UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master Degree in Physics of Data

Final Dissertation

Scaling Laws in microbial growth

Thesis supervisor:
Prof. Samir Suweis

External thesis supervisor:
Prof. Carlo Albert

Candidate:
Tommaso Amico

Academic Year 2022/2023

Abstract

Microbial growth and division are fundamental processes shaping organisms' life cycle. Mathematical models of such biological processes have been developed, but several questions remain open, especially when focusing on single-cell lineages.
The development of new microfluidic devices, combining single-molecule microscopy and automated image analysis, allows to track individual cells and their quantities of interest for many generations.
Furthermore, recent discoveries of scaling laws hint at the existence of universal growth laws, not yet formulated.

In this thesis work we hypothesize that cells operate in the vicinity of a critical point and we therefore aim at describing temporal and size scaling of lineages both from a theoretical and an empirical point of view.
Our first step is indeed an analytical one, carried out with the purpose of putting the problem in the proper theoretical framework.
Simulations are coded in order to explore the behavior of the system at different distances from the critical point. They are based on various models having different numbers of traits which range from 1 to 2, but belonging to the same universality class. Finally, we want to probe the proposed theoretical results by making use of available cell data from two different experiments and comparing them to the predictions of our model.

Analysis' results are presented both for the numerical approach and for the comparison with experimental data. We show how simulations confirm the hypothesized scalings and clarify exponents upon which different theoretical results can be found in literature.
Furthermore, the moment scaling we find in experimental data is consistent with the criticality hypothesis. To corroborate this result we infer the control parameters of a particular model from the universality class with the aid of Bayesian Inference (BI) and find that they are indeed close to the critical point.
Data quality shows to be an hurdle in probing the scaling of the autocorrelation length ξ with respect to $\langle m \rangle$. No clear behaviour arises indeed from the analysis.

Contents

1 Introduction

Through the years, significant effort has been put in answering fundamental questions on microbial growth and division [1]. Understanding the mechanisms behind cell size control is pivotal for comprehending how microorganisms regulate their growth, division, and overall population dynamics. In spite of the amount of research, significant aspects of these fundamental processes are not yet fully understood as attempts to answer some of the long-standing questions on the subject have been severely hindered by restricted experimental access to single-cell attributes [1, 2]. This underscores the ongoing relevance of studying microbial growth as it continues to yield new insights into biological processes thanks to recent advancements in experimental techniques.
Furthermore key traits of microbes, e.g. cell size, often show self-similar behaviour [3] which has as a common feature power law scaling, hinting at the possibility of an underlying critical process [3, 4, 11].
The ubiquity of scale invariance in ecological systems has been explained by self-organized criticality i.e., systems formed by many interacting parts spontaneously evolve towards the critical point, leading to dynamically diverging correlation lengths, making the scaling behaviour independent of microscopic details [3, 4].

In [3], the authors analyze observed scaling in cellular properties, such as size, of phytoplankton which is responsible for half of all primary production on Earth [36]. This highlights their critical role in global nutrient cycles. We have indeed that phytoplankton growth and activity directly impact the availability of essential nutrients in aquatic ecosystems, which, in turn, affects the entire food web. The authors propose that essential aspects of the phytoplankton response to changing environmental conditions can be described, in a robust quantitative manner, by scaling laws. Understanding these scaling laws is crucial for predicting how phytoplankton populations will adapt to shifts in factors like light availability, temperature, and nutrient concentrations. Furthermore, the importance of the studied organisms, make the understanding of their traits' behaviour of fundamental relevance.
To study critical behaviour, in [3] a class of models equivalent to the ones here described in the Theoretical framework section is used. In this thesis we study single-cell lineages while [3] focuses on population models leading to significant differences in the studied universality class and on the behaviour of the models.

In [6], *Escherichia coli* strains' are studied at different experimental conditions. The focus is, as in this thesis, on single-cell lineages tracked for many generations making [6] insightful for our work. In [6] the autocorrelation function (acf) is analyzed in depth in an effort to explain the observed transient oscillations, with low frequencies dominating, in the acf. A noisy linear map, linking final and initial cell size, is proposed. This cell-size control constitutes the negative feedback used to explain the presence of oscillations. The introduction of the linear map means that the process is auto-regressive with order 1 making the correlation length ξ analytically retrievable from the map's parameters. Combining simulations of cell growth and division with experimental data, the authors show how the noisy linear map is able to reproduce the observed oscillations not just in cell size but also in constitutive gene expression.

Another work that studies single-cell lineages is [19] where a negative feedback between initial cell length and added length before division is found. Furthermore, the authors propose a mathematical model of the global interactions present between global cellular variables which succeed in reproducing several aspects of the studied dataset. Oscillations in the autocorrelation function are observed as in [6] and for which a theoretical explanation is provided.

Another work which has been proven key to the development of this thesis is [23] which presents a broad investigation of the multiplicative noise problem along the lines of previous analyses of critical phenomena from a theoretical point of view. In particular, the critical properties of multiplicative-noise problems with different symmetries are looked at both accounting for fluctuations and in the mean field setting. The case study of zero spatial dimensions is similar to to our system helping in the determination of the universality class' critical exponents. Another theoretical approach which studies multiplicative noise problems is [24].

In this thesis work, where single-cell lineages are looked at, we hypothesize that the organisms under study operate close to a critical point associated to a second order active-to-absorbing phase transition. We thoroughly investigate this universality class in order to assess the behaviour of a fundamental trait i.e. size of organisms that have key biological relevance [6].
The order parameter that continuously goes from 0 to positive values and vice-versa is size at birth averaged across the lineage. The control parameter is the ratio between the growth and division time-scales of the system, which will be respectively named, in the Theoretical framework section, as ω_1 and ω_2.

We will begin this work by investigating the universality class of the system from a theoretical point of view, assuming that the process respects the Markov property for the size at birth m.
This will allow us to have some informed expectations about scaling laws and critical exponents that can be then tested further along in the thesis.

Later on, different members of the studied universality class are introduced. We start by describing a simple and analytically tractable model before increasing the complexity and the biological plausibility.
This will allow us to generate single cell lineages with custom parameters (e.g. the distance from the critical point) belonging to the assumed universality class, i.e. they exhibit characteristic scaling laws in the vicinity of the critical point [4].

In the next step we compare our results with publicly available data. As stated above, tracking single cell statistics has proven to be a challenging task through the years.
The introduction of a new microfluidic single-cell trapping device called *mother machine* allows to track cells for numbers of generations that are orders of magnitude greater with respect to previously available methods [5]. Its dead-end channels contain cells that are clones of the mother cell while the size of the colony is maintained fixed in time because the flow in the main channel flushes away cells that grow out of the dead-end channels. A more in-depth presentation of the *mother machine* is presented in the Single-cell data section.
Both [6] and [7] make use of this new technology producing and making available significant features of each lineage e.g. size at birth, size at division and growth rate. While being the most wide spread microfluidic device [20], the *mother machine* is not the only one. [37] for example makes use of the *sister machine* which allows to track two lineages from the same mother cell. The authors propose a minimal model that tries to explain evidence of correlations in size fluctuations that can persist for many generations. The premise is that the environment plays a defining role in setting the size control parameters, and that different channels within a microfluidic device are subject to different environments.

Appendix C: *Haskell* implementation describes *Haskell*'s implementation of numerical simulations. *Haskell* [21] is a statically typed, purely functional, lazy evaluated language that allows us to achieve a significant improvement in terms of performances with respect to a *Python* implementation.
Functions in *Haskell* [21] are pure functions in the mathematical sense of the term. However not to make our simulation's code overly complicated we need some form of side effects, we thus make use of the category theory's notion of *monads*: a framework for handling effects without compromising the purity of functions [8].
Monadic computation, originally introduced by Eugenio Moggi [10], not only naturally arises in our implementation due to the need of accumulating sampled sizes in a final result vector; it is also associated to working with computations that may fail i.e. the sampling process itself. This induce us to exploit the concept of the *Maybe* monad in Haskell.

2 Theoretical framework

2.1 Single-cell models

We start out by describing the dynamics of single cells as it might be observed by the *mother machine* device [5]. The fundamental trait of our discussion is the size of the cell m. The state of the cell is however described by a vector $\mathbf{x}$ containing multiple traits of the cell and having m as its first component.
The dynamics of $\mathbf{x}$ is described by a deterministic ODE,

$$\dot{\mathbf{x}}(t) \quad = \quad g(\mathbf{x}(t)) \quad , \tag{1}$$

loosely called growth process which is punctuated by stochastic division events.
Both the time-points of the division as well as the change of the cell's state upon division can be stochastic. We assume $\mathbf{x}(t)$ to be a complete description of the cell's state, such that the division time-point can be modeled in terms of a state-dependent hazard rate function, $h(\mathbf{x}(t))$, describing the cell's instantaneous probability to divide. This hazard rate function is related to the survival probability, $S(t)$, i.e. the probability not to divide until time t as,

$$\frac{d \ln S(t)}{dt} \quad = \quad -h(\mathbf{x}(t)) \quad , \tag{2}$$

a simple derivation of eq. 2 is shown in Appendix A.
Upon division, the cell draws its new state $\mathbf{x}$ from the conditional probability distribution $J(\mathbf{x}|\mathbf{x}')$ where $\mathbf{x}'$ denotes the current state. For the single-cell process, we randomly select a single daughter cell, between the two possible choices, as the mother for the next cell cycle, which is defined as the time period in between two division time-points.
Denoting the traits at birth of a cycle with the subscript i as $\mathbf{x_i}$, we can derive from the aforementioned functions $\mathbf{g}$, h and J, the transition *p.d.f.* $k(\mathbf{x}_{i+1}|\mathbf{x}_i)$. A time series $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N)$ is called a lineage.

2.2 Markov jump process

We make here two assumptions about the stochastic process described in the Single-cell models section. First, we assume the model obeys the Markov property for the size at birth m, i.e. the marginalization w.r.t. all but the first trait m_{i+1} only depends on the initial trait variable m_i, leading to a transition probability $k(m_{i+1}|m_i)$. This does not mean that there needs to be a hazard rate function that depends on m only. It only means that whichever traits the hazard rate might depend on, can be calculated from eq. 1 knowing only the initial value m_i.
Second, we assume that the transition probability k admits an equilibrium distribution $\psi_{eq}(m)$, and that the the moments $\langle m^k \rangle : k \geq 1$ are finite. We call this the homeostasis assumption.

A class of models characterized by at least two dimensionless parameters, $\gamma = \dfrac{\omega_2}{\omega_1}$ and $h = \dfrac{u}{v}$, is considered, where ω_1 and ω_2 are frequencies related to growth and division respectively and u and v are size scales related to processes limiting small and large cell sizes, respectively. Three different members of this class are presented later in this section. W.l.o.g. we set $v = 1$. First, we consider the scale-free limiting case ($h = 0$) at small sizes. For sizes m, $m' \ll 1$, we can make the scaling approximation

$$k(m'|m) \approx \frac{1}{m'} \mathcal{G}_\gamma \left(\frac{m'}{m} \right) \quad , \tag{3}$$

where the function $\mathcal{G}_\gamma(x)$ is parameterized by γ and converging to zero in both limits $x \to 0$ and $x \to \infty$ making $k(m'|m)$ normalizable.
In the scale-free case, the asymptotics of the equilibrium distribution $\psi_{eq}(m)$, as $m \to 0$, follow a power law $\psi_{eq}(m) \sim m^{\delta(\gamma)-1}$. The function $\delta(\gamma)$ is derived by the normalization condition

$$\int \mathcal{G}_\gamma \left(\frac{1}{x} \right) \cdot x^{\delta - 1} \, dx = 1 \quad . \tag{4}$$

There is a critical point, γ_c, where $\delta(\gamma_c) = 0$. It separates an active phase, where $\psi_{eq}(m) \in L^1(\mathbb{R}_+)$ from an absorbing phase where $\psi_{eq}(m) = \delta(m)$. As the order parameter, we use the average size at birth $\langle m \rangle$, along a lineage.
Generically, the phase transition is second order, with $\lim_{h\to 0} \langle m \rangle = 0$, for $\gamma > \gamma_c$, and $\lim_{h\to 0} \langle m \rangle > 0$, for $\gamma < \gamma_c$, since

$$\delta(\gamma) \propto \gamma_c - \gamma + \mathcal{O}\left((\gamma - \gamma_c)^2\right), \quad \gamma < \gamma_c \quad . \tag{5}$$

Thus, in the absence of the parameter h, which acts as a cell regulator, the order parameter continuously transitions from positive values to 0 as the control parameter γ crosses the critical point. Such a point naturally occurs, because, in the absence of trait scales, from eq. 1 in its first dimension we can write,

$$m_{i+1} = f \cdot m_i \cdot exp[\omega_1 \tau] \quad , \tag{6}$$

where the inter-division time $\tau > 0$ and possibly the division factor $0 < f < 1$ are drawn from random variables. Thus, in the absence of a process preventing too small cell sizes, the stochastic process described by k might converge to $m = 0$ from which it cannot escape.
Of course, such limiting processes do exists, however, we hypothesize that real systems might be close enough to the critical point, for the associated scaling laws to be of biological relevance. To derive the scaling of the connected correlation function, i.e. the correlation function of the fluctuations $\delta m = m - \langle m \rangle$ [12],

$$\langle m_{i+n} m_i \rangle_c = \langle (m_i - \langle m \rangle)(m_{i+n} - \langle m \rangle) \rangle \quad , \tag{7}$$

also called autocorrelation, we define the operator

$$(T\psi)(m) := \int \psi(m') k(m'|m)\, dm' \quad . \tag{8}$$

In the active phase, due to the detailed balance condition, it is a self-adjoint operator on the Hilbert space $L^2(\psi_{eq})$, with scalar product defined as $\langle \psi | \psi' \rangle = \int \psi(m)\psi'(m)\psi_{eq}(m)\, dm$. The T operator has unit norm, which follows from the Cauchy-Schwartz inequality:

$$\|T\psi\|^2 = \langle T\psi | T\psi \rangle = \int (T\psi)^2(m)\psi_{eq}(m)\, dm \le \int \psi^2(m')k(m'|m)\psi_{eq}(m) dm' dm = \|\psi\|^2 \quad . \tag{9}$$

Thus, the spectrum of T is real and norm-bounded by one. In the active phase and away from the critical point, T has an isolated eigenvalue $\lambda = 1$ corresponding to the equilibrium distribution ψ_{eq}. The correlation length is determined by the spectral gap. In the scale-free limit, eigenfunctions $f_\lambda \in L^2(\psi_{eq})$ also follow a power-law asymptotic $f_\lambda(m) \sim m^\kappa$, as $m \to 0$. Integrability at $m \to 0$ requires that $\kappa > -\delta/2$. The associated eigenvalue satisfies

$$\lambda = \int \mathcal{G}_\gamma(x) x^{\kappa-1}\, dx = \int \mathcal{G}_\gamma(1/x) x^{-\kappa-1}\, dx \quad . \tag{10}$$

The function $\lambda(\kappa)$ is convex, and $\lambda(-\delta) = \lambda(0) = 1$. Hence, for $-\delta < \kappa < 0$, we have $\lambda(\kappa) < 1$. Furthermore, using a Taylor expansion

$$\int \mathcal{G}_\gamma(1/x) x^{\delta-1} dx = 1 + a_\lambda \delta + b_\lambda \delta^2 + \mathcal{O}(\delta^3) \quad , \tag{11}$$

together with eq. 4 and eq. 10 we derive $\lambda_{max} = 1 - \mathcal{O}(\delta^2)$.
The correlation length ξ scales as $\xi \sim (1 - \lambda_{max})^{-1}$ [12], which means $\nu = 2$ following conventional notation $\xi \sim |\gamma - \gamma_c|^{-\nu}$ [12].

