Faculté des sciences
École de Physique
IRMP

NUMERICAL STUDY OF THE INVERSE – GAMMA 1+1 DIRECTED POLYMER

Auteur : Sébastien Walschaerts
Lecteur : Dr. Alexandre Lazarescu
Promoteur : Prof. Dr. Tom Claeys
Copromoteur : Prof. Dr. Philippe Ruelle

Mémoire réalisé en vue de l'obtention du grade académique
de Master [120] en sciences physiques, finalité approfondie.

Outline and motivation

The first part of this thesis is an introduction in which directed polymer models are described and their relation to other processes is mentioned for the specific case of a random environment with weights drawn from an inverse-gamma distribution.

In the second part, a rigorous mathematical model is constructed in order to benefit from various and already established results related to directed polymer models.

The third and final part is dedicated to a numerical implementation of the model. It starts with a description of the algorithm design and follows a comparison of the numerical results obtained and with is expected from the known theorems.

At the end of this thesis, the code that was used to generate all the numerical results within this document is given in order to let anyone work with the model and its numerical implementation.

The rapid and recent developments of the directed polymer models over the last three decades and their link to other processes through their belonging to a universality class make this subject interesting to study. As for this numerical study, which doesn't seem to have been conducted before, any contribution, even a small one, feels like it can bring a lot of knowledge to the entire field.

Acknowledgements

A master's thesis represents the end of the graduate and undergraduate cycles of study. If normally this journey takes approximately 5 years, mine started 16 years ago and took so long because of struggles that happened during my life. It is therefore a good section to thank the people who contributed to this success in these recent years when I restarted my studies at the Catholic University of Louvain-la-Neuve.

I would like to start by thanking the people working at the administration of the Faculty of Sciences. It's difficult to know how good they are unless you have studied in more than one university. The robustness and kindness of their team constitute an asset for any student wishing to engage in complicated studies. For the help they gave me these years, thank you.

A lot of teachers are passionate within this university and it was a pleasure to learn with them. To not cite them all, I'd like to thank Prof. Dr. Jan Govaerts particularly because he was the one that left me the greatest desire to stay in class and keep learning everything he was inclined to share. I'm sure many of us think of him as a great teacher and a remarkable human being.

This thesis wouldn't have been possible without the help of my supervisor Prof. Dr. Tom Claeys. I could already thank him a lot for the opportunity he gave me to work with him in order to finish my graduation but it was so pleasant to follow his supervision that it wouldn't be enough. Professor Claeys is both insightful and humble, qualities that explain easily his success but also why it is a privilege to work with him. I thank him very much for his guidance through this thesis, it was really appreciated.

The reviewers of my thesis, Prof. Dr. Philippe Ruelle and Dr. Alexandre Lazarescu, gave me good feedback during my pre-defense, which was useful till the very end of the redaction since they thought of some possible solutions to the numerical results that weren't all completely satisfying at that moment. I thank them very much for the interest and time they took for my work.

I'd like to mention and thank two friends of mine who helped me during the writing of this thesis: Maxime Harvengt, who motivated me and with whom I've studied a lot during this last year ; and also Nicolas Robert, with whom it's always a pleasure to talk and take advice.

Finally, I thank my girlfriend Joana Wolfrum Caeiros. It is said that behind every successful man, there is a woman. I'm not sure if this is true in general and I certainly don't think of me as a successful man yet but I can say that in my case she was such a great help to me through all these years that this accomplishment simply couldn't have been possible otherwise.

CONTENTS

1. Introduction

This chapter explains how directed polymer was introduced and gives the foundation of the model used in this thesis before the next chapter develops it mathematically. Directed polymer models are related to plenty of other concepts, which are mentioned and seen in more detail for the particular connections to a universality class and a specific distribution because of their impact on the numerical results.

1.1. Introduction of polymer models

Polymer models

A polymer can be thought of as a long chain of molecules made up of repeating subunits called monomers. They are used as models in statistical physics to describe various concepts such as:

- interfaces in disordered systems ;
- surface growth in stochastic processes ;
- polymer evolution in random media ;
- vortex lines in superconductors ;
- flow paths in fluid dynamics, see the paper [12] for a recent example ;
- and many more.

