_index, moment_index] = boot.ci(
                data=(
                    moments[:, t_index, moment_index],
                    number_of_returns[:, t_index],
                ),
                statfunction=combinedmean,
                method='abc',
                multi=True,
                alpha=alpha,
            )

        return ret


    def histogram_counts(counts, alpha=ALPHA_1SIGMA):
        """
        The per-bin averaged empirical return period histograms and their per-bin
        confidence intervals
        The respective last return period is excluded.
        Employs the normal confidence interval.

        Parameters
        ----------
        histograms : tuple of bin counts and bin edges
            from the ``return_period_histogram`` field of the output of
            :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level

        Returns
        -------
        ret : singlerun.Histograms namedtuple

        ret.counts.mean : 2-D ndarray of float
            The average bin count for each `t` in `ts`. ``ret[0].mean[i][j]`` is
            the average bin count of the i-th simulation time in the j-th bin.

        ret.counts.ci : 3-D ndarray of float
            ``ret[0].ci[:, i, j]`` are the lower and upper bounds of the confidence
            interval for the i-th simulation time in the j-th bin.

        ret.edges : 1-D ndarray of floats
            The bin edges

        See also
        --------
        merge_runs

        single_run_statistics

        singlerun.return_period_histogram
        """

        return _normal_ci(
            quantity=counts, alpha=alpha,
        )


    def statistics(ts, merged_runs, alpha=ALPHA_1SIGMA):
        """
        Aggregate multiple runs and collect statistics

        Includes the return period distribution if it is present in the merged
        single-run statistics

        Parameters
        ----------
        ts : 1-D array_like
            finite simulation times

        merged_runs : dict
            output of :py:func:`merge_runs`

        alpha: float, optional
            Significance level. ``1 - alpha`` is the confidence level
        Returns
        -------
        ret : dict
            Result dictionary with ``ret.keys() == ['percolation_probability',
            percolation_strength', 'infinite_period', 'max_finite_period',
            'return_number', 'no_return_prob_ubound', 'moments']``.
            If `merged_runs` contains a key
            ``return_period_histogram``, there is the key ``histograms``.
            The values are the outputs of the corresponding function.

        See also
        --------
        merge_runs

        percolation_probability

        percolation_strength

        infinite_period

        max_finite_period

        return_number

        no_return_prob_ubound

        moments

        combined_moments

        histograms
        """

        has_histograms = 'histogram_counts' in merged_runs.dtype.names
        ret_dtype = list(STATISTICS_RETURN_DTYPE)
        if has_histograms:
            numbins = merged_runs.dtype['histogram_counts'].shape[0]
            ret_dtype += [
                (np.str(field[0]), ) + field[1:]
                for field in [
                    ('histogram_counts_mean', np.float, (numbins, )),
                    ('histogram_counts_ci', np.float, (numbins, 2)),
                    ('histogram_edges', np.float, numbins + 1)]
            ]

        ret = np.empty(merged_runs.shape[1], dtype=ret_dtype)

        res = percolation_probability(
            has_spanning_cluster=merged_runs['has_spanning_cluster'],
            alpha=alpha,
        )
        ret['percolation_probability_mean'] = res['mean']
        ret['percolation_probability_ci'] = res['ci']

        res = percolation_strength(
            ts=ts,
            max_return_period=merged_runs['max_return_period'],
            alpha=alpha,
        )
        ret['percolation_strength_mean'] = res['mean']
        ret['percolation_strength_ci'] = res['ci']

        res = infinite_period(
            ts=ts,
            last_return_period=merged_runs['last_return_period'],
            alpha=alpha,
        )
        ret['infinite_period_mean'] = res['mean']
        ret['infinite_period_ci'] = res['ci']

        res = max_finite_period(
            ts=ts,
            max_finite_return_period=merged_runs['max_finite_return_period'],
            alpha=alpha,
        )
        ret['max_finite_period_mean'] = res['mean']
        ret['max_finite_period_ci'] = res['ci']

        res = return_number(
            ts=ts,
            number_of_returns=merged_runs['number_of_returns'],
            alpha=alpha,
        )
        ret['return_number_mean'] = res['mean']
        ret['return_number_ci'] = res['ci']

        res = no_return_prob_ubound(
            number_of_returns=merged_runs['number_of_returns'],
            alpha=alpha,
        )
        ret['no_return_prob_ubound_mean'] = res['mean']
        ret['no_return_prob_ubound_ci'] = res['ci']

        res = moments(
            number_of_returns=merged_runs['number_of_returns'],
            moments=merged_runs['moments'],
            alpha=alpha,
        )
        ret['moments_mean'] = res['mean']
        ret['moments_ci'] = res['ci']

        res = combined_moments(
            number_of_returns=merged_runs['number_of_returns'],
            moments=merged_runs['moments'],
            alpha=alpha,
        )
        ret['combined_moments_mean'] = res['mean']
        ret['combined_moments_ci'] = res['ci']

        if has_histograms:
            res = histogram_counts(
                counts=merged_runs['histogram_counts'],
                alpha=alpha)
            ret['histogram_counts_mean'] = res['mean']
            ret['histogram_counts_ci'] = res['ci']
            ret['histogram_edges'] = merged_runs['histogram_edges'][0]

        return ret
```

Listing C.10: pytemper/examples.py

```python
# coding: utf-8  # pylint: disable=invalid-name

"""
Examples for analyzing the recurrence--transience transition
"""

import itertools

import numpy as np


def mm1q_trajectories(seed, n_runs, n_steps, rho):
    '''
    Simulate n_runs M/M/1 queue sample paths with n_steps at parameters rho.
    Initialize the seeds with the master seed parameter.
    '''
    rho = np.asanyarray(rho)
    if not np.all(rho > 0.0):
        raise ValueError

    rng = np.random.RandomState(seed=seed)
    trajectories = np.zeros((n_steps, n_runs, rho.size))

    interarrival_time = np.zeros((n_steps, n_runs, rho.size))
    interarrival_time[1:] = np.rollaxis(
        np.tile(
            rng.exponential(size=(n_steps - 1, n_runs)),
            (rho.size, 1, 1)),
        axis=0, start=3) / rho

    service_time = np.empty((n_steps, n_runs, rho.size))
    service_time[:] = np.rollaxis(
        np.tile(
            rng.exponential(size=(n_steps, n_runs)),
            (rho.size, 1, 1)),
        axis=0, start=3)

    residual_work = np.zeros((n_steps, n_runs, rho.size))
    for n in range(1, n_steps):
        residual_work[n] = np.maximum(
            residual_work[n - 1] + service_time[n - 1] - interarrival_time[n],
            0.0)

    arrival_time = interarrival_time.cumsum(axis=0)
    departure_time = arrival_time + residual_work + service_time

    arrivals = np.rollaxis(
        np.tile(np.arange(n_steps), (n_runs, rho.size, 1)), axis=2)
    departures = np.zeros((n_steps, n_runs, rho.size))
    for i, j in itertools.product(range(n_runs), range(rho.size)):
        departures[:, i, j] = np.searchsorted(
            departure_time[:, i, j], arrival_time[:, i, j])

    trajectories[:] = arrivals - departures
    return trajectories

```

```python
def random_walk_trajectories(seed, n_runs, n_steps, p):
    '''
    Simulate n_runs one-sided random walks with n_steps at parameters p.
    Initialize the seeds with the master seed parameter.
    '''
    p = np.asanyarray(p)
    rng = np.random.RandomState(seed=seed)
    random_numbers = rng.rand(n_steps, n_runs)
    trajectories = np.zeros((n_steps + 1, n_runs, p.size))
    for n in range(n_steps):
        p_mesh, random_numbers_mesh = np.meshgrid(p, random_numbers[n])
        trajectories[n + 1] = trajectories[n] + np.where(trajectories[n], 2 * (
            random_numbers_mesh < p_mesh) - 1, random_numbers_mesh < p_mesh)
    return trajectories


def random_walk_cdf(p, t_max):
    """
    Calculate the cumulative distribution function of the return periods
    of a random walk

    p -- a numpy array of parameters p
    t_max -- maximum return period
    """

    p = np.asanyarray(p, dtype=np.longdouble)
    x = p * (1. - p)

    fn = np.zeros(
        (p.size, t_max),
        dtype=np.longdouble
    )

    # $f_0^p(1)$
    fn[:, 0] = 1. - p

    if t_max == 1:
        return np.squeeze(fn)

    # $f_0^p(2)$, $k = 1$
    fn[:, 1] = x

    for n in range(2, t_max - 1, 2):
        # $f_0^p(n+2) = f_0^p(n) * x * 4 * (n - 1) / (n + 2)$
        fn[:, n + 1] = fn[:, n - 1] * x * float(4 * (n - 1) / (n + 2))

    return np.squeeze(fn.cumsum(axis=1))
```

C.5 FSSA PYTHON PACKAGE

fssa version: 0.7.6

Listing C.11: fssa/__init__.py, available online at `https://github.com/andsor/pyfssa/blob/0.7.6/fssa/__init__.py`

```python
#!/usr/bin/env python
```

```
# encoding: utf-8

r"""
Implements algorithmic finite-size scaling analysis at phase transitions

This module implements the algorithmic finite-size scaling analysis at phase
transitions as demonstrated by Oliver Melchert and his superb autoscale.py
script.