2.3 Observable scaling laws

In the active phase, the scaling is determined by $\delta\gamma := \gamma - \gamma_c$, as long as $h \ll \delta\gamma$. If h is too large, it will determine the scaling. In the absorbing phase, the scaling of the order parameter and the auto-correlation time is always determined by h. In both phases and for $m \ll 1$, we can make the scaling ansatz

$$\psi_{eq}(m) \approx m^{\delta(\gamma)-1}\mathcal{F}(h/m)\,, \tag{12}$$

where $\mathcal{F}(x)$ is a cutoff function decaying sufficiently fast as $x \to \infty$ and as $x \to 0$. As a consequence, with

$$\langle m^k \rangle = \frac{\int_0^\infty m^{k-\delta+1} \cdot \mathcal{F}\left(\frac{h}{m}\right)}{\int_0^\infty m^{-\delta+1} \cdot \mathcal{F}\left(\frac{h}{m}\right)} \quad , \tag{13}$$

we have that $\langle m^k \rangle \sim |\gamma - \gamma_c|\, \forall k > 0$ as similarly derived in [3] because of the pole appearing in the denominator. Hence, following conventional notation, i.e. $\langle m \rangle \approx |\gamma - \gamma_c|^\beta$ [12], $\beta = 1$.

In the absorbing phase ($\delta < 0$), the scaling of the order parameter and the autocorrelation time is determined by the parameter h leading to anomalous scaling laws [3],

$$\langle m^k \rangle \sim \begin{cases} h^k, & k < -\delta, \\ h^{\delta-1}, & k \geq -\delta \end{cases} \quad . \tag{14}$$

Furthermore, approaching criticality the return to equilibrium follows a much slower decay as perturbations decay as a power law for times much shorter than the correlation length [3], with the autocorrelation function that is thus captured by an exponential distorted by a power law as if self-similar behaviour is present time correlations should decay in a power-law fashion [4].
The power law exponent, is denoted as $2 - \eta$, where η is the correlation critical exponent, following conventional notation [12] in the case of zero spatial dimensions. For the determination of η we draw from literature [23]: for a class of models with multiplicative noise in zero dimensions, which is indeed similar to the model class we consider here, when there is a transition to an absorbing phase $2 - \eta$ evaluates to $\dfrac{1}{2}$ i.e. $\langle m_n, m_0 \rangle_c \sim n^{-\frac{1}{2}}$ for $n \ll \xi$.
However [24] reports that the only observable value of $2 - \eta$ is $2 - \eta = \dfrac{3}{2}$. One of the goals of this thesis work is to asses numerically $2 - \eta$.

Finally, unless we assume a particular model we cannot determine the distance to the critical point experimentally. However, we can combine size and temporal scaling laws to arrive at $\xi \sim \langle m \rangle^{-2}$ which, in appropriate experimental conditions, could be verified using real data.

2.4 Single-trait models: size only

We start with cells that are completely described by just one trait: their size m. A minimal model [3] requires two time scales, associated with growth and division, respectively, as well as two size-scales, associated with size-limiting processes at the upper and lower end of the size range. The simplest member of this class has growth and division rates depending linearly on the size:

$$g(m;\boldsymbol{\theta}) = \omega_1 \cdot (u + m)\,, \quad d(m;\boldsymbol{\theta}) = \omega_2\left(1 + \frac{m}{v}\right) \quad . \tag{15}$$

With $\boldsymbol{\theta}$ being the parameter vector $\boldsymbol{\theta} = (\omega_1, \omega_2, u, v)^T$ composed of two frequencies defining time-scales, i.e. $\omega_{1,2}$ and two trait scales u and v.
The model defined by eq. 15 assumes that the decay of the distribution at small cell sizes is due to a faster growth. Only to get an intuition, for $m \ll u$, $\dot{m} \approx \omega_1 u$ and thus $m(t) \approx \omega_1 ut + m_0$ pointing to linear growth. For $m \gg u$ instead, $\dot{m} \approx \omega_1 \cdot m$ leading to $m(t) \approx m_0 \cdot exp(\omega_1 t)$ and suggesting exponential growth (m_0 denotes the integration constant representing the initial size).
Hence, when the the size m becomes small, the growth behaviour switches from exponential to linear which, in the range of small times, is actually faster as simply shown graphically in Fig. 1 for $u = 0.5$, $\omega_1 = 1$ and $m_0 = 0.1 < u$ (notice how both growth equations coincide for $t = 0$).

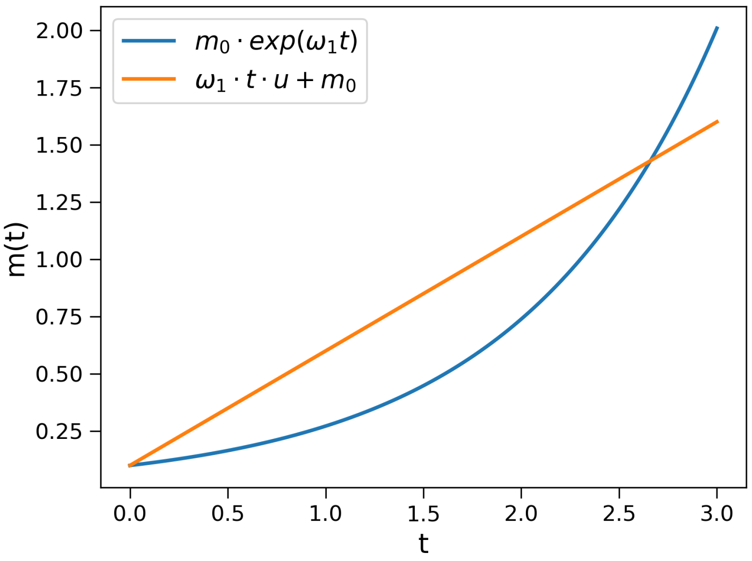


Figure 1: Linear growth is actually faster for small times with low cell size.

The decay at large sizes is due to faster division. For simplicity, we assume cells to divide into two daughter cells of equal size, i.e. we set the division factor at $f = \frac{1}{2}$ and

$$J(m_1, m_1|m) = \delta\left(m_1 - \frac{m}{2}\right) \cdot \delta\left(m_2 - \frac{m}{2}\right) \quad . \tag{16}$$

In dimensionless notation, single-cell lineages are described by the transition probability

$$k(m|m') = \frac{\gamma(1+2m)}{m + \frac{h}{2}} \left(\frac{2m+h}{m'+h}\right)^{\gamma(h-1)} \cdot exp\left[-\gamma(2m - m')\right] \chi(2m > m') \quad , \tag{17}$$

For $h = 0$, the operator T defined in eq. 8 is non-compact, and its equilibrium distribution needs to be scale-free, $\psi \sim m^{\delta-1}$, for $m \ll 1$. Setting $h = 0$ in eq. 17 we find that

$$\left(T\psi_{eq}\right)(m) = \frac{\gamma 2^{\delta}}{\gamma + \delta} \cdot \psi_{eq}(m)\left(1 + \mathcal{O}(m)\right) = \psi_{eq} \tag{18}$$

Hence, there is a critical point at $\gamma_c = (ln2)^{-1}$, and $0 < \delta = \mathcal{O}(\gamma_c - \gamma)$, for $\gamma < \gamma_c$, whereas $\psi_{eq}(m) = \delta(m)$, for $\gamma > \gamma_c$.

We present here a slightly different single-cell lineage model. As in the previous case, cell growth is here governed by cell size only. However now the lower bound u does not mark a passage between different growth behaviours but instead separates exponential growth from a region where cells cannot divide. The model is described by,

$$g(m;\boldsymbol{\theta}) = \omega_1 \cdot m \qquad d(m;\boldsymbol{\theta}) \begin{cases} 0, & m < u \\ \omega_2 \dfrac{m+v}{u+v}, & m \geq u \end{cases} \quad , \tag{19}$$

adding thus biological plausibility. We won't focus though on this model too much as our numerical simulations do not make use of eq. 19.

2.5 Two trait models: a licensing protein triggering division

Single-cell data shows that, to describe the instantaneous probability for a cell to divide, a single trait does not suffice and a second one is needed [13].
As such, an unspecified cellular protein level can be be used, which, depending on the species,

might be related to the initiation of chromosome replication or the formation of new cell membrane.
Thus, we set $\mathbf{x} = (m, p)^T$ and we assume that both m and the protein content p accumulate at a rate proportional to the mass,

$$g(\boldsymbol{x}, \boldsymbol{\theta}) = \omega_1 m \cdot \begin{pmatrix} 1 \\ c \end{pmatrix} \quad . \tag{20}$$

For the division rate, we choose a functional form equivalent to eq. 19,

$$h(p) = \begin{cases} 0 & p < u \\ \dfrac{\omega_2(p+v)}{u+v} & p \geq u \end{cases} \quad . \tag{21}$$

We assume that the protein content is depleted upon division and, as previously, the division factor is set at $f = \dfrac{1}{2}$ leading to

$$J(m_1, p_1; m_2, p_2 | m) = \delta\left(m_1 - \frac{m}{2}\right) \delta(p_1) \delta\left(m_2 - \frac{m}{2}\right) \delta(p_2) \quad . \tag{22}$$

The parameters of this model are $\theta = (\omega_1, \omega_2, u, v, c)$. W.l.o.g. we set $\omega_1 = 1$, $v = 1$ and $c = 1$ and are left again with the two dimensionless relevant parameters $\gamma = \dfrac{\omega_2}{\omega_1}$ and $h = \dfrac{u}{v}$.

3 Numerical simulations

To study the universality class of the system under study in this thesis work we rely on numerical simulations.
We thus exploit what we introduced in the previous section to sample lineages and study their behaviour approaching the critical point, keeping in mind that the scaling behaviour has two regimes: the first is controlled by the distance from the critical point $|\gamma - \gamma_c|$, the second by the control parameter h.

3.1 Sampling sizes at birth

We have mainly two methods to sample sizes at birth. The first exploits the relation introduced in eq. 2 to retrieve the survival function $S(t)$ from the hazard rate. Given $S(t)$, which is a monotone decreasing function in the closed interval $[0,1]$, we consider a uniform sample $u \sim \mathcal{U}[0,1]$ (not to be confused with the model parameter u) and we use it to find the corresponding division time τ.
This is achieved by inverting S setting thus $\tau = S^{-1}(u)$ analogously to what we would do in the inverse transform sampling method and as briefly sketched for a dummy survival function in Fig. 2.

Denoting as N the length of our simulation, we are now able to get the division times

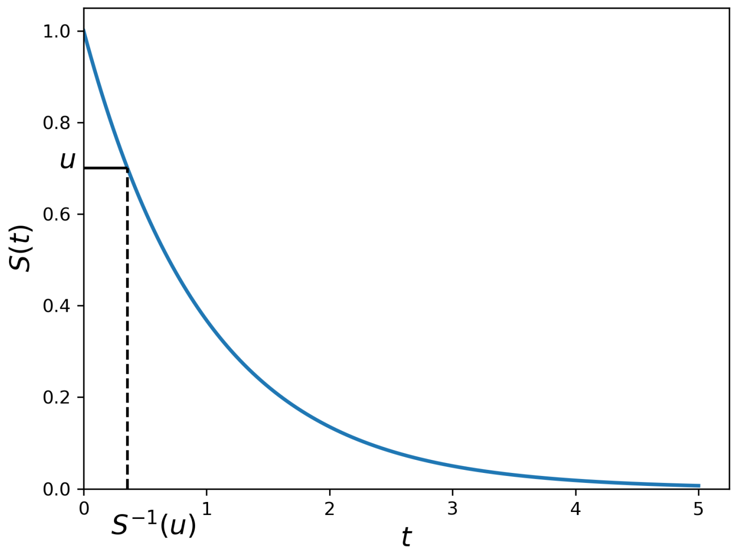


Figure 2: We use a random sample u to find the inter-division time $\tau = S^{-1}(u)$

$\tau_1 \dots \tau_N$. The generic size at birth m_i is then readily retrieved by making the system evolve between τ_{i-1} and τ_i according to its growth law and by setting $m_i = \frac{1}{2} \cdot m(\tau_i)$ according to our convention of the division factor being $f = \frac{1}{2}$.
This process allows us to sample lineages of arbitrary length N that belong to our selected model as long as we have, as previously mentioned, an analytical understanding of the survival function $S(t)$. The latter is, for the model described in eq. 15,

$$S(t) = exp\left\{\omega_2 t \cdot \left(\frac{u}{v} - 1\right) + \frac{\omega_2}{\omega_1}\left(\frac{u+m_0}{v}\right) \cdot \left(1 - e^{\omega_1 t}\right)\right\} \quad . \tag{23}$$

For eq. 20 and eq. 21 we have instead,

$$S(t) = exp\left\{-\left[\frac{m_0}{u+v}\frac{\omega_2}{\omega_1} \cdot \left(e^{\omega_1 t} - e^{\omega_1 t_0}\right) + \frac{v - m_0}{u+v} \cdot \omega_2 (t - t_0)\right] \theta(t - t_0)\right\} \quad , \tag{24}$$

with $t_0 \equiv \frac{1}{\omega_1} ln\left(1+\frac{1}{m_0}\right)$ being the minimum time at which the cell can divide and θ is the Heaviside function. The derivations of eq. 23 end 24 are shown in Appendix A.
In both cases $S(t)$ is not analytically invertible, to deal with this fact [3] introduces an approximation of the survival function valid for short times but which holds true in the setting of cells population. Here we study instead single lineages, hence to deal with S^{-1} we use numerical methods mainly through *Python's Scipy* library [14] and *Haskell's math-functions* [15].

The second method used to sample size at birth can be exploited only for the model described by eq. 15 due to the analytical understanding we have of it. In fact eq. 17 gives the transition probability that can be used to sample a new m. To be more computationally efficient though we pass to log-space by making the change of variable $m \to e^x$ (and thus $x = ln\,(m)$) obtaining for the special case $h = 0$,

$$pdf(x|x') = \gamma 2^{-\gamma} \cdot exp\left\{\gamma \cdot \left(-2e^x + e^{x'} + x_0 - x\right) + ln\,(1+2e^x)\right\} \quad , \tag{25}$$

where $x' \equiv ln\,(m')$. The few mathematical steps to arrive to eq. 25 are shown in Appendix A.
We can note that the only parameters eq. 25 depends on are h and γ. In other instances were ω_1, ω_2, u and v appear explicitly, numerical computation is performed, unless otherwise stated, by setting $v = 1$ and $\omega_1 = 1$.

Once we have at our disposal the transition probability, thanks to the *Mathematica* software [16], we compute the connected CDF that allows us to sample log-sizes by inverting it numerically.
An example of 100 sizes at birth drawn at $h = 0$ and $\gamma = \gamma_c - 1$ using the *Haskell* [21] language is shown in Fig. 3.

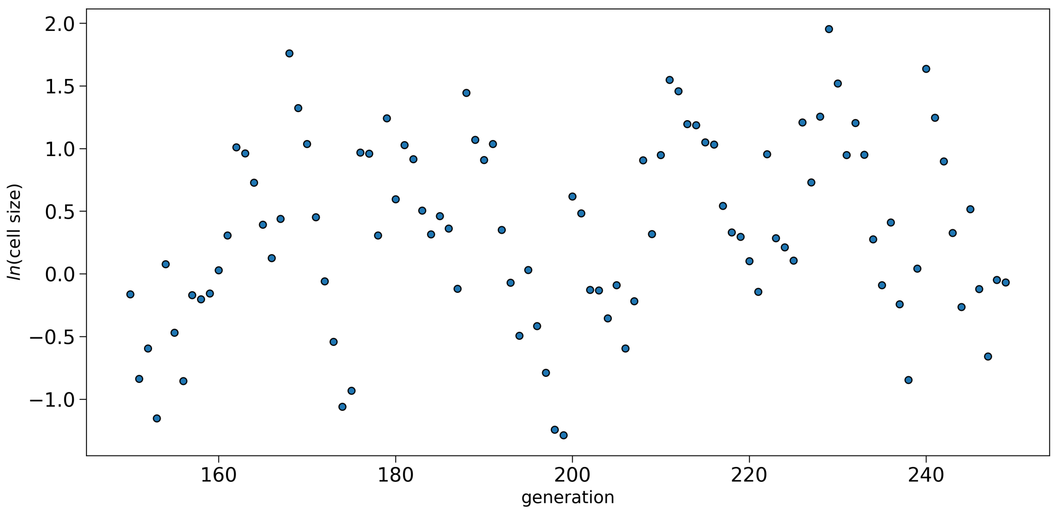


Figure 3: Example of 100 simulated generations with $h = 0$ and $\gamma = \gamma_c - 1$

3.2 Small sizes approximation

One of the main purposes of numerical simulations in this thesis is to test the system's behaviour in the vicinity of the critical point. As we will see in the Results: numerical simulations section, many critical exponents will be confirmed and tested using the simplest member of our universality class, i.e. the model described by eq. 15. We focus on the active phase with $h = 0$.
We know that for $\gamma > \gamma_c$, $\langle m \rangle = 0$, hence having a second order phase transition as $\gamma \to \gamma_{c_-}$, $\langle m \rangle \to 0$ and we will thus deal with increasingly small sizes at birth the closer we get to the critical point.
In particular, dealing with log-sizes, x will go to $-\infty$ and this can lead to numerical

problems. To invert the CDF we sample $u \sim \mathcal{U}[0,1]$ and we find the x which is the root of $0 = CDF(x|x') - u$, thanks to [14] or [15].
In both cases, the former using *Brent*'s [17] method and the latter exploiting *Ridders'* [18] algorithm, the definition of a proper interval bracketing the solution is needed. With no insight on the solution though, the bracket risks to become very large and can possibly harm both performance and feasibility of the root finding procedure as the algorithm might not converge.