Polymer models were first introduced in 1985 by Huse and Henley in the article [9] through the notion of an interface, which they called a "domain wall," between two distinct regions of different spin values in the two-dimensional Ising model, see Figure 1 below.

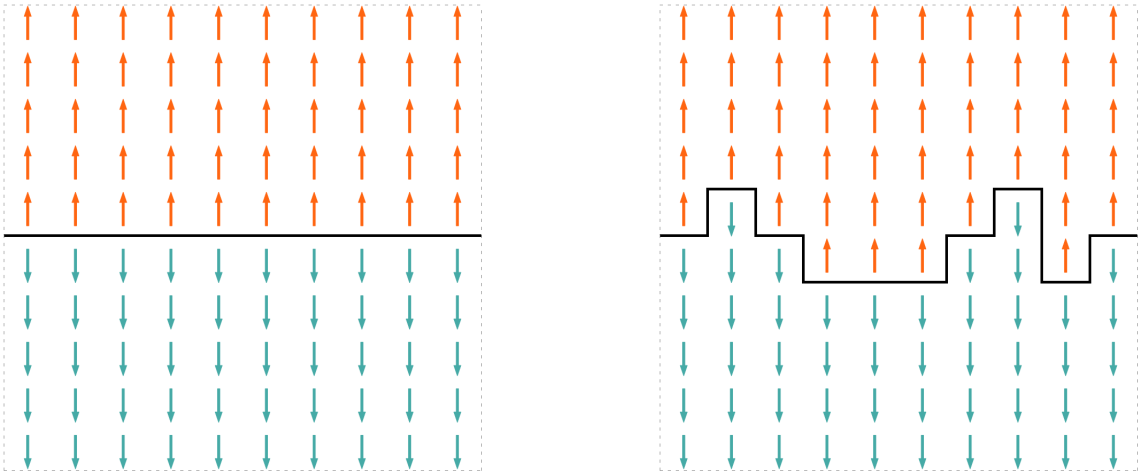


Figure 1: Pure (on the left) and with random impurities (on the right) states of the 2D Ising model with the domain wall (the interface) in black that separate the two regions of different spin values.

The study of phase transitions in disordered magnets through the notion of domain walls which are the consequences of randomly generated impurities wasn't new at that time. The novelty was that the random couplings $\Delta J_{i,j}$ between two nearest neighbour sites i and j in the Hamiltonian:

$$H = H_{\text{pure}} + H_{\text{impure}} = -J \sum_{<i,j>} s_i \, s_j \; + \sum_{<i,j>} \Delta J_{i,j} \; s_i \, s_j$$

induced a disorder with only short-range correlations and in a way that the ferromagnetic or antiferromagnetic ordering was preserved at low temperatures.
They studied the ground state, called the "roughness," of that interface and demonstrated that the domain wall in a disordered Ising model can be understood as a directed path (or polymer) in a random potential (or media), making a direct connection between the behaviour of domain walls and the theory of directed polymers.

Directed Polymer

This concept of interfaces is now encompassed in the theory of directed polymer that emerged formally in 1988 with the work of Derrida and Spohn in [13], and with the generalisation that came later the same year by Imbrie and Spencer in [10] as a random walk influenced by a random environment [RWRE]. Random walks are represented by paths in a lattice such as a $(t+d)$-dimensional grid where d denotes the ***spatial*** dimension and where one additional $t = 1$ ***time*** direction or dimension indicates the growth. The words "directed polymer" refer to the graph of a random walk for which it is not allowed to go back in the values of the time dimension. The definition of a random walk they introduced and which is still in use in a lot of models is given by:

$$\begin{aligned} &\text{random walk } x : [0, T] \cap \mathbb{Z} \mapsto \mathbb{Z}^d \\ &\text{such that } \begin{cases} x(0) = 0 \\ |x(t+1) - x(t)| = 1 \, . \end{cases} \end{aligned}$$

For $d := 1$, it can be visualized in Figure 2 below, which shows that the path is constrained to evolve inside a space-time wedge due to its definition.