The :mod:`fssa` module provides these high-level functions from the
:mod:`fssa.fssa` module:

.. autosummary::

   fssa.scaledata
   fssa.quality
   fssa.autoscale

See Also
--------

fssa.fssa : low-level functions

Notes
-----

The :func:`fssa.scaledata` function scales finite-size data in order for the
data to hopefully collapse onto a single universal scaling function, also
known as master curve.
The :func:`fssa.quality` function assesses the quality of this very data
collapse onto a single curve.
Finally, the :func:`fssa.autoscale` function frames the data collapse as an
optimization problem and searches for the critical values that minimize the
quality function.

The **fssa** package expects finite-size data in the following setting.

.. math::

   A_L(\varrho) = L^{\zeta/\nu} \tilde{f}\left(L^{1/\nu} (\varrho -
   \varrho_c)\right), \qquad (L \to \infty, \varrho \to \varrho_c),

`l` is like a 1-D numpy array which contains the finite system sizes :math:`L`.
`rho` is like a 1-D numpy array which contains the parameter values
:math:`\varrho`.
`a` is like a 2-D numpy array which contains the observations (the data)
:math:`A_L(\varrho)`, where `a[i, j]` is the data at the `i`-th system size and
the `j`-th parameter value.
`da` is like a 2-D numpy array which contains the standard errors in the
observations.
This implementation uses the quality function by Houdayer & Hartmann [1]_
which measures the quality of the data collapse, see the sections
:ref:`data-collapse-method` and :ref:`quality-function` in the manual.

This function and the whole fssa package have been inspired by Oliver
Melchert and his superb **autoScale** package [2]_.

The critical point and exponents, including its standard errors and
(co)variances, are fitted by the Nelder--Mead algorithm, see the section
:ref:`neldermead` in the manual.
```

```
Currently, the module only implements homogeneous data arrays:
Data must be available for all finite system sizes and parameter values.

References
----------
.. [1] J. Houdayer and A. Hartmann, Physical Review B 70, 014418+ (2004)
    `doi:10.1103/physrevb.70.014418
    <http://dx.doi.org/doi:10.1103/physrevb.70.014418>`_

.. [2] O. Melchert, `arXiv:0910.5403 <http://arxiv.org/abs/0910.5403>`_
    (2009)

.. todo::

   `Implement heterogeneous finite-size data handling`__

__ https://github.com/andsor/pyfssa/issues/2
"""
from __future__ import absolute_import

import pkg_resources
from .fssa import scaledata, quality, autoscale

__version__ = pkg_resources.get_distribution(__name__).version
```

Listing C.12: fssa/fssa.py, available online at https://github.com/andsor/pyfssa/blob/0.7.6/fssa/fssa.py

```
#!/usr/bin/env python
# encoding: utf-8

r"""
Low-level routines for finite-size scaling analysis

See Also
--------

fssa : The high-level module

Notes
-----

The **fssa** package provides routines to perform finite-size scaling analyses
on experimental data [10]_ [11]_.

It has been inspired by Oliver Melchert and his superb **autoScale** package
[3]_.

References
----------

.. [10] M. E. J. Newman and G. T. Barkema, Monte Carlo Methods in Statistical
   Physics (Oxford University Press, 1999)

.. [11] K. Binder and D. W. Heermann, `Monte Carlo Simulation in Statistical
   Physics <http://dx.doi.org/10.1007/978-3-642-03163-2>`_ (Springer, Berlin,
   Heidelberg, 2010)
```

```
.. [3] O. Melchert, `arXiv:0910.5403 <http://arxiv.org/abs/0910.5403>`_
   (2009)

"""

# Python 2/3 compatibility
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

import warnings
from builtins import *
from collections import namedtuple

import numpy as np
import numpy.ma as ma
import scipy.optimize

from .optimize import _minimize_neldermead


class ScaledData(namedtuple('ScaledData', ['x', 'y', 'dy'])):
    """
    A :py:func:`namedtuple <collections.namedtuple>` for :py:func:`scaledata`
    output
    """

    # set this to keep memory requirements low, according to
    # http://docs.python.org/3/library/collections.html#namedtuple-factory-
        function-for-tuples-with-named-fields
    __slots__ = ()


def scaledata(l, rho, a, da, rho_c, nu, zeta):
    r'''
    Scale experimental data according to critical exponents

    Parameters
    ----------
    l, rho : 1-D array_like
       finite system sizes `l` and parameter values `rho`

    a, da : 2-D array_like of shape (`l`.size, `rho`.size)
       experimental data `a` with standard errors `da` obtained at finite
       system sizes `l` and parameter values `rho`, with
       ``a.shape == da.shape == (l.size, rho.size)``

    rho_c : float in range [rho.min(), rho.max()]
       (assumed) critical parameter value with ``rho_c >= rho.min() and rho_c
       <= rho.max()``

    nu, zeta : float
       (assumed) critical exponents

    Returns
    -------
    :py:class:`ScaledData`
       scaled data `x`, `y` with standard errors `dy`

    x, y, dy : ndarray
```

```
        two-dimensional arrays of shape ``(l.size, rho.size)``

    Notes
    -----
    Scale data points :math:`(\varrho_j, a_{ij}, da_{ij})` observed at finite
    system sizes :math:`L_i` and parameter values :math:`\varrho_i` according
    to the finite-size scaling ansatz

    .. math::

       L^{-\zeta/\nu} a_{ij} = \tilde{f}\left( L^{1/\nu} (\varrho_j -
       \varrho_c) \right).

    The output is the scaled data points :math:`(x_{ij}, y_{ij}, dy_{ij})` with

    .. math::

       x_{ij} & = L_i^{1/\nu} (\varrho_j - \varrho_c) \\
       y_{ij} & = L_i^{-\zeta/\nu} a_{ij} \\
       dy_{ij} & = L_i^{-\zeta/\nu} da_{ij}

    such that all data points :ref:`collapse <data-collapse-method>` onto the
    single curve :math:`\tilde{f}(x)` with the right choice of
    :math:`\varrho_c, \nu, \zeta` [4]_ [5]_.

    Raises
    ------
    ValueError
       If `l` or `rho` is not 1-D array_like, if `a` or `da` is not 2-D
       array_like, if the shape of `a` or `da` differs from ``(l.size,
       rho.size)``

    References
    ----------

    .. [4] M. E. J. Newman and G. T. Barkema, Monte Carlo Methods in
       Statistical Physics (Oxford University Press, 1999)

    .. [5] K. Binder and D. W. Heermann, `Monte Carlo Simulation in Statistical
       Physics <http://dx.doi.org/10.1007/978-3-642-03163-2>`_ (Springer,
       Berlin, Heidelberg, 2010)
    '''

    # l should be 1-D array_like
    l = np.asanyarray(l)
    if l.ndim != 1:
        raise ValueError("l should be 1-D array_like")

    # rho should be 1-D array_like
    rho = np.asanyarray(rho)
    if rho.ndim != 1:
        raise ValueError("rho should be 1-D array_like")

    # a should be 2-D array_like
    a = np.asanyarray(a)
    if a.ndim != 2:
        raise ValueError("a should be 2-D array_like")

    # a should have shape (l.size, rho.size)
    if a.shape != (l.size, rho.size):
        raise ValueError("a should have shape (l.size, rho.size)")

    # da should be 2-D array_like
    da = np.asanyarray(da)
    if da.ndim != 2:
        raise ValueError("da should be 2-D array_like")

    # da should have shape (l.size, rho.size)
    if da.shape != (l.size, rho.size):
        raise ValueError("da should have shape (l.size, rho.size)")

    # rho_c should be float
    rho_c = float(rho_c)

    # rho_c should be in range
    if rho_c > rho.max() or rho_c < rho.min():
        warnings.warn("rho_c is out of range", RuntimeWarning)

    # nu should be float
    nu = float(nu)

    # zeta should be float
    zeta = float(zeta)

    l_mesh, rho_mesh = np.meshgrid(l, rho, indexing='ij')

    x = np.power(l_mesh, 1. / nu) * (rho_mesh - rho_c)
    y = np.power(l_mesh, - zeta / nu) * a
    dy = np.power(l_mesh, - zeta / nu) * da

    return ScaledData(x, y, dy)


def _wls_linearfit_predict(x, w, wx, wy, wxx, wxy, select):
    """
    Predict a point according to a weighted least squares linear fit of the
    data

    This function is a helper function for :py:func:`quality`. It is not
    supposed to be called directly.