It turns out though that as $x \to -\infty$ many simplifications are available as exponentials approach 0, the CDF in log-space for $h = 0$ indeed reads,

$$P_{<}(x|x') = 1 - 2^{-\gamma} \cdot exp\left\{\left(-2e^{x} + e^{x'} - x + x'\right) \cdot \gamma\right\} \quad . \tag{26}$$

The equation to numerically solve becomes thus,

$$0 = 1 - 2^{-\gamma} \cdot exp\left\{\left(-2e^{x} + e^{x'} - x + x'\right) \cdot \gamma\right\} - u \quad , \tag{27}$$

with u the uniform sample and this reads

$$\begin{aligned} (1-u) \cdot 2^{\gamma} &= \exp\left\{\left(-2e^{x} + e^{x'} - x + x'\right)\gamma\right\} && \text{Separating the exponential from the rest.} \\ x + 2e^{x} &= -\frac{1}{\gamma} \cdot ln\ (2^{\gamma} \cdot (1-u)) + x' && \text{Taking the logarithm on both sides and assuming } e^{x'} \approx 0. \end{aligned} \tag{28}$$

That, solved graphically, looks like Fig. 4.

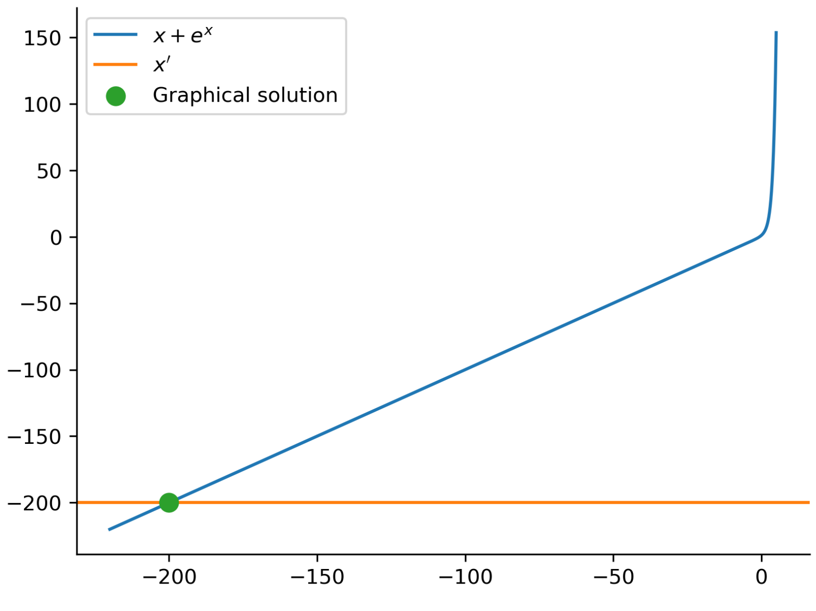


Figure 4: Graphical solution of eq. 28

Where we assumed $(1-u) \equiv q = \frac{1}{e}$, $x' = -200$ and that we are at criticality i.e. $\gamma = \frac{1}{ln\,2}$. So, unless $-\frac{1}{\gamma} \cdot \ln q$ is high enough[1] to balance x', $x + e^{x}$ lies in its linear part and we can

[1]e.g. to have a value $\frac{1}{\gamma} \cdot ln\,q > 10$ (that still far lower than the x' in the example) we need $q < e^{-10\gamma}$, that happens with probability $< e^{-10\gamma}$

approximate $x + 2e^x \approx x$.
We are left with the equation,

$$x = -ln\,2 - \frac{1}{\gamma} \cdot ln\,q + x' \quad . \tag{29}$$

If we now call x_i the $i - th$ size at birth of a certain lineage, by recursion we find,

$$x_n = \underbrace{-ln\,2 \cdots - ln\,2}_{n \text{ times}} - \frac{1}{\gamma} \cdot ln\,q_0 - \cdots - \frac{1}{\gamma} \cdot ln\,q_n + x_0 \quad ;$$

$$x_n = -n \cdot ln\,2 - \frac{1}{\gamma} \left(\sum_{i=1}^{n} ln\,q_i \right) + x_0 \quad ; \tag{30}$$

$$x_n = -n \cdot ln\,2 - \frac{1}{\gamma} \cdot ln \left(\prod_{i=1}^{n} q_i \right) + x_0 \quad .$$

With this we achieve two things:

1. We confirm that above the critical point, x_n diverges to $-\infty$, as expected, whereas above it, it would go to $+\infty$ if it weren't for the neglected non linear terms.
   Furthermore, close to the critical point, those divergences are random-walk-like, i.e. scale like $\sqrt{n}$. Both conclusions derive from the central limit theorem which tells us that $\frac{1}{n} \sum q_i$ converges toward a normal distribution centered at -1 and with a standard deviation of $\sigma = \sqrt{n}$.

2. We get either a very good approximation of x which helps us define a proper and more efficient bracket or directly a method to sample x given x' that can be "activated" whenever x' is low enough by setting a threshold.

For completeness we mention that, calling $Q \equiv \prod_{i=1}^{n} q_i$ in eq. 30, being Q the product of n i.i.d. uniform random numbers, Q is distributed like,

$$pdf\,(Q) = \begin{cases} \dfrac{(-ln\,(z))^{n-1}}{(n-1)!} & 0 < z \leq 1 \\ \\ 0 & \text{otherwise} \end{cases} \tag{31}$$

4 Single-cell data

Lineages used in this thesis work refer to previously published data [6, 19]. Both data sets are collected thanks to a *mother machine* device that allows to track cells in a regime of steady-state growth, which is necessary for reproducible quantitative studies [20]. Such measurements provide ample new information about the processes studied here [19].
For this reason and for the high number of generations a lineage can be observed for, the *mother machine* has attracted much interest for its potential studies that comprehend but are not limited to bacterial physiology, cell mechanics and cell aging [20, 5].
This kind of design consists of a series of growth channels, oriented at right angles to a trench through which growth medium is passed at a constant rate. The cell at the end of the growth channel, distal to the trench, is referred to as the "old-pole mother cell" (or mother cell) because one of its poles, abutting the end of the channel, is inherited from one generation to the next. The diameter of the growth channels prevents the mother cell from moving around [5].
All the cells in each channel are clones of the mother cell that resides in the dead-end side of the channel. Furthermore, the colony has a fixed size through time because flow in the main channel flushes away extra cells that grow out from the dead-end channels. The same flow also maintains a constant media environment in the growth channels by replenishing nutrients and removing metabolic waste products [20].
An illustration of the *mother machine* device, taken from [7] is showed in Fig. 5,
Let's now describe a little more in depth the two data sets at our disposal. From now on

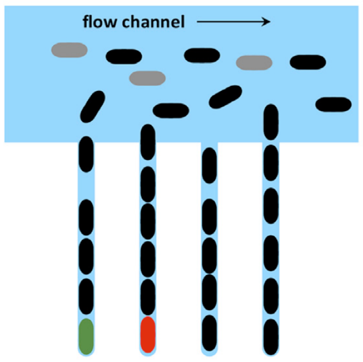


Figure 5: Graphical sketch of a *mother machine* device [7].

we will refer to the data available from [19] as *long lineages* set and the one published by [6] as *short lineages* set, the name choice will be made clear in the following.
The data is formatted in a *csv* file containing multiple columns which describe the status of the cell at each generation, the minimal set of entries used in this thesis are summed up in tab. 1.

length_birth	**length_final**	**lineage_ID**	**generation**
Length at birth of the studied cell	Length at division of the studied cell	Unique identifier of the lineage	Unique identifier of the generation within the lineage

Table 1: Minimal set of parameters used from each data set.

The *long lineages* set is composed of 20 different lineages with a maximum length of 250 generations and a minimum of 47, an example, where an exponential is fit from the initial to the final cell length, is shown in Fig. 6 for the first 50 generations.
The *short lineages* one is instead sub-divided in two different data sets obtained at different experimental conditions (25^oC and 37^oC respectively). The variance of the generation

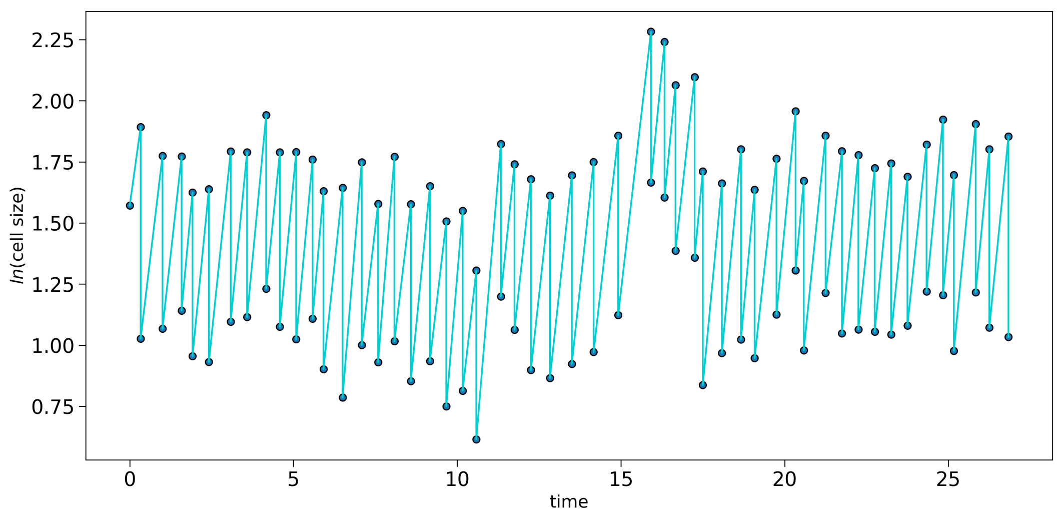


Figure 6: Example of the first 50 generations of a lineage from [19].

length is in this case much lower as the lineages have either 69 or 70 samples. The 25^oC case contains 65 lineages whether in the 37^oC set we can find 160 lineages.
Tab. 2 summarizes the experimental data's features.

	***Long-lineages* set**	***Short-lineages* set**	
		25°C	**37°C**
Number of lineages	20	65	160
Min. length of lineages	47	69	69
Max. length of lineages	250	70	70

Table 2: Main features of each dataset considered in this thesis.

4.1 Outliers

To identify and eliminate outliers in experimental data we take the same approach of [6]. Cells indeed occasionally undergo aberrant cell growth such as filamentation, to recognize such instances we check for the following:

1. initial cell length is larger then $\bar{L} + 2\sigma_L$,
2. final cell length is larger than $2 \cdot (\bar{L} + 2\sigma_L)$,
3. initial cell length is smaller then $\bar{L} - 2\sigma_L$.

Where $\bar{L}$ and σ_L are the average and standard deviation of the cell size distribution, respectively. If a data point does belong to the 3 categories above the latter part of the lineage (the one after the outlier) gets discarded from the analysis. Only a very small percentage of data ($< 1\%$) is affected by this procedure.

5 Results: numerical simulations

5.1 Autocorrelation's oscillations

In [6] it was reported that the autocorelation of the tested lineages, belonging to what we called here the *short lineages* set, displayed strong oscillations that canceled out upon averaging.
We see the same behaviour in the lineages simulated with our models, as seen in Fig. 7 for a lineage of 250 samples drawn according to the growth law described in 15 at $h = 0$ and $|\gamma - \gamma_c| = 1$.

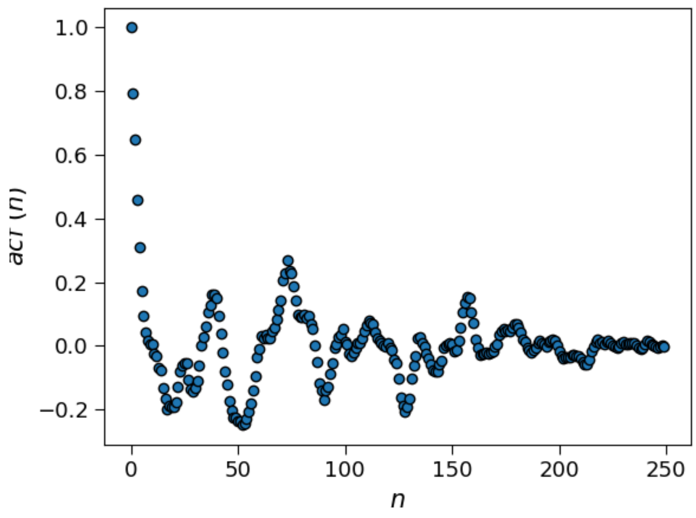


Figure 7: Autocorrelation function for a sampled lineage according to 15 having 250 generations. The parameter of the model are set to $h = 0$ and $|\gamma - \gamma_c| = 1$.

The strong oscillations may harm, if not averaged out, the quality of the different fits performed on the autocorrelation and thus prevent a correct determination of the various scalings. The fact that oscillations do not survive the average also excludes the presence of an underlying periodic component of the autocorrelation (acf), ruling out building and fitting an envelope containing the maxima of the acf.
Recent literature [6] proposes a noisy linear map, based on the linear relationship observed between initial and final length of a generation, of the form

$$L_i(n+1) = (aL_i(n) + b + \epsilon) \quad , \tag{32}$$

where $L_i(n)$ is the initial size at generation n and ϵ is Gaussian white noise whether a and b are the linear coefficients.
Eq. 32 describes an auto regressive model of order 1, whose autocorrelation function yields $acf(n) = a^n$ [22] with n representing the lag. This latter relation would allow us to estimate the autocorrelation length ξ as $\xi = -\frac{1}{ln\, a}$, obtained setting $a^n = e^{-\frac{n}{\xi}}$. Making use of the linear map proves to be way too noisy for a reliable estimation of the autocorrelation length, especially when looking for a scaling behaviour, on real data. Furthermore, when talking about our simulations, the long nature of our sampled lineages act as a way of averaging at small lags, as shown in Fig. 8, representing the autocorrelation of the last 1 million samples of an 8 million generations lineage at $h = 0$ and $|\gamma - \gamma_c| = 0.1$, 1000 lags are shown for visual clarity. We are thus allowed to traditionally fit the acf (with a pure exponential or with an exponential distorted by a power law depending on the distance from the critical point) and we will not assume any underlying linear auto regressive relation.

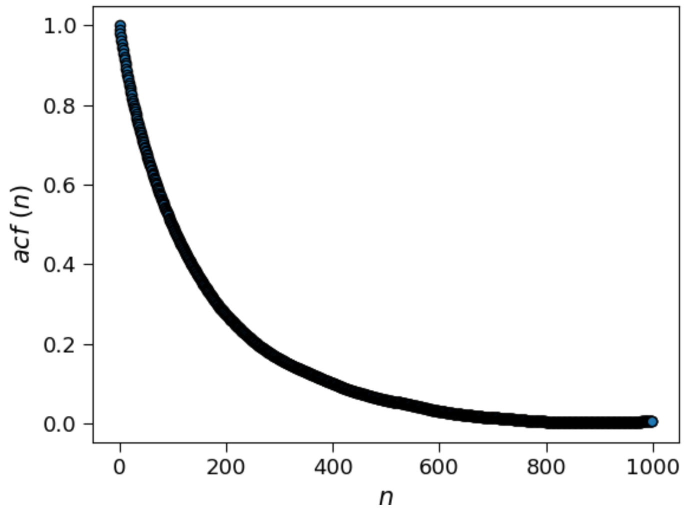


Figure 8: 1000 lags of the autocorrelation function for a lineage sampled according to eq. 15 at $h = 0$ and $|\gamma - \gamma_c| = 0.1$. The *acf* is computed for the last 1 million generations of an 8 million simulated lineage.