    Parameters
    ----------
    x : float
        The position for which to predict the function value

    w : ndarray
        The pre-calculated weights :math:`w_l`

    wx : ndarray
        The pre-calculated weighted `x` data :math:`w_l x_l`

    wy : ndarray
        The pre-calculated weighted `y` data :math:`w_l y_l`

    wxx : ndarray
    The pre-calculated weighted :math:`x^2` data :math:`w_l x_l^2`

    wxy : ndarray
        The pre-calculated weighted `x y` data :math:`w_l x_l y_l`
    select : indexing array
        To select the subset from the `w`, `wx`, `wy`, `wxx`, `wxy` data

    Returns
    -------
    float, float
        The estimated value of the master curve for the selected subset and the
        squared standard error
    """

    # linear fit
    k = w[select].sum()
    kx = wx[select].sum()
    ky = wy[select].sum()
    kxx = wxx[select].sum()
    kxy = wxy[select].sum()
    delta = k * kxx - kx ** 2
    m = 1. / delta * (k * kxy - kx * ky)
    b = 1. / delta * (kxx * ky - kx * kxy)
    b_var = kxx / delta
    m_var = k / delta
    bm_covar = - kx / delta

    # estimation
    y = b + m * x
    dy2 = b_var + 2 * bm_covar * x + m_var * x**2

    return y, dy2


def _jprimes(x, i, x_bounds=None):
    """
    Helper function to return the j' indices for the master curve fit

    This function is a helper function for :py:func:`quality`. It is not
    supposed to be called directly.

    Parameters
    ----------
    x : mapping to ndarrays
        The x values.

    i : int
        The row index (finite size index)

    x_bounds : 2-tuple, optional
        bounds on x values

    Returns
    -------
    ret : mapping to ndarrays
        Has the same keys and shape as `x`.
        Its element ``ret[i'][j]`` is the j' such that :math:`x_{i'j'} \leq
        x_{ij} < x_{i'(j'+1)}`.
        If no such j' exists, the element is np.nan.
        Convert the element to int to use as an index.
    """

    j_primes = - np.ones_like(x)
```

```python

    try:
        x_masked = ma.masked_outside(x, x_bounds[0], x_bounds[1])
    except (TypeError, IndexError):
        x_masked = ma.asanyarray(x)

    k, n = x.shape

    # indices of lower and upper bounds
    edges = ma.notmasked_edges(x_masked, axis=1)
    x_lower = np.zeros(k, dtype=int)
    x_upper = np.zeros(k, dtype=int)
    x_lower[edges[0][0]] = edges[0][-1]
    x_upper[edges[-1][0]] = edges[-1][-1]

    for i_prime in range(k):
        if i_prime == i:
            j_primes[i_prime][:] = np.nan
            continue

        jprimes = np.searchsorted(
            x[i_prime], x[i], side='right'
        ).astype(float) - 1
        jprimes[
            np.logical_or(
                jprimes < x_lower[i_prime],
                jprimes >= x_upper[i_prime]
            )
        ] = np.nan
        j_primes[i_prime][:] = jprimes

    return j_primes


def _select_mask(j, j_primes):
    """
    Return a boolean mask for selecting the data subset according to the j'

    Parameters
    ----------
    j : int
        current j index

    j_primes : ndarray
        result from _jprimes call
    """

    ret = np.zeros_like(j_primes, dtype=bool)
    my_iprimes = np.invert(np.isnan(j_primes[:, j])).nonzero()[0]
    my_jprimes = j_primes[my_iprimes, j]
    my_jprimes = my_jprimes.astype(np.int)
    ret[my_iprimes, my_jprimes] = True
    ret[my_iprimes, my_jprimes + 1] = True

    return ret


def quality(x, y, dy, x_bounds=None):
    r'''
    Quality of data collapse onto a master curve defined by the data

    This is the reduced chi-square statistic for a data fit except that the
    master curve is fitted from the data itself.

    Parameters
    ----------
    x, y, dy : 2-D array_like
        output from :py:func:`scaledata`, scaled data `x`, `y` with standard
        errors `dy`

    x_bounds : tuple of floats, optional
        lower and upper bound for scaled data `x` to consider

    Returns
    -------
    float
        the quality of the data collapse

    Raises
    ------
    ValueError
        if not all arrays `x`, `y`, `dy` have dimension 2, or if not all arrays
        are of the same shape, or if `x` is not sorted along rows (``axis=1``),
        or if `dy` does not have only positive entries

    Notes
    -----
    This is the implementation of the reduced :math:`\chi^2` quality function
    :math:`S` by Houdayer & Hartmann [6]_.
    It should attain a minimum of around :math:`1` for an optimal fit, and be
    much larger otherwise.

    For further information, see the :ref:`quality-function` section in the
    manual.

    References
    ----------
    .. [6] J. Houdayer and A. Hartmann, Physical Review B 70, 014418+ (2004)
        `doi:10.1103/physrevb.70.014418
        <http://dx.doi.org/doi:10.1103/physrevb.70.014418>`_

    '''

    # arguments should be 2-D array_like
    x = np.asanyarray(x)
    y = np.asanyarray(y)
    dy = np.asanyarray(dy)

    args = {"x": x, "y": y, "dy": dy}
    for arg_name, arg in args.items():
        if arg.ndim != 2:
            raise ValueError("{} should be 2-D array_like".format(arg_name))

    # arguments should have all the same shape
    if not x.shape == y.shape == dy.shape:
        raise ValueError("arguments should be of same shape")

    # x should be sorted for all system sizes l
    if not np.array_equal(x, np.sort(x, axis=1)):
        raise ValueError("x should be sorted for each system size")

    # dy should have only positive entries
    if not np.all(dy > 0.0):
        raise ValueError("dy should have only positive values")

    # first dimension: system sizes l
    # second dimension: parameter values rho
    k, n = x.shape

    # pre-calculate weights and other matrices
    w = dy ** (-2)
    wx = w * x
    wy = w * y
    wxx = w * x * x
    wxy = w * x * y

    # calculate master curve estimates
    master_y = np.zeros_like(y)
    master_y[:] = np.nan
    master_dy2 = np.zeros_like(dy)
    master_dy2[:] = np.nan

    # loop through system sizes
    for i in range(k):

        j_primes = _jprimes(x=x, i=i, x_bounds=x_bounds)

        # loop through x values
        for j in range(n):

            # discard x value if it is out of bounds
            try:
                if not x_bounds[0] <= x[i][j] <= x_bounds[1]:
                    continue
            except:
                pass

            # boolean mask for selected data x_l, y_l, dy_l
            select = _select_mask(j=j, j_primes=j_primes)

            if not select.any():
                # no data to select
                # master curve estimate Y_ij remains undefined
                continue

            # master curve estimate
            master_y[i, j], master_dy2[i, j] = _wls_linearfit_predict(
                x=x[i, j], w=w, wx=wx, wy=wy, wxx=wxx, wxy=wxy, select=select
            )

    # average within finite system sizes first
    return np.nanmean(
        np.nanmean(
            (y - master_y) ** 2 / (dy ** 2 + master_dy2),
            axis=1
        )
    )


def _neldermead_errors(sim, fsim, fun):
    """
    Estimate the errors from the final simplex of the Nelder--Mead algorithm

    This is a helper function and not supposed to be called directly.

    Parameters
    ----------
    sim : ndarray
        the final simplex

    fsim : ndarray
        the function values at the vertices of the final simplex

    fun : callable
        the goal function to minimize
    """

    # fit quadratic coefficients
    n = len(sim) - 1

    ymin = fsim[0]

    sim = np.copy(sim)
    fsim = np.copy(fsim)

    centroid = np.mean(sim, axis=0)
    fcentroid = fun(centroid)

    # enlarge distance of simplex vertices from centroid until all have at
    # least an absolute function value distance of 0.1
    for i in range(n + 1):
        while np.abs(fsim[i] - fcentroid) < 0.01:
            sim[i] += sim[i] - centroid
            fsim[i] = fun(sim[i])

    # the vertices and the midpoints x_ij
    x = 0.5 * (
        sim[np.mgrid[0:n + 1, 0:n + 1]][1] +
        sim[np.mgrid[0:n + 1, 0:n + 1]][0]
    )

    y = np.nan * np.ones(shape=(n + 1, n + 1))
    for i in range(n + 1):
        y[i, i] = fsim[i]
        for j in range(i + 1, n + 1):
            y[i, j] = y[j, i] = fun(x[i, j])

    y0i = y[np.mgrid[0:n + 1, 0:n + 1]][0][1:, 1:, 0]

    y0j = y[np.mgrid[0:n + 1, 0:n + 1]][0][0, 1:, 1:]

    b = 2 * (y[1:, 1:] + y[0, 0] - y0i - y0j)

    q = (sim - sim[0])[1:].T

    varco = ymin * np.dot(q, np.dot(np.linalg.inv(b), q.T))
    return np.sqrt(np.diag(varco)), varco


def autoscale(l, rho, a, da, rho_c0, nu0, zeta0, x_bounds=None, **kwargs):
```

```
"""
Automatically scale finite-size data and fit critical point and exponents

Parameters
----------
l, rho, a, da : array_like
    input for the :py:func:`scaledata` function

rho_c0, nu0, zeta0 : float
    initial guesses for the critical point and exponents

x_bounds : tuple of floats, optional
    lower and upper bound for scaled data `x` to consider

Returns
-------
res : OptimizeResult

res['success'] : bool
    Indicates whether the optimization algorithm has terminated
    successfully.

res['x'] : ndarray

res['rho'], res['nu'], res['zeta'] : float
    The fitted critical point and exponents, ``res['x'] == [res['rho'],
    res['nu'], res['zeta']]``

res['drho'], res['dnu'], res['dzeta'] : float
    The respective standard errors derived from fitting the curvature at
    the minimum, ``res['errors'] == [res['drho'], res['dnu'],
    res['dzeta']]``.

res['errors'], res['varco'] : ndarray
    The standard errors as a vector, and the full variance--covariance
    matrix (the diagonal entries of which are the squared standard errors),
    ``np.sqrt(np.diag(res['varco'])) == res['errors']``

See also
--------
scaledata
    For the `l`, `rho`, `a`, `da` input parameters

quality
    The goal function of the optimization

scipy.optimize.minimize
    The optimization wrapper routine

scipy.optimize.OptimizeResult
    The return type

Notes
-----
This implementation uses the quality function by Houdayer & Hartmann [8]_
which measures the quality of the data collapse, see the sections
:ref:`data-collapse-method` and :ref:`quality-function` in the manual.

This function and the whole fssa package have been inspired by Oliver
Melchert and his superb **autoScale** package [9]_.
```

```
    The critical point and exponents, including its standard errors and
    (co)variances, are fitted by the Nelder--Mead algorithm, see the section
    :ref:`neldermead` in the manual.

    References
    ----------
    .. [8] J. Houdayer and A. Hartmann, Physical Review B 70, 014418+ (2004)
        `doi:10.1103/physrevb.70.014418
        <http://dx.doi.org/doi:10.1103/physrevb.70.014418>`_

    .. [9] O. Melchert, `arXiv:0910.5403 <http://arxiv.org/abs/0910.5403>`_
       (2009)

    Examples
    --------
    >>> # generate artificial scaling data from master curve
    >>> # with rho_c == 1.0, nu == 2.0, zeta == 0.0
    >>> import fssa
    >>> l = [ 10, 100, 1000 ]
    >>> rho = np.linspace(0.9, 1.1)
    >>> l_mesh, rho_mesh = np.meshgrid(l, rho, indexing='ij')
    >>> master_curve = lambda x: 1. / (1. + np.exp( - x))
    >>> x = np.power(l_mesh, 0.5) * (rho_mesh - 1.)
    >>> y = master_curve(x)
    >>> dy = y / 100.
    >>> y += np.random.randn(*y.shape) * dy
    >>> a = y
    >>> da = dy
    >>>
    >>> # run autoscale
    >>> res = fssa.autoscale(l=l, rho=rho, a=a, da=da, rho_c0=0.9, nu0=2.0,
         zeta0=0.0)
    """

    def goal_function(x):
        my_x, my_y, my_dy = scaledata(
            rho=rho, l=l, a=a, da=da, nu=x[1], zeta=x[2], rho_c=x[0],
        )
        return quality(
            my_x, my_y, my_dy, x_bounds=x_bounds,
        )

    ret = scipy.optimize.minimize(
        goal_function,
        [rho_c0, nu0, zeta0],
        method=_minimize_neldermead,
        options={
            'xtol': 1e-2,
            'ftol': 1e-2,
        }
    )

    errors, varco = _neldermead_errors(
        sim=ret['final_simplex'][0],
        fsim=ret['final_simplex'][1],
        fun=goal_function,
    )

    ret['varco'] = varco
    ret['errors'] = errors
    ret['rho'], ret['nu'], ret['zeta'] = ret['x']
    ret['drho'], ret['dnu'], ret['dzeta'] = ret['errors']