5.2 β critical exponent

In order to test whether β assumes the expected value of 1 we simulate 100 lineages at $h = 0$. Trying to go sufficiently close to the critical point we choose $|\gamma - \gamma_c|$ as 100 equally spaced (in *log-space*) points going from $|\gamma - \gamma_c| = 10^{-1}$ to $|\gamma - \gamma_c| = 10^{-6}$. Being at $h = 0$ we are forced to place ourself at $\gamma < \gamma_c$ i.e. in the active phase.
Going closer to the critical point comes with an added complication: the return to equilibrium follows a much slower decay as perturbations follow a power law for times much shorter than the correlation length [3], this fact requires us to simulate longer lineages in order to arrive at stationarity.
We thus generate 8 million samples per lineage, with only the latter portion (1 million samples) taken for the analysis.
Fig. 9 shows how the first 3 moments of the initial size at birth scale with the distance to the critical point, in linear scale and *log-log* respectively. Other moments are shown in Appendix B.

Both a visual inspection for the linear scale plots and the comparison with the drawn red line, representing a line with slope 1, confirm the expected value of $\beta = 1$. Furthermore, linear behaviour is also noticeable for higher moments.
Looking at the figures in *log-log* scale, we see how the quality of the linear scaling progressively worsens as we approach the critical point. We hypothesize that this effect is due to the aforementioned critical slowing down and longer lineages would lead to clearer scaling also closer to the critical point.
We tested this assumption by simulating shorter lineages, whose scaling indeed worsens further from γ_c with respect to the the ones in Fig. 9 confirming the finite-size effect.

5.3 η critical exponent

As outlined in the Observable scaling laws section, numerical simulations can clarify the nature of the power law exponent $2 - \eta$ that characterizes the autocorrelation function when self-similar behaviour arises.
The lineages that will help us give a more informed assessment about $2 - \eta$ are the same used for the β exponent 5.2 i.e. we are at $h = 0$ with $|\gamma - \gamma_c|$ ranging from 10^{-1} to 10^{-6} approaching γ_c from the active phase.
We estimate the autocorrelation function with the *statsmodels Python* library [25], while

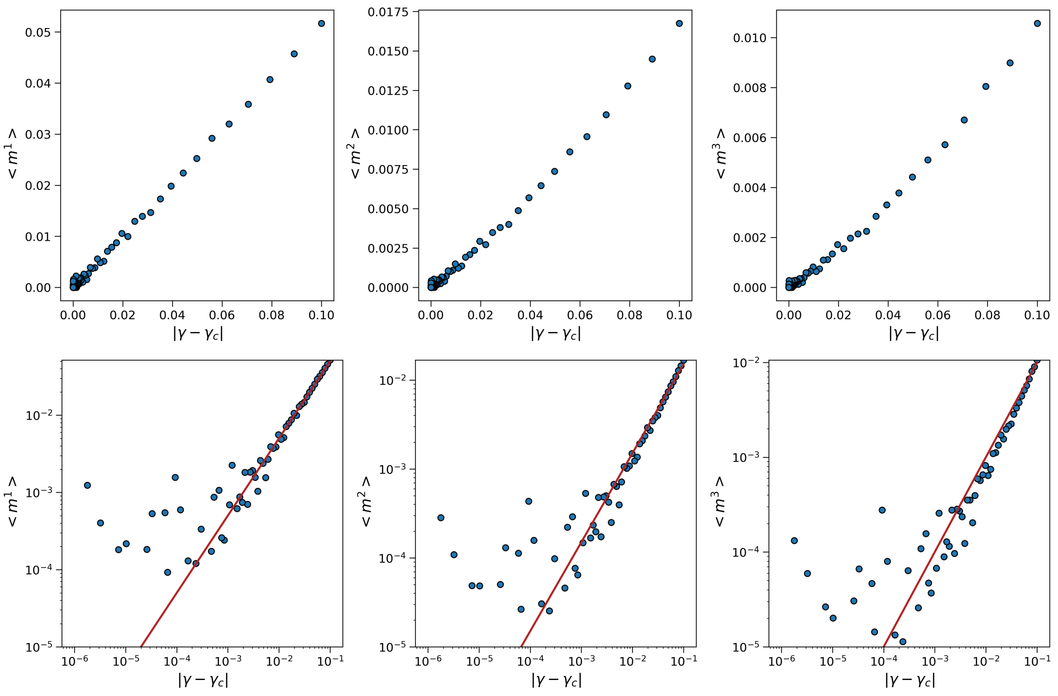


Figure 9: Scaling of the first 3 moments of the size at birth for lineages of 8 million generations simulated at $h = 0$ and with $|\gamma - \gamma_c|$ transitioning from 10^{-1} to 10^{-6}. Linear scale (top) and *log-log* scale (bottom). The red line is a line with slope 1 drawn as a guide for the eye.

the fit of $acf(n)$ with the exponential distorted by a power law is performed using *scipy*'s *curve_fit* function [14] which exploits the *least-squares* algorithm [26] to find both the optimal parameters and the their covariance matrix.
Fig. 10 sums up the results, the *x-axis* is plotted in *log* scale as $|\gamma - \gamma_c|$ is equally spaced in *log-space*. As we approach the critical point $2 - \eta$ seems to be converging to $\frac{1}{2}$, confirming

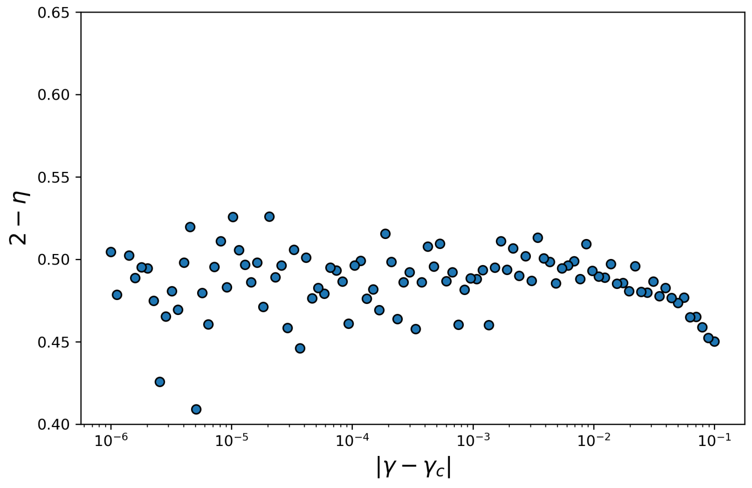


Figure 10: Autocorrelation's power law critical exponent approaching γ_c, the *x-axis* is plotted in *log* scale for visual clarity as $|\gamma - \gamma_c|$ points are equally spaced in *log-space*. The behaviour of $2 - \eta$ confirms the result stated in [23].

thus what was stated in [23].

5.4 Correlation length's scaling

Using the lineages analyzed and described in the previous two sections we try to assess numerically the correlation length critical exponent i.e. we look for ν in $\xi = |\gamma - \gamma_c|^{-\nu}$ [12]. Inferring ξ fitting the studied lineages as already described when assessing η's value, we obtain Fig. 11.

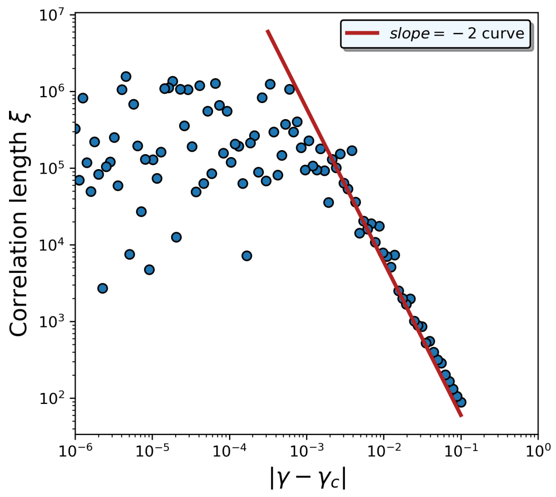


Figure 11: Correlation length approaching the critical point. ξ diverges as expected. Furthermore $\nu = 2$ seems confirmed by numerical simulations. The same *flattening* effect noticed for the β exponent's derivation is noticeable.
The red line (right) represents a line with slope -2 drawn as a guide for the eye.

We can notice how the same effect, attributed to finite-size arises the closer we get to γ_c, analogously to what happened in Fig. 9. Also in this case other experiments, performed with shorter lineages, saw the *flattening* effect appear further from the critical hinting at a finite-size problem.
Fig 11 clearly shows that ξ is diverging as $\gamma \to \gamma_c$, as expected for critical systems. ξ remains also below the system's size by which it's bounded [12].

We also probe whether ξ scales quadratically with the inverse of the average size at birth, i.e. $\xi \sim \langle m \rangle^{-2}$ (we refer to this critical exponent as ρ) as stated in the Observable scaling laws section. Fig. 12 which draws ξ and $\langle m \rangle$ in *log-log* scale, answers positively the question.

If both $\xi \sim |\gamma - \gamma_c|^{-2}$ and $\xi \sim \langle m \rangle^{-2}$ hold then we should also see that $\langle m \rangle \sim |\gamma - \gamma_c|$. This latter relation, already confirmed in Fig.9, does not depend on any fit of the autocorrelation function connecting instead one manually set variable (the distance to critical point) to a statistic of the lineage that can be easily retrieved (the mean).
Tab. 3 summarizes the values of the critical exponents we looked at thanks to numerical simulations in this thesis.

ν	$2-\eta$	β	ρ
2	1/2	1	2

Table 3: Critical exponents of the system studied in this thesis work.

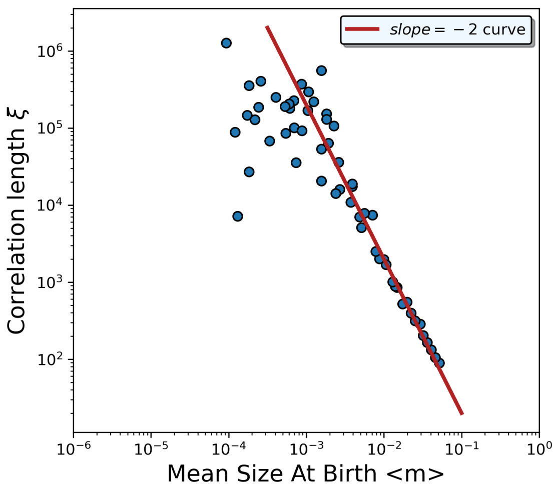


Figure 12: Correlation length plotted against average size at birth. The expected quadratic behaviour is confirmed. The red line represents a line with slope 2 drawn as a guide for the eye.

6 Results: experimental data

6.1 Correlation length

We described experimental data in the Single-cell data section and we already stated how, without assuming a specific model, we cannot determine the distance from the critical point. This greatly reduces our possibilities to confirm the different scalings probed in the previous sections. The main one analyzed in the Results: numerical simulations section is the relation $\xi \sim \langle m \rangle^{-2}$. Moment scaling is also something we can compare and that we will look at.
We encounter though 2 main problems in verifying $\xi \sim \langle m \rangle^{-2}$. First of all, up to now we have benefited from the fact that we could make lineages very long e.g. numerical results are presented for 1 million samples (the last million out of the 8 total). This has the effect of averaging out oscillations making the fit of the autocorrelation function much more reliable. When dealing with experimental data though, oscillations are indeed present and this can blur the scaling effect.
To probe this finite-size shortcoming of experimental data we reproduce Fig. 12 using the same generated lineages used for Fig. 12 but instead of taking the last million generations we pick the last 250 i.e. the typical length of our *long-lineages* set. The new plot in *log-log* scale is shown in Fig. 13

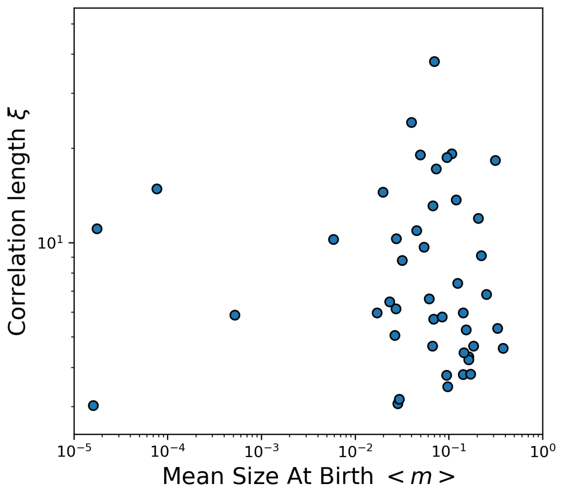


Figure 13: Using the same lineages that showed a nice $\xi \sim \langle m \rangle^{-2}$ scaling when taking the last million samples, we instead see no clear behaviour arising considering 250 samples.

On top of oscillations being present, we also expect real observations to be noisier than simulations making the ξ-$\langle m \rangle$ relation even more difficult to see.
The second main point lies on the fact that the data we use here exhibit a very limited range as far as average size at birth $\langle m \rangle$, this naturally prevents the signal, even if present, to arise from noise. Fig. 14 takes into account all the data at our disposal and confirms what we discussed above and what we saw considering a few samples of simulated lineages, i.e. the $\xi \sim \langle m \rangle^{-2}$ behaviour is hidden.

6.2 Comparison with experimental data: moments' scaling

We now focus on moment scaling away from the critical point. Specifically, we focus on $k = 2$ and $k = 3$.

Fig. 15 shows moment scaling for $k = 2$ and $k = 3$. We can immediately notice how results are noisy (much noisier than simulations). Furthermore, in all 3 cases the mean sizes at birth $\langle m \rangle$ span a small segment. This feature of the *x-axis* having a very limited range makes it hard to fit $\langle m^k \rangle$. We can for example see how, visually (Fig.15), the data seems linearly distributed while a *log-log* plot (Fig. 16) tells us otherwise.

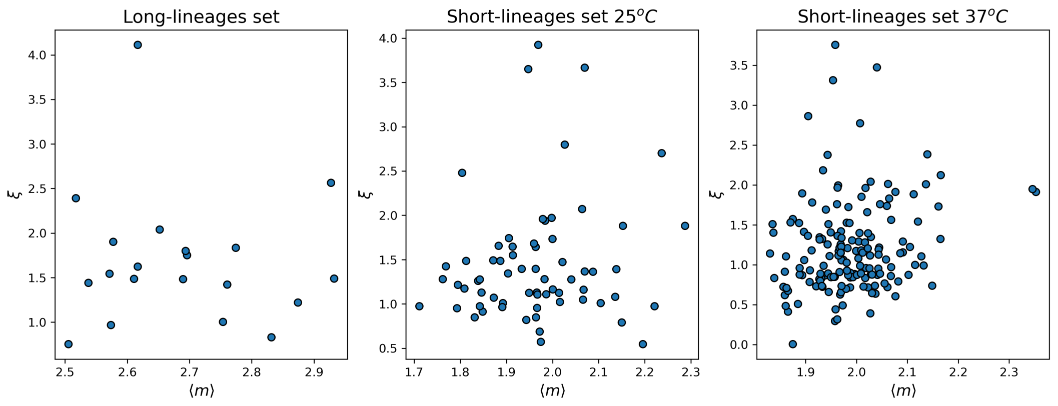


Figure 14: Correlation length (y-*axis*) plotted against the average size at birth (x-*axis*) for each experimental data set used in this thesis work. No clear scaling is observable connecting the correlation length ξ to the average size at birth $\langle m \rangle$ in experimental data.