    return ret
```

BIBLIOGRAPHY

[1] H. Ronellenfitsch and E. Katifori, “Global Optimization, Local Adaptation, and the Role of Growth in Distribution Networks,” Physical Review Letters **117**, 138301 (2016).

[2] C. Rueffler, J. Hermisson, and G. P. Wagner, “Evolution of functional specialization and division of labor,” Proceedings of the National Academy of Sciences **109**, E326 (2012).

[3] P. W. English, “The Origin and Spread of Qanats in the Old World,” Proceedings of the American Philosophical Society **112**, 170 (1968).

[4] A. T. Hodge, *Roman Aqueducts & Water Supply* (Duckworth, 2002), 518 pp.

[5] R. L. Hills, *Power from Steam: A History of the Stationary Steam Engine* (Cambridge University Press, Aug. 19, 1993), 360 pp.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A View of Cloud Computing,” Commun. ACM **53**, 50 (2010).

[7] T.-H. Hu, *A Prehistory of the Cloud* (MIT Press, Cambridge, MA, USA, 2015), 240 pp.

[8] E. Durkheim, *The Division of Labor in Society* (Simon and Schuster, Feb. 25, 2014), 416 pp.

[9] P. R. Krugman, *Geography and Trade* (MIT Press, 1991), 160 pp.

[10] N. Luhmann, *Social Systems* (Stanford University Press, 1995), 692 pp.

[11] A. Smith, *An Inquiry into the Nature and Causes of the Wealth of Nations* (W. Strahan and T. Cadell, London, 1776).

[12] K. W. Axhausen and T. Gärling, “Activity-based approaches to travel analysis: conceptual frameworks, models, and research problems,” Transport Reviews **12**, 323 (1992).

[13] S. Schönfelder and K. W. Axhausen, *Urban Rhythms and Travel Behaviour: Spatial and Temporal Phenomena of Daily Travel* (Ashgate Publishing, Ltd., 2010), 252 pp.

[14] P. C. Bressloff and J. M. Newby, “Stochastic models of intracellular transport,” Reviews of Modern Physics **85**, 135 (2013).

[15] M. Schliwa and G. Woehlke, “Molecular motors,” Nature **422**, 759 (2003).

[16] R. D. Vale, “The Molecular Motor Toolbox for Intracellular Transport,” Cell **112**, 467 (2003).

[17] P. Kaluza, A. Kolzsch, M. T. Gastner, and B. Blasius, “The complex network of global cargo ship movements,” Journal of The Royal Society Interface **7**, 1093 (2010).

[18] L. Hufnagel, D. Brockmann, and T. Geisel, "Forecast and control of epidemics in a globalized world," Proceedings of the National Academy of Sciences of the United States of America **101**, 15124 (2004).

[19] R. Louf and M. Barthelemy, "A typology of street patterns," Journal of The Royal Society Interface **11**, 20140924 (2014).

[20] E. Strano, V. Nicosia, V. Latora, S. Porta, and M. Barthélemy, "Elementary processes governing the evolution of road networks," Scientific Reports **2** (2012) `10.1038/srep00296`.

[21] M. Barthélemy and A. Flammini, "Modeling Urban Street Patterns," Physical Review Letters **100** (2008) `10.1103/PhysRevLett.100.138702`.

[22] D. Brockmann, L. Hufnagel, and T. Geisel, "The scaling laws of human travel," Nature **439**, 462 (2006).

[23] M. C. González, C. A. Hidalgo, and A.-L. Barabási, "Understanding individual human mobility patterns," Nature **453**, 779 (2008).

[24] K. Windt and M. Hülsmann, "Changing Paradigms in Logistics — Understanding the Shift from Conventional Control to Autonomous Cooperation and Control," in *Understanding Autonomous Cooperation and Control in Logistics*, edited by P. D. M. Hülsmann and D.-I. K. Windt (Springer Berlin Heidelberg, 2007), pp. 1–16.

[25] D. E. Bloom, D. Canning, and G. Fink, "Urbanization and the Wealth of Nations," Science **319**, 772 (2008).

[26] United Nations Population Fund, *State of World Population 2007* (New York, NY, USA, 2007).

[27] United Nations, *World Urbanization Prospects: The 2014 Revision* (2015).

[28] WBGU – German Adivsory Council on Global Change, ed., *Humanity on the move: Unlocking the transformative power of cities* (WBGU, Berlin, 2016), 514 pp.

[29] D. Banister, K. Anderton, D. Bonilla, M. Givoni, and T. Schwanen, "Transportation and the Environment," Annual Review of Environment and Resources **36**, 247 (2011).

[30] J. Rockstrom, W. Steffen, K. Noone, A. Persson, F. S. Chapin III, E. Lambin, T. M. Lenton, M. Scheffer, C. Folke, H. J. Schellnhuber, B. Nykvist, C. A. de Wit, T. Hughes, S. van der Leeuw, H. Rodhe, S. Sorlin, P. K. Snyder, R. Costanza, U. Svedin, M. Falkenmark, L. Karlberg, R. W. Corell, V. J. Fabry, J. Hansen, B. Walker, D. Liverman, K. Richardson, P. Crutzen, and J. Foley, "Planetary Boundaries: Exploring the Safe Operating Space for Humanity," Ecology and Society **14** (2009).

[31] WBGU – German Advisory Council on Global Change, ed., *World in Transition: A Social Contract for Sustainability* (WBGU, Berlin, 2011), 396 pp.

[32] P. D. H. Kagermann, "Change Through Digitization—Value Creation in the Age of Industry 4.0," in *Management of Permanent Change*, edited by H. Albach, H. Meffert, A. Pinkwart, and R. Reichwald (Springer Fachmedien Wiesbaden, 2015), pp. 23–45.

[33] J. P. Jokinen, T. Sihvola, E. Hyytiä, and R. Sulonen, "Why urban mass demand responsive transport?" In 2011 IEEE Forum on Integrated and Sustainable Transportation System (FISTS) (June 2011), pp. 317–322.

[34] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Quantifying the Benefits of Vehicle Pooling with Shareability Networks," Proceedings of the National Academy of Sciences **111**, 13290 (2014).

[35] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Reply to Lopez et al.: Sustainable implementation of taxi sharing requires understanding systemic effects," Proceedings of the National Academy of Sciences **111**, E5489 (2014).

[36] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," Proceedings of the National Academy of Sciences, 201611675 (2017).

[37] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route Planning in Transportation Networks," (2015).

[38] E. Hyytiä, A. Penttinen, and R. Sulonen, "Congestive Collapse and its Avoidance in a Dynamic Dial-a-Ride System with Time Windows," in International Conference on Analytical and Stochastic Modeling Techniques and Applications (2010), pp. 397–408.

[39] A. Horni, K. Nagel, and K. W. Axhausen, eds., *The Multi-Agent Transport Simulation MATSim* (Ubiquity Press, London, Aug. 10, 2016).

[40] M. Rohden, A. Sorge, M. Timme, and D. Witthaut, "Self-Organized Synchronization in Decentralized Power Grids," Physical Review Letters **109**, 064101 (2012).

[41] J. H. Brown, J. F. Gillooly, A. P. Allen, V. M. Savage, and G. B. West, "Toward a metabolic theory of ecology," Ecology **85**, 1771 (2004).

[42] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," Science **286**, 509 (1999).

[43] L. M. A. Bettencourt, "The Origins of Scaling in Cities," Science **340**, 1438 (2013).

[44] R. Tachet, O. Sagarra, P. Santi, G. Resta, M. Szell, S. H. Strogatz, and C. Ratti, "Scaling Law of Urban Ride Sharing," Scientific Reports **7**, 42868 (2017).

[45] N. Ferreira, J. Poco, H. T. Vo, J. Freire, and C. T. Silva, "Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips," IEEE Transactions on Visualization and Computer Graphics **19**, 2149 (2013).

[46] V. Belik, T. Geisel, and D. Brockmann, "Natural Human Mobility Patterns and Spatial Spread of Infectious Diseases," Physical Review X **1**, 011001 (2011).

[47] F. Simini, M. C. González, A. Maritan, and A.-L. Barabási, “A universal model for mobility and migration patterns,” Nature **484**, 96 (2012).

[48] C. M. Schneider, V. Belik, T. Couronné, Z. Smoreda, and M. C. González, “Unravelling daily human mobility motifs,” Journal of The Royal Society Interface **10**, 20130246 (2013).

[49] R. Sinatra, P. Deville, M. Szell, D. Wang, and A.-L. Barabási, “A century of physics,” Nature Physics **11**, 791 (2015).

[50] L. P. Kadanoff, “More is the Same; Phase Transitions and Mean Field Theories,” Journal of Statistical Physics **137**, 777 (2009).

[51] S. Herminghaus, *Mean-field approach to demand-driven public transportation*, 43 (MPI for Dynamics and Self-Organization, Göttingen, 2015).

[52] A. Sorge, *Demand-Driven Directed Transport (D3T) Specification*, Dec. 3, 2015.

[53] A. Sorge, M. Timme, and D. Manik, “The Simulation Unification: Towards a Python Toolbox to Model and Analyze Collective Mobility Systems,” Contributed Talk, Dresden, Mar. 21, 2017.

[54] A. Sorge, D. Manik, J. Nagler, and M. Timme, “Temporal Percolation in Critical Collective Mobility Systems,” Invited Talk, Dresden, Mar. 22, 2017.

[55] A. Sorge, J. Nagler, S. Herminghaus, and M. Timme, “To return or not to return? Phase transition to transience and congestion in random walks and queueing systems,” Contributed Talk, Berlin, Mar. 18, 2015.

[56] G. Poore, *Pythontex*, version 0.15, 2016.

[57] G. M. Poore, “PythonTeX: reproducible documents with LaTeX, Python, and more,” Computational Science & Discovery **8**, 014010 (2015).

[58] G. M. Poore, “Reproducible Documents with PythonTeX,” in Proceedings of the 12th Python in Science Conference, edited by S. van der Walt, J. Millman, and K. Huff (2013), pp. 78–84.

[59] L. Pantieri, *ArsClassica – A different view on the ClassicThesis package*, version March 2017, 2017.

[60] A. Miede, *Package ClassicThesis*, version 4.2, 2015.

[61] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” Computing in Science Engineering **13**, 22 (2011).

[62] E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–.

[63] T. E. Oliphant, “Python for Scientific Computing,” Computing in Science Engineering **9**, 10 (2007).

[64] K. J. Millman and M. Aivazis, “Python for Scientists and Engineers,” Computing in Science Engineering **13**, 9 (2011).

[65] J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” Computing in Science & Engineering **9**, 90 (2007).

[66] F. Pérez and B. E. Granger, “IPython: a System for Interactive Scientific Computing,” Computing in Science and Engineering **9**, 21 (2007).

[67] T. Kluyver, R.-K. Benjamin, P. Fernando, G. Brian, B. Matthias, F. Jonathan, K. Kyle, H. Jessica, G. Jason, C. Sylvain, I. Paul, A. Dami&aacute;n, A. Safia, W. Carol, and J. D. Team, "Jupyter Notebooks – a publishing format for reproducible computational workflows," Stand Alone, 87 (2016).

[68] M. Droettboom, T. A. Caswell, J. Hunter, and et al., "Matplotlib v2.0.1," Zenodo (2017) `10.5281/zenodo.570311`.

[69] A. Collette, *Python and HDF5* (O'Reilly, 2013).

[70] M. Waskom, O. Botvinnik, drewokane, P. Hobson, David, Y. Halchenko, S. Lukauskas, J. B. Cole, J. Warmenhoven, J. de Ruiter, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, M. Martin, A. Miles, K. Meyer, T. Augspurger, T. Yarkoni, P. Bachant, M. Williams, C. Evans, C. Fitzgerald, Brian, D. Wehner, G. Hitz, E. Ziegler, A. Qalieh, and A. Lee, "Seaborn v0.7.1," Zenodo (2016) `10.5281/zenodo.54844`.

[71] Dask Development Team, *Dask: Library for dynamic task scheduling* (2016).

[72] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in Proceedings of the 14th Python in Science Conference, edited by K. Huff and J. Bergstra (2015), pp. 130–136.

[73] P. W. Anderson, "More Is Different," Science **177**, 393 (1972).

[74] R. B. Laughlin and D. Pines, "The Theory of Everything," Proceedings of the National Academy of Sciences **97**, 28 (2000).

[75] K. Binder, "Theory of first-order phase transitions," Reports on Progress in Physics **50**, 783 (1987).

[76] H. E. Stanley, *Introduction to phase transitions and critical phenomena.* (Oxford University Press, 1987).

[77] D. Sornette, *Critical phenomena in natural sciences: chaos, fractals, selforganization, and disorder: concepts and tools*, 2nd ed, Springer series in synergetics (Springer, Berlin ; New York, 2004), 528 pp.

[78] M. Scheffer, *Critical transitions in nature and society* (Princeton University Press, Princeton, N.J., 2009).

[79] A. Lesne and M. Lagües, *Scale Invariance* (Springer, Berlin, Heidelberg, 2012).

[80] C. Kuehn, "A mathematical framework for critical transitions: Bifurcations, fast–slow systems and stochastic dynamics," Physica D: Nonlinear Phenomena **240**, 1020 (2011).

[81] C. Kuehn, "A Mathematical Framework for Critical Transitions: Normal Forms, Variance and Applications," Journal of Nonlinear Science **23**, 457 (2013).

[82] M. Scheffer, J. Bascompte, W. A. Brock, V. Brovkin, S. R. Carpenter, V. Dakos, H. Held, E. H. van Nes, M. Rietkerk, and G. Sugihara, "Early-warning signals for critical transitions," Nature **461**, 53 (2009).

[83] X. Zhang, C. Kuehn, and S. Hallerberg, "Predictability of critical transitions," Physical Review E **92**, 052905 (2015).

[84] M. Scheffer, S. R. Carpenter, T. M. Lenton, J. Bascompte, W. Brock, V. Dakos, J. Van De Koppel, I. A. Van De Leemput, S. A. Levin, E. H. Van Nes, et al., "Anticipating critical transitions," science **338**, 344 (2012).

[85] H. E. Stanley, "Scaling, universality, and renormalization: Three pillars of modern critical phenomena," Reviews of Modern Physics **71**, S358 (1999).

[86] D. Stauffer and A. Aharony, *Introduction to percolation theory* (Taylor & Francis, London, 1994).

[87] A. Hunt, R. Ewing, and B. Ghanbarian, *Percolation Theory for Flow in Porous Media*, Vol. 880, Lecture Notes in Physics (Springer International Publishing, Cham, 2014).

[88] M. Sahimi, *Applications of percolation theory* (Taylor & Francis, 2014).

[89] D. Lee, Y. S. Cho, and B. Kahng, "Diverse types of percolation transitions," Journal of Statistical Mechanics: Theory and Experiment **2016**, 124002 (2016).

[90] A. A. Saberi, "Recent advances in percolation theory and its applications," Physics Reports **578**, 1 (2015).

[91] P. J. Flory, "Molecular size distribution in three dimensional polymers. i. gelation1," Journal of the American Chemical Society **63**, 3083 (1941).

[92] W. H. Stockmayer, "Theory of Molecular Size Distribution and Gel Formation in Branched-Chain Polymers," The Journal of Chemical Physics **11**, 45 (1943).

[93] S. R. Broadbent and J. M. Hammersley, "Percolation processes: I. Crystals and mazes," Mathematical Proceedings of the Cambridge Philosophical Society **53**, 629 (1957).

[94] D. Stauffer, "Scaling theory of percolation clusters," Physics reports **54**, 1 (1979).

[95] M. Fisher, "The Theory of Condensation and the Critical Point," Physics **3**, 255 (1967).

[96] D. Stauffer and C. Jayaprakash, "Critical exponents for one-dimensional percolation clusters," Physics Letters A **64**, 433 (1978).

[97] R. M. D'Souza and J. Nagler, "Anomalous critical and supercritical phenomena in explosive percolation," Nature Physics **11**, 531 (2015).

[98] W. Chen, M. Schröder, R. M. D'Souza, D. Sornette, and J. Nagler, "Microtransition Cascades to Percolation," Physical Review Letters **112** (2014) `10.1103/PhysRevLett.112.155701`.

[99] J. Nagler, T. Tiessen, and H. W. Gutch, "Continuous Percolation with Discontinuities," Physical Review X **2** (2012) `10.1103/PhysRevX.2.031009`.

[100] P. Erdös and A. Rényi, "On the Evolution of Random Graphs," Publications of the Mathematical Institute of the Hungarian Academy of Science, 17 (1960).

[101] Y. S. Cho, M. G. Mazza, B. Kahng, and J. Nagler, "Genuine non-self-averaging and ultraslow convergence in gelation," Physical Review E **94**, 022602 (2016).

[102] M. Schröder, W. Chen, and J. Nagler, "Discrete scale invariance in supercritical percolation," New Journal of Physics **18**, 013042 (2016).

[103] A. A. Saberi, S. H. E. Rahbari, H. Dashti-Naserabadi, A. Abbasi, Y. S. Cho, and J. Nagler, "Universality in boundary domain growth by sudden bridging," Scientific Reports **6** (2016) `10.1038/srep21110`.

[104] N. Araújo, P. Grassberger, B. Kahng, K. J. Schrenk, and R. M. Ziff, "Recent advances and open challenges in percolation," The European Physical Journal Special Topics **223**, 2307 (2014).

[105] J. Nagler, A. Levina, and M. Timme, "Impact of single links in competitive percolation," Nature Physics **7**, 265 (2011).

[106] Y. S. Cho, S. Hwang, H. J. Herrmann, and B. Kahng, "Avoiding a Spanning Cluster in Percolation Models," Science **339**, 1185 (2013).

[107] D. Achlioptas, R. M. D'Souza, and J. Spencer, "Explosive Percolation in Random Networks," Science **323**, 1453 (2009).

[108] E. J. Friedman and A. S. Landsberg, "Construction and Analysis of Random Networks with Explosive Percolation," Physical Review Letters **103** (2009) `10.1103/PhysRevLett.103.255701`.

[109] R. A. da Costa, S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "Explosive Percolation Transition is Actually Continuous," Physical Review Letters **105** (2010) `10.1103/PhysRevLett.105.255701`.

[110] M. Schröder, S. H. E. Rahbari, and J. Nagler, "Crackling noise in fractional percolation," Nature Communications **4** (2013) `10.1038/ncomms3222`.

[111] M. Sheinman, A. Sharma, J. Alvarado, G. H. Koenderink, and F. C. MacKintosh, "Anomalous Discontinuity at the Percolation Critical Point of Active Gels," Physical Review Letters **114** (2015) `10.1103/PhysRevLett.114.098104`.

[112] A. Bar and D. Mukamel, "Mixed-Order Phase Transition in a One-Dimensional Model," Physical Review Letters **112**, 015701 (2014).

[113] J. M. Schwarz, A. J. Liu, and L. Q. Chayes, "The onset of jamming as the sudden emergence of an infinite k-core cluster," EPL (Europhysics Letters) **73**, 560 (2006).