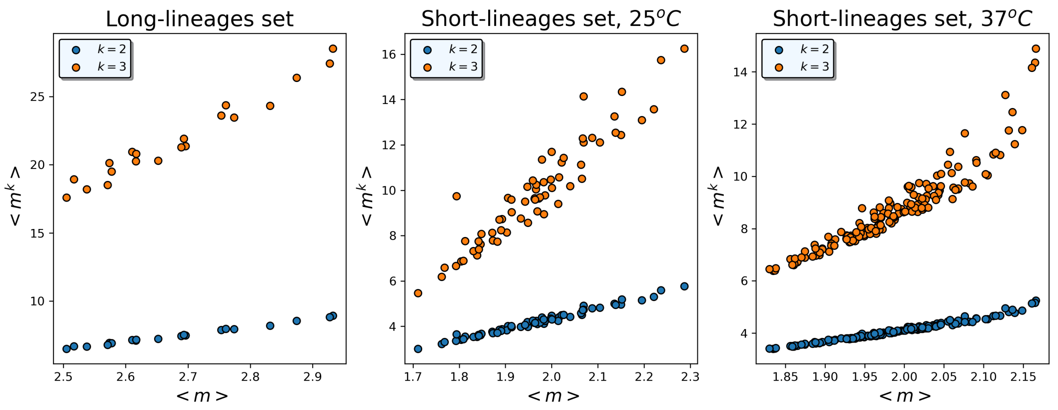


Figure 15: Moment's scaling of experimental data. Although the scatter points seem to lie on a linear curve, we are limited by the $\langle m \rangle$ range that hides power law behaviour (we are looking at the tangent) as a linear fit in *log-log* yells exponents different from 1. The x-*axis*' short range also harms a reliable assessment of the power law exponent.

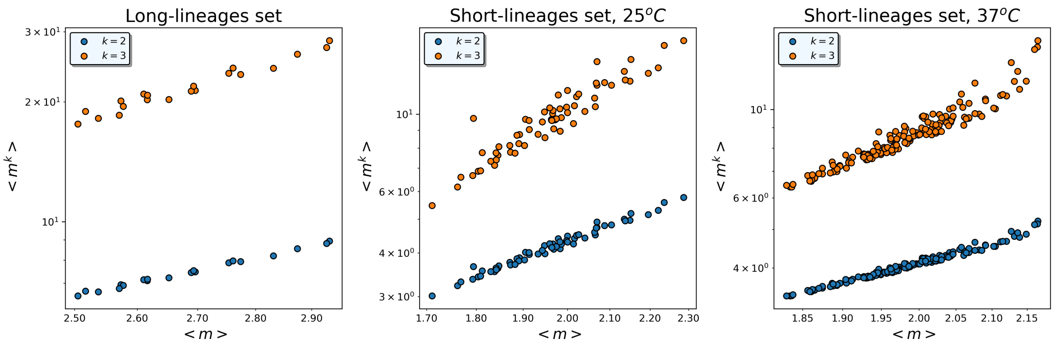


Figure 16: Moment's scaling of experimental data in *log-log* scale.

The results of linear plots in log-space, measuring the power-law exponents, made via the *Scipy* [14] library are shown in Tab. 4. The error of each result is computed taking the square root of the covariance matrix's diagonal term corresponding to the exponent. Rows are indexed with the keyword *slope* as the slope in a *log-log* plot correspond to the variable we are looking for i.e. the power law exponent. We notice, for the *long-lineages* set, the

	Long-lineages set	**Short-lineages set**	
		25°C	**37°C**
slope, $k = 2$	1.94 ± 0.04	2.17 ± 0.04	2.30 ± 0.03
slope, $k = 3$	2.8 ± 0.1	3.5 ± 0.2	4.0 ± 0.1

Table 4: Fitted power law exponent for experimental data's moment scaling.

values being below trivial moment scaling, especially for $k = 3$. The *short-lineages* set give back values above 2 and 3 for $k = 2$ and $k = 3$ respectively. We find this result oddly looking and we hypothesize that this deviation from the expected behaviour is caused by the poor quality of the data (both noise and short *x-range*).
To probe how conclusive the fit is, we proceed by generating a heatmap hoping to get more insight: we sample slopes and intercepts uniformly from an interval wrapping the fitted slope and intercept respectively. Then, we compute the mean squared error between the points represented in Fig. 15 and the line having the sampled slope and intercept.

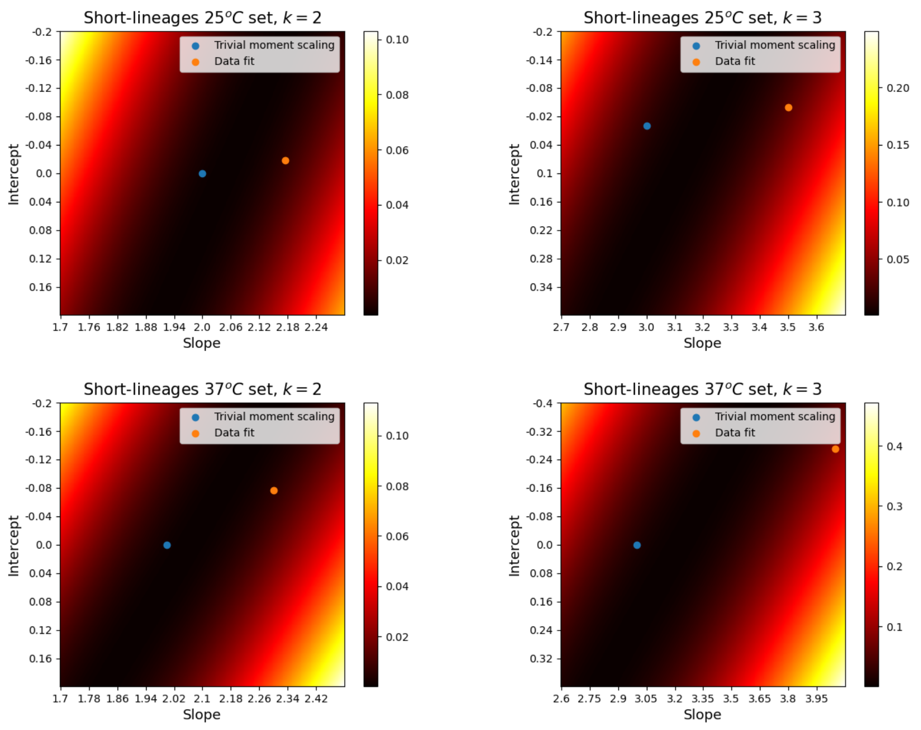


Figure 17: Heatmaps representing the mean squared error (*MSE*) (in the colorbar) for different slopes and intercepts. The blue point marks trivial moment scaling.

Fig. 17 shows how lower values of the power law values w.r.t tab. 4 are compatible with the Mean Squared Error *MSE* (represented by the color and visible in the colorbar) landscape. We have in fact that the valley representing low *MSE* wraps around an x value of 2 and 3 for plots on the left and on the right of Fig. 17 respectively. The observed behaviour of the *MSE* landscape w.r.t. to the fitted exponents is probably related to the landscape being very flat around the minimum.

We further study scaling by getting back to simulations. This time we want to closely emulate experimental data and we do not need to approach the critical point to find any critical exponent. The question thus lie on how to properly set the h and γ values.
Only guessing the h and γ values won't probably suffice though in this case. To answer this problem we rely on Bayesian Inference for which we use *Python*'s *emcee* [27] library which exploites the *Montecarlo-Hastings* algorithm, a brief introduction to Bayesian Inference and its application to our case is in Appendix B. We run the inference using the model described

in eq. 20 and eq. 21 for every lineage at our disposal. This way we get, for each lineage, the posterior for every parameter of the model whose inferred value is estimated taking the maximum of the posterior.
Examples of posterior estimates can be found in Appendix B while results of the inference process is summed up in Fig. 18 where h values (top row) and γ values (bottom row) are showed for each dataset as a function of the average size at birth $\langle m \rangle$, the critical point is also shown as a purple horizontal line. We can notice that the inferred values of γ confirm our initial hypothesis that lineages operate in the proximity of the critical point γ_c.

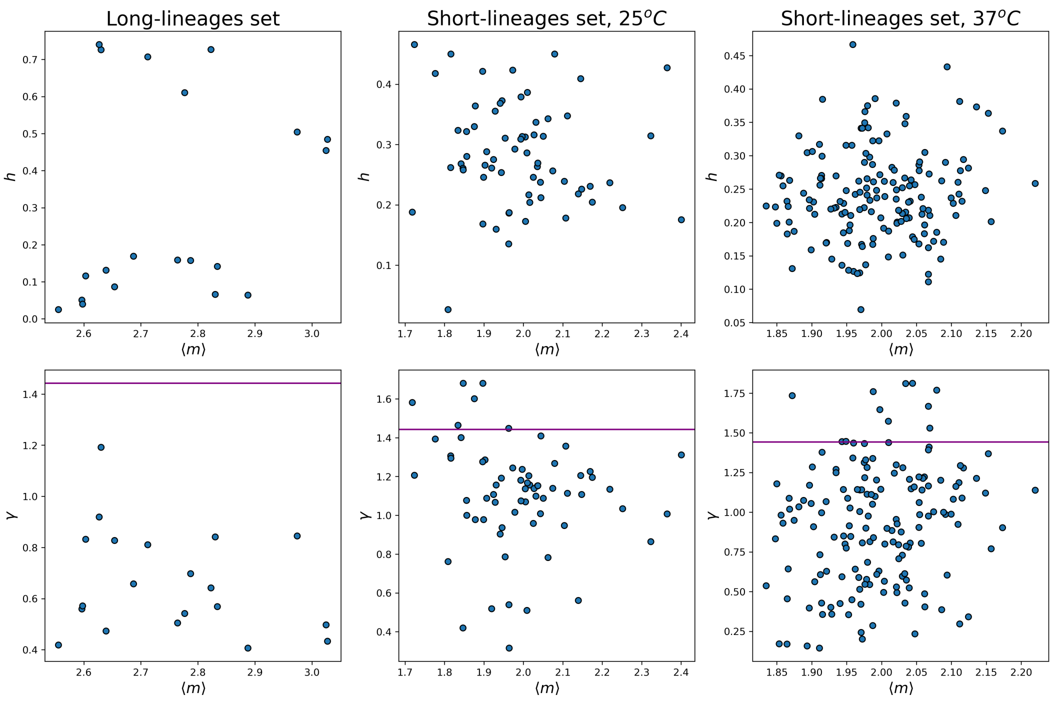


Figure 18: h values (top row) and γ values (bottom row) infered for each lineage of each dataset. Data is shown as a function of the average size at birth. The purple line (bottom row) represents the critical point γ_c.

With the parameters of our two trait model now set, we can proceed with the new simulations.

The next step is that of repeating the analysis that lead to tab. 4. In the *Short-lineages* set case, the power law exponent is now below 2 and 3 for $k = 2$ and $k = 3$ respectively as showed in tab. 5 while it's interesting to notice how we get very similar results w.r.t. tab. 4 for the *long-lineages* set.

	Long-lineages set	**Short-lineages set**	
		25°C	**37°C**
slope, $k = 2$	1.90 ± 0.02	1.971 ± 0.008	1.970 ± 0.003
slope, $k = 3$	2.74 ± 0.05	2.90 ± 0.02	2.901 ± 0.007

Table 5: Fitted power law exponent for lineages simulated using the two trait model with h and γ vales found with Bayesian Inference as in Fig. 18.

7 Conclusions

In this thesis work we have studied single-cell lineages, under the hypothesis that the system operates near a critical point associated with a second order active-to-absorbing phase transition.
We used the size at birth averaged across a lineage ($\langle m \rangle$) as the order parameter while the control parameter is γ i.e. the ratio between the growth and division time-scale of the system.

In the Theoretical framework section we have introduced the universality class of the system, setting up the various critical exponents to be later verified.
Unless we assumed a particular model though, we could not determine the distance from the critical point. We thus introduced 3 different models belonging to the same universality class in the Theoretical framework section. Each model adds biological plausibility to the previous one but all share 4 key parameters i.e. the trait scales u and v regulating too small and too large sizes respectively, and the time scales ω_1 and ω_2 representing the growth factor and the division rate, respectively.

We then probed the critical exponents thanks to numerical simulations performed by sampling lineages according to the aforementioned models. We succeed in confirming multiple exponents and we also clarifyed how the autocorrelation one $(2 - \eta)$ approaches the critical point as in literature 2 different results can be found.

Simulations were coded with *Haskell* [21], a purely functional programming language, using monadic computations [10] which allows programming with effect in a non effectful environment [8]. The Appendix C: *Haskell* implementation section contains the description of the code implemented and the performance improvements w.r.t. a plain *Python* implementation.

In Single-cell data we described experimental data used in this thesis. Both datasets, which we called *long-lineage* set and *short-lineage* set are collected thanks to a *mother machine*: a device that allows to track cells for many generations.
The *long-lineages* set contained 20 lineages of size ranging from 49 to 250. The *short-lineages* one is instead further divided in two sets collected at different experimental conditions having respectively 65 lineages with sizes ranging from 69 to 70 and 160 lineages with sizes ranging from 69 to 70.

In the last step of our analysis we looked at experimental data and its scaling. Due to the small range of the observed average sizes at birth, relatively short lineages not allowing to average out oscillations and due to noise in the data, no scaling relation between ξ and $\langle m \rangle$ could be observed.

In this thesis we find evidence for criticality in two ways:

- Via direct observation of anomalous moment scaling in experimental data.
- Via Bayesian Inference, used to infer the control parameter for a particular member of the universality class confirming the initial hypothesis of proximity to the critical point. Appendix B presents the application of Bayesian Inference in this thesis.

8 Appendix A

8.1 Hazard rate and survival probability relation

Here we want to show a simple derivation of eq. 2 that links the probability not to divide until time t, i.e. $S(t)$ and the hazard rate $h(\mathbf{x}(\mathbf{t}))$.

First of all, let's set a little bit of notation: we will denote as T the random variable representing the survival time i.e. the time-to-event.
The probability that the event did occur by time t, i.e. the cumulative density function will be marked as $CDF(t)$, $CDF(t) = 1 - S(t)$.
We can then rewrite the hazard rate using its limit definition,

$$h(t) = \lim_{\Delta t \to 0} \left(\frac{P(t < T \leq t + \Delta t | T > t)}{\Delta t} \right) \quad , \tag{33}$$

With the $|$ sign denoting conditional probability. The cumulative hazard rate function $H(t)$ is instead,

$$H(t) = \int h(t) \cdot dt \quad . \tag{34}$$

Eq. 33 can be reworked using the product rule, i.e. $P(AB) = P(A|B) \cdot P(B) = P(B|A) \cdot P(A)$, and the relation,

$$P(t < T \leq t + \Delta t) \wedge P(T > t) = P(t < T \leq t + \Delta t) \quad , \tag{35}$$

as:

$$h(t) = \lim_{\Delta t \to 0} \left(\frac{P(t < T \leq t + \Delta t) \wedge P(T > t)}{P(T > t) \cdot \Delta t} \right)$$
Using the product rule.

$$h(t) = \lim_{\Delta t \to 0} \left(\frac{P(t < T \leq t + \Delta t)}{S(t) \cdot \Delta t} \right)$$
Exploiting eq. 35 and the fact that $P(T > t) = S(t)$.

$$h(t) = \lim_{\Delta t \to 0} \left(\frac{P(T \leq t + \Delta t) - P(T \leq t)}{S(t) \cdot \Delta t} \right)$$
Refactoring the numerator.

$$h(t) = \lim_{\Delta t \to 0} \left(\frac{CDF(t + \Delta t) - CDF(t)}{\Delta t} \right) \cdot \frac{1}{S(t)} \tag{36}$$
Exploiting CDF's definition and bringing $S(t)$ out of the parenthesis.

$$h(t) = \frac{dCDF(t)}{dt} \cdot \frac{1}{S(t)}$$
Using the derivative definition.

$$h(t) = \frac{d(1 - S(t))}{dt} \cdot \frac{1}{S(t)}$$
Using survival function's definition.

$$h(t) = -\frac{dS(t))}{dt} \cdot \frac{1}{S(t)} \tag{37}$$
Computing the derivative.

Finally we notice that for a general function $f(x)$, using the chain rule we find $\frac{d}{dx} \ln(f(x)) = \frac{d}{dx} f(x) \cdot \frac{1}{f(x)}$ and thus we can rewrite eq. 37 as $h(t) = -\frac{d}{dt} \ln S(t)$ that is indeed equation 2.

8.2 Survival derivation

Here we derive eq. 23 and eq. 24 finding $S(t)$ needed to retrieve the inter-division times of a lineage beginning with the former.
Let's start in the single trait case. First of all we need the expression of the growth rate, we notice that the *ODE* in the left of eq. 15 is integrable by separation of variables, yielding

$$m(t) = (m_0 + u) \cdot e^{\omega_1 t} - u \quad , \tag{38}$$

in which we have used the substitution $m(t = 0) = m_0$ to find the integration constant.
Then,

$$\begin{gathered} \frac{d\, lnS(t)}{dt} = -\omega_2 \cdot \left(1 + \frac{m(t)}{v}\right) \quad ; \\ \frac{d\, lnS(t)}{dt} = -\omega_2 \cdot \left(1 + \frac{e^{\omega_1 t} \cdot (u + m_0) - u}{v}\right) \quad , \end{gathered} \tag{39}$$

separating the variables and integrating we get,

$$lnS(t) = -\omega_2 t \left(\frac{u}{v} - 1\right) - \frac{\omega_2}{\omega_1} \cdot \frac{(u + m_0)}{v} \cdot e^{\omega_1 t} + c \quad . \tag{40}$$

With c being the integration constant, found by imposing the initial condition $S(t = 0) = 1$ i.e. at time 0 no cell has encountered a division event, leading to eq. 23.

We proceed analogously for the two trait model finding,

$$m(t) = m_0 \cdot e^{\omega_1 t}; \quad p(t) = m_b \cdot (e^{\omega_1 t} - 1) \quad . \tag{41}$$

The time t_0 at which the cell can start to divide, is retrieved by imposing $p(t) = u$, indeed,

$$m_0 \cdot \left(e^{\omega_1 t_0} - 1\right) = u \quad . \tag{42}$$

Taking the logarithm on both sides we find,

$$t_0 = \frac{1}{\omega_1} ln \left(1 + \frac{u}{m_0}\right) \quad . \tag{43}$$

For the derivation of S we start from the $t < t_0$ case for which $h = 0$ and thus $d\, lnS(t) = 0$ pointing at $S(t)$ being a constant that evaluates to 1 exploiting the properties of the survival function at $t = 0$.
At $t \geq t_0$ instead,

$$\begin{aligned} \frac{d\, lnS(t)}{d\,t} &= -\omega_2 \cdot \frac{m_0\,(e^{\omega_1 t} - 1) + v}{u + v} && \text{Putting } p(t) \text{ in the equation for } h. \\ lnS(t) &= -\omega_2 \int \frac{m_0 e^{\omega_1 t} - m_0 + v}{u + v} d\,t && \text{Separating the variables and integrating.} \\ lnS(t) &= -\omega_2 \left(\frac{v - m_0}{u + v} \cdot t \frac{m_0}{\omega_1} \cdot e^{\omega_1 t}\right) + c && \text{With } c \text{ the integration constant.} \end{aligned} \tag{44}$$

c is found requiring the condition $S(t = t_0) = 1$.
Putting back together the $t < t_0$ and the $t \geq t_0$ case we retrieve eq. 24.

8.3 Passing to *log-space*

We retrieve here the conditional probability $pdf(x|x')$ where $x = ln(m)$ for $h = 0$ starting from eq 17. The $h > 0$ case is analogous.

$$pdf(m|m')\big|_{h=0} = e^{-\gamma(2m-m')}\gamma\left(\frac{1}{m}+2\right)\cdot\left(\frac{2m}{m'}\right)^{-\gamma} \quad \text{Our starting point.}$$

$$pdf(m|m') = e^{-\gamma(2m-m')}\gamma\left(\frac{m'}{m}\right)^{\gamma}\cdot 2^{1-\gamma}\left(\frac{1}{2m}+1\right) \quad \text{Bringing a 2 out of both parenthesis and flipping the first.} \tag{45}$$

Now we make the change $x = ln(m)$ and we multiply by the jacobian e^x leading to,

$$pdf(x|x') = exp\left\{-\gamma\left(2e^x - e^{x'}\right)\right\}\cdot\gamma\left(\frac{e^{x'}}{e^x}\right)^{\gamma}\cdot 2^{1-\gamma}\left(\frac{1}{2e^x}+1\right)e^x \quad \text{We are now in } \textit{log-space.}$$

$$pdf(x|x') = exp\left\{\gamma\left(-2e^x + e^{x'} + x' - x\right)\right\}\gamma 2^{-\gamma}\left(1+2e^x\right) \quad \text{Putting a 2 and } e^x \text{ inside the last parenthesis.} \tag{46}$$

We finish off by rewriting $1 + 2e^x$ as $exp\{ln\ (1 + 2e^x)\}$.

9 Appendix B

9.1 Anomalous scaling, higher moments

We extend here what was shown in the Results: numerical simulations section that showed anomalous scaling for the first 3 moments of lineages' size at birth.
In Fig. 19 we instead display scaling up to the 6-*th* moment.

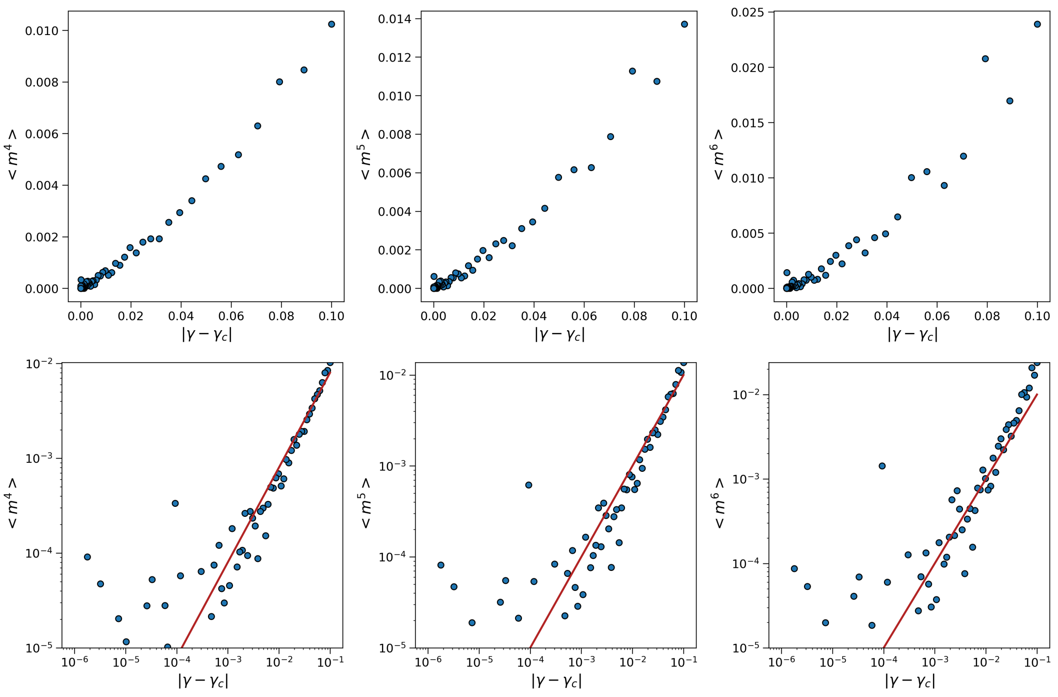


Figure 19: Scaling of the *4-th*, *5-th* and *6-th* moment of the size at birth for lineages of 8 million generations simulated at $h = 0$ and with $|\gamma - \gamma_c|$ transitioning from 10^{-1} to 10^{-6}. Linear scale (top) and *log-log* scale (bottom). The red line is a line with slope 1 drawn as a guide for the eye.

We can extend the reasoning of the first 3 moments also here, confirming the expected anomalous scaling behaviour already reported in [3] for cells populations.

9.2 Bayesian Inference

In this section we briefly outline how we made use of Markov Chain Monte Carlo (MCMC) techniques to obtain random samples from the posterior distribution of the parameters of our models.
We used *Python*'s *emcee* library [27] which exploits a variant of the *Montecarlo-Hastings* algorithm, for the exact implementation we refer to [27].

9.2.1 ***Metropolis-Hastings*** algorithm

Sampling from a distribution can prove to be a difficult task. Markov Chain Monte Carlo algorithms face the problem by designing a Markov chain whose limiting distribution is the desired one, and by obtaining samples by running the chain at equilibrium.
The *Metropolis* algorithm was the first Markov Chain Monte Carlo Method developed [28] and is presented in [29]. Let's call the desired target distribution π (in reality we will see that just something proportional to π is needed) and let us consider, for simplicity, the case of a finite space $X_n \in \{1, \ldots, k\}$. We need an arbitrary jump probability T_{ij}, signaling the probability to jump from element j to i, it has to be symmetric with 0 on the diagonal, i.e.

the probability to stay in the same state is 0 [30],

$$T_{ij} = T_{ji}, \quad T_{ii} = 0 \quad . \tag{47}$$

T is used to generate candidate transitions. Each transition from j to i is accepted with probability

$$A_{ij} = min\left\{1, \frac{\pi_i}{\pi_j}\right\} \quad , \tag{48}$$

meaning that if $\pi_i > \pi_j$ the jump is always accepted, otherwise it is accepted with probability $\frac{\pi_i}{\pi_j} < 1$. The continuous case is analogous if we consider $X \in \mathbb{R}$, with the distribution π being defined on a continuous space $\pi(x)$ and the jump proposal from x to x' being $T = T(x', x)$. The MCMC procedure has the advantage that we do not need to compute the evidence as it goes away in the ratio needed to evaluate A_{ij}, furthermore the markov chain defined by the *Metropolis* algorithm is guaranteed to converge to the desired distribution π [30].

Metropolis' algorithm can be simply modified to allow for non-symmetric jump proposals. In the *Metropolis-Hastings* algorithm one has that the new transition probability, for the continuous case is,

$$A(x', x) = min\left\{1, \frac{\pi(x') \cdot T(x, x')}{\pi(x) \cdot T(x', x)}\right\} \quad . \tag{49}$$

9.2.2 Bayesian inference for our model

We used Bayesian inference considering the model described in 20, 21, inferring the following 7 parameters,

$$\boldsymbol{\theta} = \{u, v, \omega_2, a, b, c, d\} \quad . \tag{50}$$

u, v, ω_2 are the already discussed scales. The two couples of parameters a, b and c, d are instead connected with the division ratio f and the ω_1 parameter respectively. We in fact assume that f is represented by a beta distribution (defined between 0 and 1) i.e,

$$f \sim \frac{x^{a-1} \cdot (1-x)^{b-1}}{B(a, b)} \quad , \tag{51}$$

with $B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$ and Γ the Gamma function.
c and d instead determine the growth parameter ω_1, we use indeed an underlying *Gamma* distribution, which is strictly positive,

$$\omega_1 \sim \frac{1}{\Gamma(c)d^c} \cdot x^{c-1} e^{-\frac{x}{d}} \quad . \tag{52}$$

Applying now the Bayes' theorem we get,

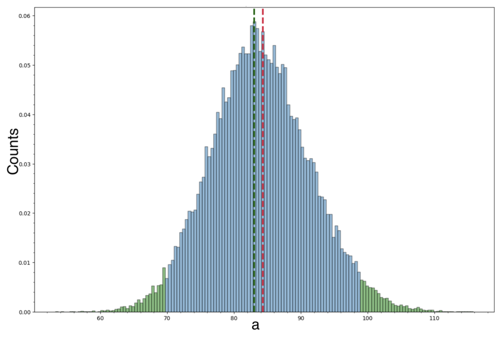
$$P(\boldsymbol{\theta}|\tau, \omega_1, f) \propto \mathcal{L}(\tau, \omega_1, f|\boldsymbol{\theta}) \cdot p(\boldsymbol{\theta}) \tag{53}$$

Where $P(\boldsymbol{\theta}|\tau, \omega_1, f)$ represents the posterior, $\mathcal{L}(\tau, \omega_1, f|\boldsymbol{\theta})$ the likelihood, $p(\boldsymbol{\theta})$ is the prior on our set of parameters $\boldsymbol{\theta}$ and τ is the interdivision time. Notice that we have disregarded the evidence as it is not needed in the Markov Chain Monte Carlo algorithm.
As far as the likelihood goes we can exploit the product rule and the fact that ω_1 and f are independent to find

$$\mathcal{L}(\tau, \omega_1, f|\boldsymbol{\theta}) = \mathcal{L}_1(\tau|\omega_1, f, \boldsymbol{\theta}) \cdot \mathcal{L}_2(\omega_1|\boldsymbol{\theta}) \cdot \mathcal{L}_3(f|\boldsymbol{\theta}) \quad , \tag{54}$$

Where $\mathcal{L}_2$ and $\mathcal{L}_3$ are $Gamma(c, d)$ and $Beta(a, b)$ respectively while $\mathcal{L}_1$ is the probability density function of interdivision times, which depends on the model and it is minus the derivative of the survival function.

We show below, in Fig. 20, two examples of sampled posterior distributions, in blue we have the 95% confidence interval, median (red) and mode (green) are also highlighted. Both left posterior (a parameter) and right posterior (d parameter) have nicely converged. Fig. 21 instead shows example of sampled chains, we see the they already converged as we used a burnout period of 5000, i.e. the firsts 5000 sample of each chain are thrown away.



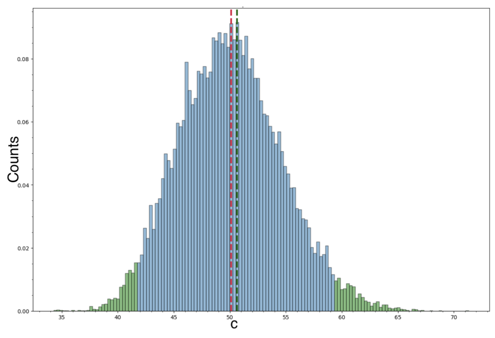


Figure 20: Examples of the posterior distribution for the a parameter (left) and the d parameter (right).

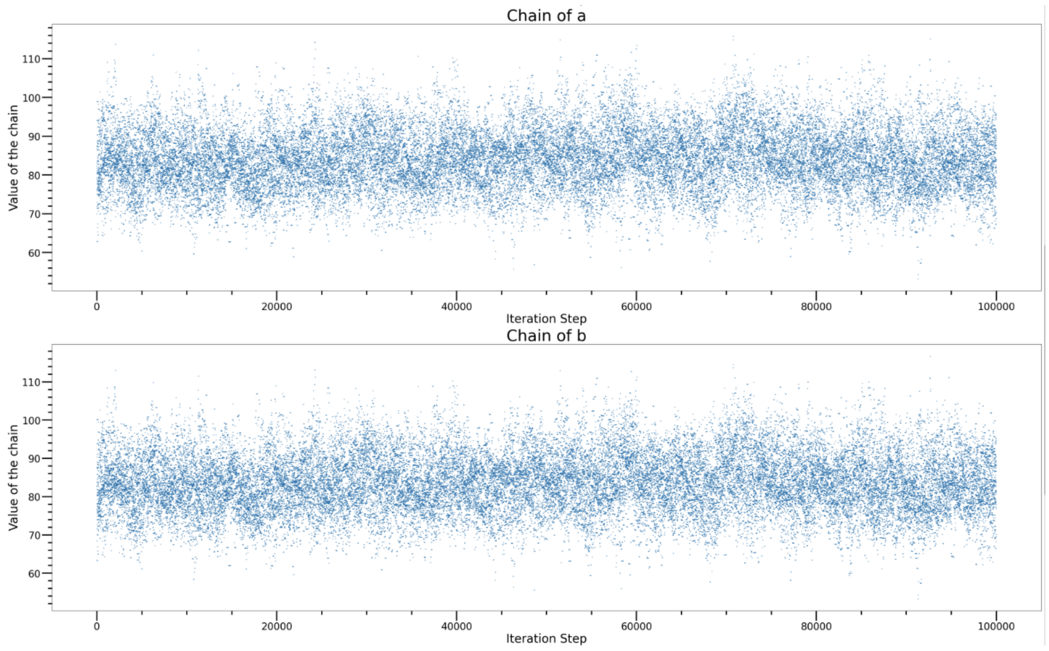


Figure 21: Chains of the a parameter (top) and b parameter (bottom). A burnout period of 5000 samples has been implemented.

10 Appendix C: *Haskell* implementation

Before looking into the specifics of our code implementation we make a brief overview of the *Haskell* programming language highlighting its challenges and advantages in simulating sizes at birth.