[114] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "K -Core Organization of Complex Networks," Physical Review Letters **96** (2006) `10.1103/PhysRevLett.96.040601`.

[115] D. Lee, W. Choi, J. Kertész, and B. Kahng, "Universal mechanism for hybrid percolation transitions," (2016).

[116] K. Binder and D. W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Vol. 0, Graduate Texts in Physics (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010).

[117] M. E. J. Newman and G. T. Barkema, *Monte Carlo methods in statistical physics* (Clarendon Press ; Oxford University Press, Oxford; New York, 1999).

[118] M. E. Fisher and M. N. Barber, "Scaling Theory for Finite-Size Effects in the Critical Region," Physical Review Letters **28**, 1516 (1972).

[119] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering* (Avalon Publishing, 1994), 520 pp.

[120] J. J. Sakurai and J. Napolitano, *Modern Quantum Mechanics* (Addison-Wesley, 2011), 550 pp.

[121] C. Werndl, "Are deterministic descriptions and indeterministic descriptions observationally equivalent?" Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics **40**, 232 (2009).

[122] V. Capasso and D. Bakstein, *An Introduction to Continuous-Time Stochastic Processes*, Modeling and Simulation in Science, Engineering and Technology (Birkhäuser Boston, Boston, MA, 2012).

[123] R. Serfozo, *Basics of Applied Stochastic Processes*, red. by J. Gani, C. Heyde, P. Jagers, and T. G. Kurtz, Probability and Its Applications (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009).

[124] A. Khintchine, "Korrelationstheorie der stationären stochastischen Prozesse," Mathematische Annalen **109**, 604 (1934).

[125] D. Gusak, A. Kukush, A. Kulik, Y. Mishura, and A. Pilipenko, *Theory of Stochastic Processes*, Problem Books in Mathematics (Springer New York, New York, NY, 2010).

[126] D. J. Daley and D. Vere-Jones, *An Introduction to the Theory of Point Processes: Volume II: General Theory and Structure*, Probability and Its Applications (Springer, New York, NY, 2008).

[127] R. G. Gallager, *Stochastic Processes: Theory for Applications* (Cambridge University Press, 2013), 536 pp.

[128] S. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability* (Cambridge University Press, 2009), 623 pp.

[129] A. A. Borovkov, *Probability Theory*, Universitext (Springer, London, 2013).

[130] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems* (Acad. Press, Amsterdam, 2010).

[131] G. A. Wainer, *Discrete-Event Modeling and Simulation: a Practitioner's Approach* (CRC Press, Boca Raton, 2009).

[132] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism," in Proceedings of Winter Simulation Conference (Dec. 1994), pp. 716–722.

[133] J. J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++* (John Wiley & Sons, Inc., Hoboken, NJ, USA, Dec. 6, 2010).

[134] S. Raczynski, *Modeling and Simulation: The Computer Science of Illusion* (Wiley, Chichester, England; Hoboken, NJ; Baldock, Hertfordshire, 2006).

[135] A. Syropoulos, "Mathematics of Multisets," in Multiset Processing (Aug. 21, 2000), pp. 347–358.

[136] J. F. Thomson, "Tasks and Super-Tasks," Analysis **15**, 1 (1954).

[137] N. Huggett, "Zeno's Paradoxes," in *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Winter 2010 (Metaphysics Research Lab, Stanford University, 2010).

[138] Aristotle, *Physics* (350 BCE).

[139] B. Scholz-Reiter, K. Windt, and M. Freitag, "Autonomous logistic processes: New demands and first approaches," in Proceedings of the 37th CIRP international seminar on manufacturing systems (2004), pp. 357–362.

[140] M. Hülsmann, B. Scholz-Reiter, and K. Windt, eds., *Autonomous Cooperation and Control in Logistics* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011).

[141] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia, "A Review of Dynamic Vehicle Routing Problems," European Journal of Operational Research **225**, 1 (2013).

[142] L. Häme, "Demand-Responsive Transport: Models & Algorithms," PhD thesis (School of Science, Aalto University, Espoo, Finland, 2013).

[143] G. Berbeglia, J.-F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," European Journal of Operational Research **202**, 8 (2010).

[144] A. Sorge, D. Manik, S. Herminghaus, and M. Timme, "Towards a unifying framework for demand-driven directed transport (D3T)," in Proceedings of the 2015 Winter Simulation Conference (2015), pp. 2800–2811.

[145] M. M. Deza and E. Deza, "General Definitions," in *Encyclopedia of Distances* (Springer, Berlin, Heidelberg, 2016), pp. 3–62.

[146] J. J. Nutaro, *Adevs (A Discrete Event Simulator) Library*, 2013.

[147] D. Gross, J. F. Shortie, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory* (Wiley, Hoboken, New Jersey, 2008).

[148] A. K. Erlang, "The theory of probabilities and telephone conversations," Nyt Tidsskrift for Matematik B **20**, 16 (1909).

[149] P. Van Mieghem, *Performance analysis of complex networks and systems.* (Cambridge University Press, Cambridge, 2014).

[150] T. Jia and R. V. Kulkarni, "Intrinsic Noise in Stochastic Models of Gene Expression with Molecular Memory and Bursting," Physical Review Letters **106** (2011) `10.1103/PhysRevLett.106.058102`.

[151] N. A. Cookson, W. H. Mather, T. Danino, O. Mondragon-Palomino, R. J. Williams, L. S. Tsimring, and J. Hasty, "Queueing up for enzymatic processing: correlated signaling through coupled degradation," Molecular Systems Biology **7**, 561 (2014).

[152] S. Halfin and W. Whitt, "Heavy-Traffic Limits for Queues with Many Exponential Servers," Operations Research **29**, 567 (1981).

[153] D. Gamarnik and D. A. Goldberg, "Steady-state GI/G/n queue in the Halfin–Whitt regime," The Annals of Applied Probability **23**, 2382 (2013).

[154] W. Whitt, *Stochastic-process limits: an introduction to stochastic-process limits and their application to queues* (Springer, New York, 2002), 602 pp.

[155] H. C. Tijms, *A First Course in Stochastic Models* (Wiley, New York, 2003), 478 pp.

[156] A. Arenas, A. Díaz-Guilera, and R. Guimerà, "Communication in Networks with Hierarchical Branching," Physical Review Letters **86**, 3196 (2001).

[157] H. Poincaré, "Sur le probleme des trois corps et les équations de la dynamique," Acta mathematica **13**, A3 (1890).

[158] V. Balakrishnan, G. Nicolis, and C. Nicolis, "Recurrence time statistics in chaotic dynamics: multiple recurrences in intermittent chaos," Stochastics and Dynamics **01**, 345 (2001).

[159] N. Hadyn, J. Luevano, G. Mantica, and S. Vaienti, "Multifractal Properties of Return Time Statistics," Physical Review Letters **88** (2002) 10.1103/PhysRevLett.88.224502.

[160] M. Hirata, B. Saussol, and S. Vaienti, "Statistics of Return Times:¶A General Framework and New Applications," Communications in Mathematical Physics **206**, 33 (1999).

[161] G. Robinson and M. Thiel, "Recurrences determine the dynamics," Chaos: An Interdisciplinary Journal of Nonlinear Science **19**, 023104 (2009).

[162] J. B. Gao, "Recurrence Time Statistics for Chaotic Systems and Their Applications," Physical Review Letters **83**, 3178 (1999).

[163] V. Balakrishnan, G. Nicolis, and C. Nicolis, "Recurrence time statistics in chaotic dynamics. I. Discrete time maps," Journal of Statistical Physics **86**, 191 (1997).

[164] W. Feller, "Fluctuation Theory of Recurrent Events," Transactions of the American Mathematical Society **67**, 98 (1949).

[165] S. Foss and T. Konstantopoulos, "An Overview of Some Stochastic Stability Methods," Journal of the Operations Research Society of Japan **47**, 275 (2004).

[166] A. A. Borovkov and V. Yurinsky, *Ergodicity and stability of stochastic processes* (J. Wiley, Chichester [England]; New York, 1998).

[167] G. Iommi and M. Todd, "Transience in dynamical systems," Ergodic Theory and Dynamical Systems **33**, 1450 (2013).

[168] O. Sarig, "Continuous Phase Transitions for Dynamical Systems," Communications in Mathematical Physics **267**, 631 (2006).

[169] S. Maslov, "Infinite series of exact equations in the Bak-Sneppen model of biological evolution," Physical review letters **77**, 1182 (1996).

[170] M. E. Fisher and B. Felderhof, "Phase transitions in one-dimensional cluster-interaction fluids IA. Thermodynamics," Annals of Physics **58**, 176 (1970).

[171] B. Felderhof and M. E. Fisher, "Phase transitions in one-dimensional cluster-interaction fluids II. Simple logarithmic model," Annals of Physics **58**, 268 (1970).

[172] E. Frey and K. Kroy, "Brownian motion: a paradigm of soft matter and biological physics," Annalen der Physik **14**, 20 (2005).