The *Haskell* programming language has its roots in the lambda calculus mathematical theory of functions. In the 1930s Alonzo Church attempted to create a system of logic that used only functions and variables, what started as an inquiry on set theory turned out to be a discovery of an universal model of computation, equivalent to the Turing machine [9]. *Haskell* is a functional programming language which can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments. In turn, a functional programming language is one that supports and encourages the functional style [8].
In *Haskell*, functions have to follow 3 rules to make them behave as *pure* functions in the mathematical sense of the term:

- All functions must take an argument.
- All functions must return a value.
- Anytime a function is called with the same argument, it must return the same value.

None of the 3 rules are generally respected in an imperative programming language. Furthermore, the third one, in the context of programming languages takes the name of referential transparency [9].

The pure nature of *Haskell* implies that simple equational reasoning can be used to execute programs, to transform programs, to prove properties of programs and to derive programs directly from specifications of their behaviour. Equational reasoning is particularly powerful when combined with the use of induction to reason about functions that are defined using recursion. We have indeed that cutting-edge research in programming languages is experimenting with ways to mathematically prove that programs will do exactly what you expect [8, 9].
A dummy example of a function in *Haskell* can be given by the identity function.

```haskell
id x = x
```

The above example shows how, reflecting its primary status in the language, function application in *Haskell* is denoted silently using spacing. Parenthesis can be used (but when looking into the code we'll see there are valid alternatives) in applications of the form $f(g(x))$, e.g.

```haskell
doubleId x = id (id x)
```

with `doubleId` a function applying twice the identity.
As mathematical functions do, *Haskell* functions need a domain and a co-domain. *Haskell*'s strong type system achieves exactly that, although it is strongly recommended, you can avoid writing explicitly the types involved in your functions as the compiler's type inference will deduce the right types for you.
The identity function definition with types explicitly written down becomes:

```haskell
id :: a -> a
id x = x
```

with `a` representing a catch for all types as `id` can be applied to any type.
Haskell's type system allows for easily defining type synonyms and new types with as many type constructors as you like. The advantages of this type system are countless, for example it provides an object oriented paradigm that's more flexible then traditional imperative programming languages.
Its main feature though is probably how it makes programs intuitive to understand and easy to read. The types and type synonyms defined in this thesis work are shown in 10.8.

10.1 A safe programming language leads to recursion

Given the time-series structure of our work we will deal with looping a whole lot. Writing down the basics of looping in *Haskell* is thus key in understanding how the main functions we use in this thesis work.
Programs are said to be safe when they always behave exactly the way you expect them to and you can thus easily reason about their behavior. A safe programming language is one that forces your programs to behave as expected.
Moreover, when you change a value in your programming environment, you're changing the program's state. Changing state creates side effects in your code, and these side effects can make code hard to reason about and therefore unsafe [9].

We can for example look at a simple `for` loop in *Python* which computes the factorial of a given number,

```
@typechecked
def factorialFn(n:int) -> int:
    factorial = 1
    for i in range(1,n + 1):
        factorial *= i
    return factorial

print(factorialFn(5))
```

```
>>> 120
```

The `for` loop in the function is inherently unsafe, we are indeed accessing a global (in the context of the function) variable i.e. `factorial` and we are changing it each time. Furthermore the `*=` infix operator could not exists in a language like *Haskell* as it violates the third of our function rules: each time it is called we get a different result.
Finally, in *Haskell* it is better to think at variables as definitions, while the code below compiles with no problems,

```
x :: Int
x = 1
```

the following will raise an error,

```
x :: Int
x = 1

x :: Int
x = 2
```

as we are harming referential transparency so that in our *Python* example we would not be able to assign to `factorial` a new value at each step.

Haskell's basic mechanism for looping is recursion [8]. We can thus write our simple factorial function as follows.

```
factorialFn::Int -> Int
factorialFn 0 = 1
factorialFn n = n * factorial (n-1)

factorialFn 5
```

```
>>> 120
```

The above snippet shows an example of pattern matching in *Haskell*, with the first line being executed if the input is 0 and representing the base case while the second is the recursive condition. For more insights on pattern matching we refer to [9, 8]
A little note on the previous function. We have previously mentioned how being able to reason about your program is one of the main features of *Haskell*, the `factorialFn` is not built toward this goal as we defined overlapping patterns. The only reason why the function works is indeed that the compiler gives precedence to the first condition. A better solution would be to use guards whose in-depth introduction is given in [9, 8], as below.

```
factorialFn::Int -> Int
factorialFn n | n > 0 = n * factorialFn (n-1)
              | otherwise = 1
```

```

factorialFn 5
```

```
>>> 120
```

Throughout these examples we are using the `Int` type which is bounded, i.e. it cannot assume arbitrary high values, if we would need a type with no bounds we could rely on the `Integer` data type.

10.2 Dealing with computations that may fail: the *Maybe* type and finding the root of a function

In the above section we wrote a simple function that computes the factorial of an integer `n`. The factorial though is defined only for integers $n \geq 0$ and our function will fail its purpose for negative integers returning simply 1. While it is possible to raise errors in *Haskell* through its `error` function it is considered bad practice. This is because it then becomes easy to introduce bugs in your code that your compiler cannot check [9].
A great workaround is the `Maybe` type, i.e. a parameterized type whose definition in *Haskell*'s standard *Prelude* reads:

```
data  Maybe a  =  Nothing | Just a
```

with | signaling the logical *or*. So `Maybe` is parameterized by a type `a` (any type) and can be either `Just a` or `Nothing`. `Just a` stands for a computation that completed successfully while `Nothing` highlights that something went wrong. Let's now rewrite our `factorialFn` function so that we don't have to worry about negative integers.

```
factorialFn::Int -> Maybe Int
factorialFn n | n > 0 =  Just $ n * fromJust (factorialFn (n-1))
              | n == 0 = Just 1
              | otherwise = Nothing
```

Some explaining to do. First of all note how our function has changed its type and now returns a `Maybe Int` and not an `Int` anymore. We then see for the first time the `$` operator which just allows us not to overcrowd the code with parenthesis e.g. we could just as well have written `Just ( n * fromJust (factorialFn (n-1)))`. There's nothing fancy about the `$`, its definition in the standard *Prelude* indeed reads,

```
($)::(a -> b) -> a -> b
f $ x =  f x
```

i.e. it takes a function from `a` to `b`, a value of type `a` and returns a value of type `b` and from its definition we see that's just a plain function application. The trick lies in the fact that `$` is a binary operator, so it has lower precedence than any other function we are using [9].
In the definition of `$` we see a snapshot of how *Haskell* deals with multi-argument functions i.e. through currying (celebrating the work of Haskell Curry), for our purposes we just need to know that the last argument of the type signature is the return type, the other ones are the argument types.
Finally `fromJust` takes a value inside the `Just` constructor and brings it out e.g.

```
x :: Int
x = fromJust $ Just 5

print(x)
```

```
>>> 5
```

Let's then see the new factorial function in action.

```
factorialFn 5
```

```
>>> Just 120
```

```
factorialFn (-1)
```

```
>>> Nothing
```

So, although it is not yet the most elegant function, `factorialFn` does its job. The introduction of concepts like *functors*, *monads* and *applicative functors* will allow a better implementation of `factorialFn`.

A careful introduction of computations that may fail in *Haskell* is needed here because the *numericRootFinding* [15] library, which we will use a lot, exploits this concept thoroughly. The major shortcoming of the `Maybe` type is its lack of information on the error message. In our factorial function, if `Nothing` is returned, we know it is due to a negative integer. But if we are dealing with more complex tasks like querying from a database, there are countless things that may go wrong and for all of them we see just `Nothing`. *Haskell* deals with this issues with the `Either` type that allows you to customize errors, without raising them with the `error` function, basically as you like.
For our purposes instead we work with the *numericRootFinding* [15] library. Specifically, we use its `ridders` function, which relies on *Ridders* algorithm [18], to numerically find the root of a function $f : \mathbb{R} \to \mathbb{R}$. This operation takes two inputs: a bracket of doubles defining the interval in which to look for the solution and the function to find the root of. This leads to the following type signature.

```haskell
ridders :: RiddersParam -> (Double, Double) -> (Double -> Double) -> Root
    Double
```

So the function apart from the discussed bracket and function f, it takes as input a value of type `RiddersParam` that is no more than a tuple specifying the maximum number of iterations and the error tolerance for the root approximation. Having a computation that is in danger of failing it returns a `Root Double`, a type introduced below. As per the *numericRootFinding* [15] documentation there are two possible sources of failing,

1. The function does not have opposite signs when evaluated at the lower and upper bounds of the search.
2. The search failed to converge within the given error tolerance after the given number of iterations.

Hence, the library deals with this two cases defining the `Root` data type:

```haskell
-- | The result of searching for a root of a mathematical function.
data Root a = NotBracketed
            | SearchFailed
            | Root a
```

we can see that `Root` is exactly equivalent to the `Maybe` type with the only difference being the specification of the error message: basically the `Nothing` constructor has been split in `NotBracketed` and `SearchFailed` while the `Root` contructor takes the place of `Just`.
We are now fully equipped to understand the function that draws samples from the CDF.

```haskell
drawCDF :: CdfParameters -> RandomNumber -> Root LogSize
drawCDF param unif
| hValue param > 0.0 = ridders def bracketPositiveH $ functionToInvert param
    unif
| xBirth param > -20 = ridders def bracket $ functionToInvert param unif
| otherwise = pure $ - log 2 - (1 / gammaValue param) * log unif + xBirth
    param
  where
    bracket = getBracket $ xBirth param
```

With `def` being the default error tolerance and bound for the number of iterations of the `ridders` function while `pure` is a fancier way of writing `Root` as we will see when talking about *applicative functors*. The conditions following the firsts two guards can be better understood by taking a grasp at *record syntax* as explained in 10.8.

We have now solved the issue of drawing from the CDF and we are left with understanding how to keep a running state and code with side effects in a language like *Haskell*. Actually we have also raised a problem: during simulations we indeed need the size at birth at step i to generate the size at step $i + 1$ but now we do not have a plain `Double` to work with but a `Root Double` and basic operations like multiplication with an `Int` are not defined. For every basic operation we would need a wrapper of the type,

```haskell
customMultiplication :: Root Double -> Int -> Root Double
customMultiplication (Root n) m = (Root m*n)
customMultiplication errorType m = errorType
```

both these 2 hurdles can be overcome through the tools introduced in the next section.

10.3 Monads in *Haskell*

To handle effects in *Haskell* we need *Monads* and *monadic* computations [10], working without them is possible but code quickly becomes cumbersome and hard to understand not to mention the high probability of making errors and the the difficulty in debugging.
But what exactly is a *monad*? It is an *applicative functor* that implements the `(>>=)` (pronounced *bind*) operator. And what is an *applicative functor*? It is a *functor* that implements the `pure` function and the `(<*>)` (*app*) operator: safe to say a little introduction to each notion is needed.

What every mentioned concept have in common is that they work with types in a context i.e. parameterized type that accept only one parameter exactly like `Maybe` and `Root`. We then introduce the graphical notation we will used to get some visual intuition. The graphical notation such as several parts of this section are borrowed from [9].
We will signal different types with different shapes e.g. a circle for an `Int` and a square for a `String`. An empty circle will then represent the embedding in a context while a line stands for a function, Fig. 22 sums up the basics of the notation.

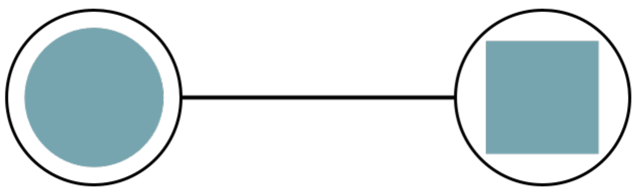


Figure 22: Resume of the graphical notation: we see a type in a context connected through a function to another type in a context e.g. we could be representing a function `Maybe Int -> Maybe Double`.

Throughout the section we will use the parameterized type `Maybe` as an example for its intuitiveness.

10.4 The *functor* type class

We have previously mentioned how with types in a context also the most basic operations like multiplication can be hard. *Haskell* deals with this problem by making the parameterized type a member of the `Functor` class, i.e. we need to define a function called `fmap` whose type signature is shown below.

```haskell
fmap :: Functor f => (a -> b) -> f a -> f b
```

We can make `Maybe` an instance of *functor*:

```haskell
instance Functor Maybe where
-- fmap :: Functor f => (a -> b) -> f a -> f b
fmap func (Just n) = Just (func n)
fmap func Nothing = Nothing
```

where the type signature has been commented out because, without language extensions, it is not allowed in instance declarations. Hence, the functor takes a computation and applies it inside the context if we have a `Just` of something otherwise it propagates `Nothing` as we can see in the following examples were we are using `(<$>)`: an infix synonym of `fmap`.

```haskell
(* 5) <$> Just 2.0
```

```
>>> Just 10.0
```

```
(* 5) <$> Nothing
```

```
>>> Nothing
```

We can also visualize `fmap` with our graphical notation in Fig. 23.

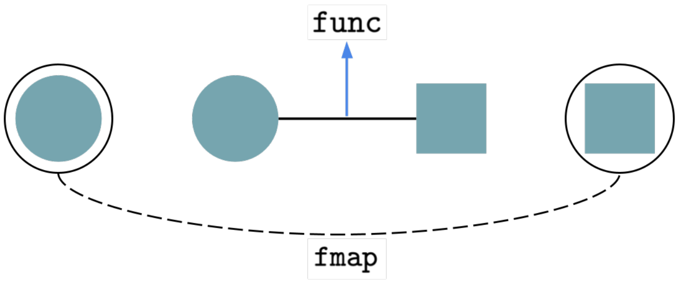


Figure 23: Graphical visualization of what `fmap` does: we have a function `func` that connects type `a` to type `b` while we have `a` in a context (e.g. `Maybe`) and we want to arrive to `b` in a context. `fmap` Does exactly that.

10.5 The applicative type class

Applicative functors generalize *functors* when we deal with functions that take multiple arguments. We did not focus on *currying* here and we will thus present how *applicative* works without getting into the specifics.
We need to define two functions: `(<*>)` (*app*) and `pure`.

```
(<*>) :: Applicative f :: f (a -> b) -> f a -> f b
pure :: Applicative f :: a -> f a
```

i.e. *app* takes a function from `a` to `b` in a context and a value of type `a` in a context giving back a value of type `b` in a context. `pure` allows you to simply put something in a context as we can see graphically in Fig. 24.

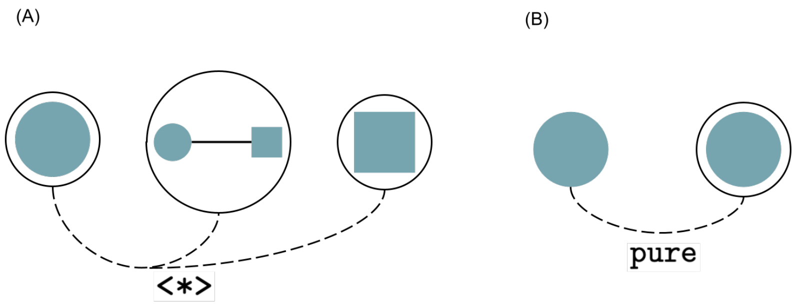


Figure 24: Graphical visualization of the `<*>` operator (A) and of `pure` (B) which simply puts a value in a context.

The *Maybe* type is made an instance of *applicative* with the following definitions

```
instance Applicative Maybe where
-- (<*>) :: f (a -> b) -> f a -> f b
```

```
(<*>) (Just func) (Just n) = Just (func n)
(<*>) Nothing _ = Nothing
(<*>) _ Nothing = Nothing
-- pure :: a -> f a
pure n = Just n
```

with _ a catchall that matches anything.

A couple of examples can show how *applicative* generalizes *functors* when we have functions of multiple arguments like multipication does.

```
pure (*) <*> Just 10 <*> Just 5
```

```
>>> Just 50
```

```
pure (*) <*> Just 10 <*> Nothing
```

```
>>> Nothing
```

10.6 Finally *Monads*

Using *functors* and *applicatives* helps you a lot when dealing with objects in a context. For example we can rewrite our `functorialFn` function in a much more elegant way as the code listing below shows. Not only that, in our previous implementation we used the `fromJust` function and that is considered bad practice as `fromJust Nothing` raises an error.

```
factorialFn::Int -> Maybe Int
factorialFn n | n > 0 =  (* n) <$> factorialFn (n-1)
              | n == 0 = Just 1
              | otherwise = Nothing
```

There is a pattern though with which we still cannot deal. If we call our context `m` (as *monad*) if we have at our disposal `m a` and a function `(a -> m b)` we have no way at this point to retrieve the final `m b`. The problem solved by making types instances of *monad* is visualized in Fig. 25.