[173] D. S. Grebenkov, "NMR survey of reflected Brownian motion," Reviews of Modern Physics **79**, 1077 (2007).

[174] I. Bajunaid, J. M. Cohen, F. Colonna, and D. Singman, "Function Series, Catalan Numbers, and Random Walks on Trees," The American Mathematical Monthly **112**, 765 (2005).

[175] D. I. Cohen and T. M. Katz, "Recurrence of a random walk on the half-line as the generating function for the Catalan numbers," Discrete Mathematics **29**, 213 (1980).

[176] *A000108 – Catalan numbers*, `https://oeis.org/A000108` (visited on 04/22/2017).

[177] S. Asmussen, *Applied Probability and Queues*, Vol. 51, Stochastic Modelling and Applied Probability (Springer, New York, NY, 2003).

[178] W. J. Stewart, *Probality, Markov chains, queues, ans simulation: the mathematical basis of performance modeling* (Princeton University Press, Princeton, 2009).

[179] L. Takács, *Introduction to the theory of queues.* (Oxford University Press, New York, 1962).

[180] M. Draief and J. Mairesse, "Services within a busy period of an M/M/1 queue and Dyck paths," Queueing Systems **49**, 73 (2005).

[181] S. Maslov, M. Paczuski, and P. Bak, "Avalanches and 1 f noise in evolution and growth models," Physical Review Letters **73**, 2162 (1994).

[182] M. Paczuski, S. Maslov, and P. Bak, "Avalanche dynamics in evolution, growth, and depinning models," Physical Review E **53**, 414 (1996).

[183] P. Bak and K. Sneppen, "Punctuated equilibrium and criticality in a simple model of evolution," Physical review letters **71**, 4083 (1993).

[184] P. Bak, C. Tang, and K. Wiesenfeld, "Self-Organized Criticality: An Explanation of l/f Noise," Self **59**, 27 (1987).

[185] J. Houdayer and A. Hartmann, "Low-temperature behavior of two-dimensional Gaussian Ising spin glasses," Physical Review B **70** (2004) `10.1103/PhysRevB.70.014418`.

[186] O. Melchert, "autoScale.py - A program for automatic finite-size scaling analyses: A user's guide," (2009).

[187] S. M. Bhattacharjee and F. Seno, "A measure of data collapse for scaling," Journal of Physics A: Mathematical and General **34**, 6375 (2001).

[188] S. Wenzel, E. Bittner, W. Janke, and A. M. Schakel, "Percolation of vortices in the 3D Abelian lattice Higgs model," Nuclear Physics B **793**, 344 (2008).

[189] N. Kawashima and N. Ito, "Critical Behavior of the Three-Dimensional $\pm J$ Model in a Magnetic Field," Journal of the Physical Society of Japan **62**, 435 (1993).

[190] P. R. Bevington and D. K. Robinson, *Data reduction and error analysis for the physical sciences* (McGraw-Hill, Boston [u.a., 2010).

[191] T. Strutz, *Data Fitting and Uncertainty* (Springer Vieweg, Wiesbaden, 2016).

[192] J. A. Nelder and R. Mead, "A simplex method for function minimization," The Computer Journal **7**, 308 (1965).

[193] T. Kolda, R. Lewis, and V. Torczon, "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods," SIAM Review **45**, 385 (2003).

[194] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the Nelder–Mead simplex method in low dimensions," SIAM Journal on optimization **9**, 112 (1998).

[195] C. J. Price, I. D. Coope, and D. Byatt, "A Convergent Variant of the Nelder–Mead Algorithm," Journal of Optimization Theory and Applications **113**, 5 (2002).

[196] S. Singer and J. Nelder, "Nelder-Mead algorithm," Scholarpedia **4**, 2928 (2009).

[197] S. Singer and S. Singer, "Efficient Implementation of the Nelder-Mead Search Algorithm," Applied Numerical Analysis & Computational Mathematics **1**, 524 (2004).

[198] W. Spendley, G. R. Hext, and F. R. Himsworth, "Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation," Technometrics **4**, 441 (1962).

[199] A. Sorge, "Pyfssa 0.7.6 – Scientific Python Package for finite-size scaling analysis," Zenodo (2015) `10.5281/zenodo.35293`.

[200] A. Sorge, *Pyfssa Documentation*, (2015) `http://pyfssa.readthedocs.io/en/stable/` (visited on 04/28/2017).

[201] A. Sorge, *ENH: Nelder-Mead to optionally return final simplex by andsor · Pull Request #5205 · scipy/scipy*, Aug. 30, 2015.

[202] O. Melchert, H. G. Katzgraber, and M. A. Novotny, "Site- and bond-percolation thresholds in $K_{n,n}$-based lattices: Vulnerability of quantum annealers to random qubit and coupler failures on chimera topologies," Physical Review E **93**, 042128 (2016).

[203] C. Castellano and R. Pastor-Satorras, "On the numerical study of percolation and epidemic critical properties in networks," The European Physical Journal B **89**, 243 (2016).

[204] E. Schofield, *Python-future*, in collab. with et al., version 0.16.0, 2016.

[205] A. Sorge, "Pypercolate 0.4.6 – Scientific Python package for Monte-Carlo simulation of percolation on graphs," Zenodo (2015) `10.5281/zenodo.35305`.

[206] M. E. J. Newman and R. M. Ziff, "Fast Monte Carlo algorithm for site or bond percolation," Physical Review E **64** (2001) `10.1103/PhysRevE.64.016706`.

[207] S. Asmussen and P. W. Glynn, "Variance-Reduction Methods," in *Stochastic Simulation: Algorithms and Analysis*, Stochastic Modelling and Applied Probability 57 (Springer New York, 2007), pp. 126–157.

[208] J. VanderPlas, *Frequentism and Bayesianism III: Confidence, Credibility, and why Frequentism and Science do not Mix | Pythonic Perambulations*, (June 12, 2014) `http://jakevdp.github.io/blog/2014/06/12/frequentism-and-bayesianism-3-confidence-credibility/` (visited on 04/29/2017).

[209] E. Cameron, "On the Estimation of Confidence Intervals for Binomial Population Proportions in Astronomy: The Simplicity and Superiority of the Bayesian Approach," Publications of the Astronomical Society of Australia **28**, 128 (2011).

[210] A. Agresti and B. A. Coull, "Approximate Is Better than "Exact" for Interval Estimation of Binomial Proportions," The American Statistician **52**, 119 (1998).

[211] A. DasGupta, T. T. Cai, and L. D. Brown, "Interval Estimation for a Binomial Proportion," Statistical Science **16**, 101 (2001).

[212] L. Wasserman, *All of Statistics*, Springer Texts in Statistics (Springer New York, New York, NY, 2004).

[213] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap* (Chapman & Hall, New York, 1993).

[214] A. Sorge, *pytemper – Scientific Python package for finite-time analysis of the recurrence-transience transition in the temporal percolation paradigm*, version 0.3.2, 2017.

[215] C. G. Evans, *Scikits-bootstrap – a Python/Scipy package for bootstrap statistics and confidence interval estimation*, version 0.3.2, 2016.

[216] NumPy Developers, *Structured arrays — NumPy v1.12 Manual*, (2017) `https://docs.scipy.org/doc/numpy/user/basics.rec.html` (visited on 04/29/2017).

[217] P. L. Krapivsky, S. Redner, and F. Leyvraz, "Connectivity of Growing Random Networks," Physical Review Letters **85**, 4629 (2000).

[218] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin, "Structure of Growing Networks with Preferential Linking," Physical Review Letters **85**, 4633 (2000).

[219] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. A. Ioannidis, and M. Taufer, "Enhancing reproducibility for computational methods," Science **354**, 1240 (2016).

[220] V. Stodden and S. Miguez, "Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research," Journal of Open Research Software **2** (2014) `10.5334/jors.ay`.

[221] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden, "Reproducible Research in Computational Harmonic Analysis," Computing in Science Engineering **11**, 8 (2009).

[222] B. Blanton and C. Lenhardt, "A Scientist's Perspective on Sustainable Scientific Software," Journal of Open Research Software **2** (2014) `10.5334/jors.ba`.

[223] A. Al Hanbali, M. Mandjes, Y. Nazarathy, and W. Whitt, "The asymptotic variance of departures in critically loaded queues," Advances in Applied Probability **43**, 243 (2011).

[224] A. P. Zwart, "Tail Asymptotics for the Busy Period in the GI/G/1 Queue," Mathematics of Operations Research **26**, 485 (2001).

[225] E. Morozov, "Weak Regeneration in Modeling of Queueing Processes," Queueing Systems **46**, 295 (2004).

[226] E. Morozov and R. Delgado, "Stability analysis of regenerative queueing systems," Automation and Remote Control **70**, 1977 (2009).

[227] K. Sigman and R. Wolff, "A Review of Regenerative Processes," SIAM Review **35**, 269 (1993).

[228] D. Witthaut and M. Timme, "Braess's paradox in oscillator networks, desynchronization and power outage," New Journal of Physics **14**, 083036 (2012).

[229] J. Bang-Jensen and G. Z. Gutin, *Digraphs*, Springer Monographs in Mathematics (Springer London, London, 2009).