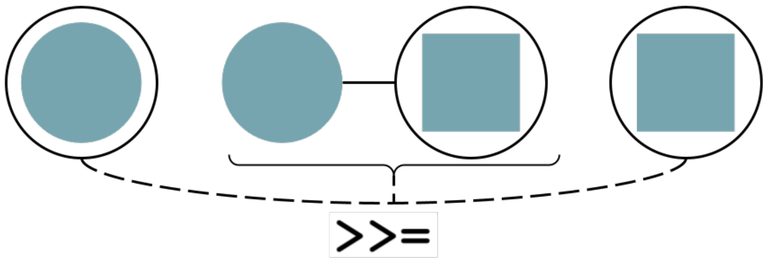


Figure 25: The *monad* class through the `(>>=)` operator takes a value in a contex `m a`, a function `a -> m b` and gives back a value `m b`.

The minimal definition of the *monad* type class requires the declaration of the `(>>=)` operator (*bind*), let's see its type signature.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

We can implement the `(>>=)` operator for the *Maybe* type as

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m g = case m of
              Nothing -> Nothing
              Just x  -> g x
```

with the `case` syntax being an alternative to pattern matching with guards.
It would be nice to make an example of an application of the *bind* operator, for that we need a function of the type `a -> m b`. Turns out we already defined such a function: it's our `factorialFn`! We can then write,

```
(factorialFn 3) >>= factorialFn
```

```
>>> 720
```

because we also needed a value in a context we chose the return of the `factorialFn` function. The result is a chaining of `factorialFn` that computes the factorial of 3 and the factorial of its result i.e. $6! = 720$.
It's exactly this ability to chain together actions that makes *monads* good for our purpose as we need chaining together operations when sampling sizes at birth.

In our function that samples sizes at birth we will use one of the main *monads* in *Haskell* i.e. the *state monad*, many resources explaining this *monad* are available, one of which is the documentation of the library used for the implementation [33], inspired by [34].
At its core though, the *state monad* maintains a state and the result of a computation. When a new computation is performed, the running state is updated and the new result stored.

10.7 Sampling sizes at birth and performances

Below we show our set of two functions responsible for simulating sizes at birth.

```
-- sizeAtBirth :: series length -> StateT state Monad resultComputation
sizeAtBirth :: Int -> StateT SimulationParameters Root LogSize
sizeAtBirth 0 = gets (xBirth . params)
sizeAtBirth n = do
  paramDraw <- get
  let cdfParams = params paramDraw
  let currentTs = accumulatedDraw paramDraw
  let gen = generator paramDraw
  let (unif, gen') = randomR (0,1) gen
  newDraw <- lift $ drawCDF cdfParams unif
  let newTs = currentTs |> newDraw
  put $ paramDraw {generator = gen', accumulatedDraw = newTs, params =
    cdfParams{xBirth = newDraw}}
  sizeAtBirth (n-1)
```

```
simulate :: Int -> SimulationParameters -> Root (Seq Double)
simulate n paramDraw =  accumulatedDraw <$> execStateT (sizeAtBirth n)
    paramDraw
```

With the `simulate` functions that is just returning the final state of our `sizeAtBirth` function and selecting the `accumulatedDraw` parameter of our `SimulationParameters` data type.
Note that we are not exactly using the *state monad* but instead a *state transformer* from [35] whose only difference is that it works smoothly with a computation that involves another *monad* i.e. the `Root` *monad* in our case.
Except for the different utility functions that are specific of the already introduced libraries and can be found in their respective documentation, the thing we have not yet discussed is the `do` notation but that is nothing other that syntactic sugar for the `(>>=)` operator that allows us to avoid complicated lambda functions. The use of the *Data.Sequence* library [31] in the definition of the `TimeSeries` data type and highlighted in the listing above by the appending operator `|>` has proven to be key to obtain competitive performances due to its $\mathcal{O}(1)$ (amortized) asymptotic behaviour in the append operation due to the structure of *Data.Sequence* that is similar to doubly linked lists. For example the *Data.Vector* library [32] while outperforming *Data.Sequence* in indexing suffers in appending ($\mathcal{O}(n)$).

Now that we have our *Haskell* function in place we can compare its performances to a plain *Python* implementation. To do this we run the two programs producing lineages of various lengths ($100k$, $200k$, $300k$, $500k$, $750k$ and 1 million) and measuring the time taken to finish the procedure. For each length 50 iterations are run, mean and standard deviation in seconds are shown in Fig. 26.
Fig. 26 shows how *Haskell* consistently outperforms *Python* providing the queried performance improvement. The scattered points represent the mean of 50 iterations, the error bar are instead 1 standard deviation.

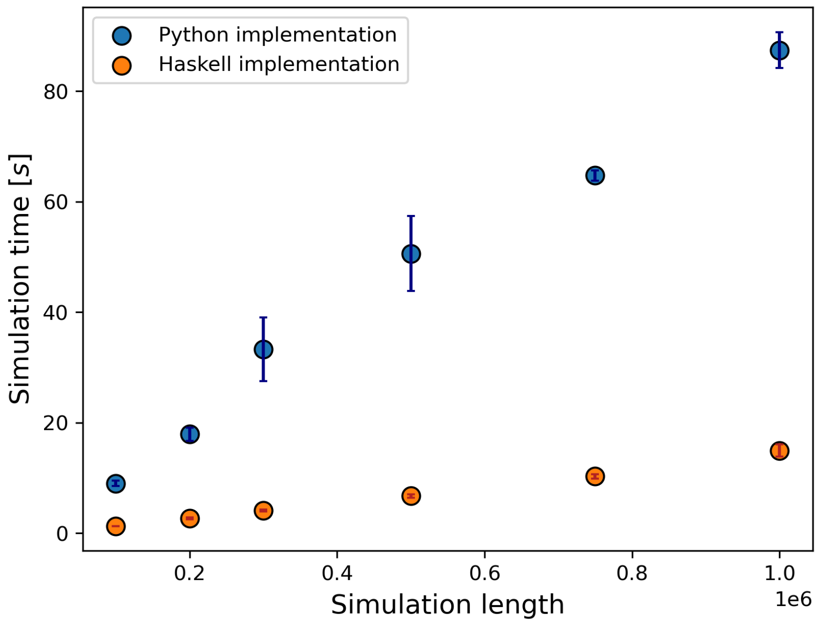


Figure 26: Mean time and standard deviation of 50 *Python* (blue) and *Haskell* (orange) runs are showed for different lineage lengths.

10.8 Custom data types and type synonyms

The type synonyms defined for this thesis and that are useful for understanding the code shown in this section are shown below.

```haskell
type LogSize = Double

type RandomNumber = Double

type Bracket = (Double, Double)

type TimeSeries = S.Seq LogSize

type Size = Double
```

All of them have the purpose of improving code readability e.g. when we are writing a function that, given a random draw, returns a new log-size for our lineage its type won't be a generic `Double -> Double` but a much nicer `RandomNumber -> LogSize`.
The `Bracket` defining a tuple of *Doubles* is mainly used in the context of root finding with *Haskell's numericRootFinding* library [15] from the *containers* package. Finally we define a `TimeSeries` as a sequence of `LogSizes` (or of `Sizes` equivalently) where `S.Seq` is the sequence's data constructor of the *Data.Sequence* library [31]. As previously discussed, *Data.Sequence* has been chosen over for example *Data.Vector* for its very efficient ($\mathcal{O}(1)$) append operation.

Defining new types in *Haskell* can be done with the `data` keyword [9]. To be honest it can be done also with the `newtype` keyword but that's true only if the type has only one constructor with exactly one field in it and it's usually used along with dummy constructors when we have a type synonym that we want to make an instance of something (e.g. an instance of the *monad* class), so we won't focus on this specific case.
Next we show our custom data types

```haskell
data CdfParameters = CdfParameters
  {xBirth :: Double, gammaValue :: Double, hValue :: Double}
  deriving (Show)

data SimulationParameters = SimulationParameters
  {params:: CdfParameters, generator::StdGen, accumulatedDraw :: TimeSeries}
    deriving (Show)
```

where *record syntax* is used, which makes easier to understand which type uses which property [9]. For a proper introduction to *record syntax* we refer to the several resources available online.
The `CdfParameters` type contains the minimal set of parameters needed to define our CDF while `SimulationParameters` represents the current state of our simulation with `generator` being the random generator used for sampling uniformly $u \sim \mathcal{U}\left[0,1\right]$.
Record syntax comes in handy also because it automatically defines functions to retrieve each parameter e.g. if `param` is an instance of the `CdfParameter` type we can retrieve its `hValue` with a function whose name is (not surprisingly) `hValue`.

References

[1] Amir A. Cell Size Regulation in Bacteria. Phys Rev Lett. 2014 May;112:208102. Available from: https://link.aps.org/doi/10.1103/PhysRevLett.112.208102.

[2] Osella L Nugent. Concerted control of Escherichia coli cell division. PNAS. 2014 February;111. Available from: https://www.pnas.org/doi/full/10.1073/pnas.1313715111.

[3] Held J, Lorimer T, Pomati F, Stoop R, Albert C. Second-order phase transition in phytoplankton trait dynamics. Chaos: An Interdisciplinary Journal of Nonlinear Science. 2020 05;30(5):053109. Available from: https://doi.org/10.1063/1.5141755.

[4] Solé RV, Manrubia SC, Benton M, Kauffman S, Bak P. Criticality and scaling in evolutionary ecology. Trends in Ecology Evolution. 1999;14(4):156-60. Available from: https://www.sciencedirect.com/science/article/pii/S0169534798015183.

[5] Wang P, Robert L, Pelletier J, Dang WL, Taddei F, Wright A, et al. Robust Growth of Escherichia coli. Current Biology. 2010;20(4):1099–1103. Available from: https://www.cell.com/current-biology/pdf/S0960-9822(10)00524-5.pdf.

[6] Yu T, Anand P, Heungwons P. A noisy linear map underlies oscillations in cell size and gene expression in bacteria. Nature. 2015;(523):357–360. Available from: https://www.nature.com/articles/nature14562#citeas.

[7] Stawsky A, Vashistha H, Salman H, Brenner N. Multiple timescales in bacterial growth homeostasis. iScience. 2022;25(2):103678. Available from: https://www.sciencedirect.com/science/article/pii/S2589004221016485.

[8] Graham H. Programming in Haskell. London: Cambridge University Press; 2007. Available from: http://www.cs.nott.ac.uk/~pszgmh/book-old.html.

[9] Kurt W. Get Programmung with Haskell. 20 Baldwin Road PO Box 761 Shelter Island, NY 11964: Manning; 2018. Available from: https://www.manning.com/books/get-programming-with-haskell.

[10] Moggi E. Notions of computation and monads. Information and Computation. 1991;93(1):55-92. Selections from 1989 IEEE Symposium on Logic in Computer Science. Available from: https://www.sciencedirect.com/science/article/pii/0890540191900524.

[11] Banavar JR, Green JL, Harte J, Maritan A. Finite Size Scaling in Ecology. Phys Rev Lett. 1999 Nov;83:4212-4. Available from: https://link.aps.org/doi/10.1103/PhysRevLett.83.4212.

[12] Orlandini E, Pagano A. Lecture Notes of Statistical Mechanics; 2020. University of Padova.

[13] Panlilio M, Grilli J, Tallarico G, Iuliani I, Sclavi B, Cicuta P, et al. Threshold accumulation of a constitutive protein explains E. coli cell-division behavior in nutrient upshifts. Proceedings of the National Academy of Sciences. 2021;118(18):e2016391118. Available from: https://www.pnas.org/doi/abs/10.1073/pnas.2016391118.

[14] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020;17:261-72. Available from: https://docs.scipy.org/doc/scipy/.

[15] O'Sullivan B, Khudyakov A. math-functions: Collection of tools for numeric computations; 2018. Available from: https://hackage.haskell.org/package/math-functions-0.3.4.2/docs/Numeric-RootFinding.html.

[16] Inc WR. Mathematica, Version 13.3;. Champaign, IL, 2023. Available from: https://www.wolfram.com/mathematica.

[17] Brent RP. Algorithms for Minimization without Derivatives, Englewood Cliffs. Englewood Cliffs, NJ: Prentice-Hall; 1973. Chapter 4: An Algorithm with Guaranteed Convergence for Finding a Zero of a Function.

[18] Ridders C. A new algorithm for computing a single root of a real continuous function. IEEE Transactions on Circuits and Systems. 1979;26(11):979-80. Available from: https://ieeexplore.ieee.org/abstract/document/1084580.

[19] Susman L, Kohram M, Vashistha H, Nechleba JT, Salman H, Brenner N. Individuality and slow dynamics in bacterial growth homeostasis. Proceedings of the National Academy of Sciences. 2018;115(25):E5679-87. Available from: https://www.pnas.org/syndication/doi/10.1073/pnas.1615526115.

[20] Yang D, Jennings AD, Borrego E, Retterer ST, Männik J. Analysis of factors limiting bacterial growth in PDMS mother machine devices. Frontiers in microbiology. 2018;9:871. Available from: https://www.frontiersin.org/articles/10.3389/fmicb.2018.00871/full.

[21] Marlow S, et al. Haskell 2010 language report. Available online (May 2011). 2010. Available from: http://www.haskell.org/.

[22] Kitagawa G. Introduction to time series modeling. CRC press; 2010. Available from: https://www.taylorfrancis.com/books/mono/10.1201/9781584889229/introduction-time-series-modeling-genshiro-kitagawa.

[23] Grinstein G, Muñoz MA, Tu Y. Phase Structure of Systems with Multiplicative Noise. Phys Rev Lett. 1996 Jun;76:4376-9. Available from: https://link.aps.org/doi/10.1103/PhysRevLett.76.4376.

[24] Graham R, Schenzle A. Carleman imbedding of multiplicative stochastic processes. Phys Rev A. 1982 Mar;25:1731-54. Available from: https://link.aps.org/doi/10.1103/PhysRevA.25.1731.

[25] Seabold S, Perktold J. statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference; 2010. Available from: https://www.statsmodels.org/stable/index.html.

[26] Christopher M B. Pattern Recognition and Machine Learning. 233 Spring Street, New York, NY 10013, USA: Springer Science+Business Media, LLC; 2006. Available from: https://link.springer.com/book/9780387310732.

[27] Foreman-Mackey D, Hogg DW, Lang D, Goodman J. emcee: the MCMC hammer. Publications of the Astronomical Society of the Pacific. 2013;125(925):306. Available from: https://iopscience.iop.org/article/10.1086/670067/meta.

[28] Koch KR. Introduction to Bayesian statistics. Springer Science & Business Media; 2007.

[29] Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E. Equation of state calculations by fast computing machines. The journal of chemical physics. 1953;21(6):1087-92. Available from: https://doi.org/10.1063/1.1699114.

[30] Allegra M. Notes on information theory and inference; 2022. University of Padova.

[31] Paterson BFDFR, Straka M. containers: Assorted concrete container types; 2014. Available from: https://hackage.haskell.org/package/containers-0.6.7/docs/Data-Sequence.html.

[32] Khudyakov RLAKA, Lelechenko A. vector: Efficient Arrays; 2008. Available from: https://hackage.haskell.org/package/vector-0.13.0.0/docs/Data-Vector.html.

[33] Gill A. mtl: Monad classes for transformers, using functional dependencies; 2001. Available from: https://hackage.haskell.org/package/mtl-2.3.1/docs/Control-Monad-State-Lazy.html.

[34] Jones MP. Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1. Springer; 1995. p. 97-136.

[35] Gill A. transformers:Concrete functor and monad transformers; 2001. Available from: https://hackage.haskell.org/package/transformers-0.6.1.1/docs/Control-Monad-Trans-State-Lazy.html.

[36] Field CB, Behrenfeld MJ, Randerson JT, Falkowski P. Primary production of the biosphere: integrating terrestrial and oceanic components. science. 1998;281(5374):237-40.

[37] ElGamel M, Vashistha H, Salman H, Mugler A. Multigenerational memory in bacterial size control; 2023.