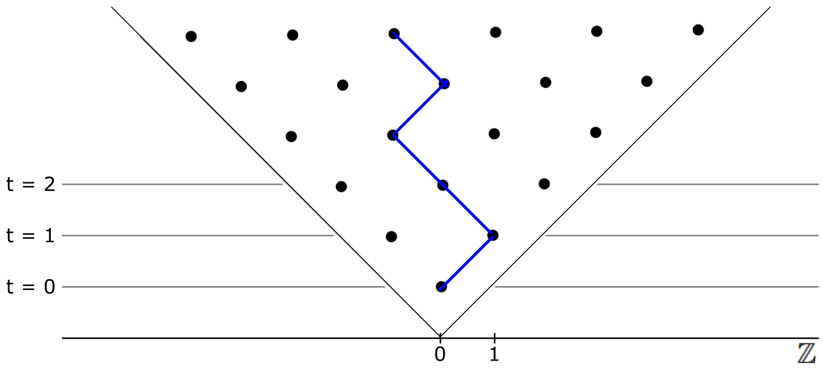


Figure 2: Representation of a 1+1 directed polymer.

Nowadays, a 1+d directed polymer system is formally defined as a statistical ensemble of paths immersed in $\mathbb{Z}^d$, parametrized by time, which is usually discrete but can be continuous, and where the polymer itself is the graph $\{(t, x(t)) \in \mathbb{N} \times \mathbb{Z}^d\}_{t=1}^n$ of the nearest neighbours $(x(t))_{t=1}^n$ where n is the length of the path.

Quenched disorder

The random environment is the second key aspect in the study of polymer models. It is introduced by assigning random energy-related quantities $\omega(t, x)$ to each site of the lattice. These energies are typically extracted from a specific probability distribution and represent random potentials or varying energy landscapes. These impurities, called "quenched disorder," within the lattice introduce complexity and lead to interesting statistical properties while studying polymer models. The random environment is therefore defined as a set of random variables $\Omega \equiv \{\omega(t, x) \in \mathbb{N} \times \mathbb{Z}^d\}$ and a probability space $(\Omega, \mathcal{F}, P)$.

Partition function and free energy

The partition function Z is an important quantity in statistical physics. It is defined as the sum of all possible configurations of the polymer, that is all the possible paths, weighted by the random medium. For a system of directed polymers of length n in a random environment given by Ω defined above, the partition function is:

$$Z_n^\omega = \sum_{\text{paths}} \exp\left(-\beta \sum_{t=0}^{n} \omega(t, x(t))\right) \tag{1}$$

where β is the inverse temperature.

The free energy F of the system is related to the partition function by the relation:

$$F_n^\omega = -\frac{1}{\beta} \log(Z_n^\omega)\,. \tag{2}$$

Remark: In this thesis, all logarithms are natural logarithms.

Studying the scaling behaviour of the free energy and the shape of the polymer itself for a large value of n informs a lot about the properties of the system, such as its roughness or its diffusion, and how it responds to the disorder in the environment.

In summary

Directed polymers are a generalisation that allows a more abstract study of the kinetics of interfaces and random walks in a random environment. These models capture essential aspects of many physical systems, from materials science to biological processes, and are key to understanding complex phenomena in disordered systems.

The reader eager to know more about the various models of directed polymer may find the book [14] interesting.

1.2. The inverse-gamma 1+1 directed polymer model

In this thesis, the specific model of a 1+1 directed polymer with a random medium constituted by weights drawn from an inverse-gamma distribution is used. The 1+1 directed polymer model is a fundamental and well-studied case in the theory of directed polymers. This simplified version with inverse-gamma weights allows for exact solutions that make the comparison to numerical results easier. Furthermore, it is possible to express the same model in a representation of the first quadrant of a two-dimensional integer lattice instead of in a space-time wedge.

1+1 directed polymer

Throughout this study, the spatial dimension is set to one, $d \coloneqq 1$. Instead of working with the representation given by Figure 2, where the restriction on the positions $x(t) \in \mathbb{Z}$ makes the path evolve in a space-time wedge $\{(t, x(t)) : t \in \mathbb{N}, x \in \mathbb{Z}, |x| \leq t\}$, it is possible to work with another representation more suitable for numerical implementation.

To do so, it suffices to rotate the previous figure 45° clockwise, see below:

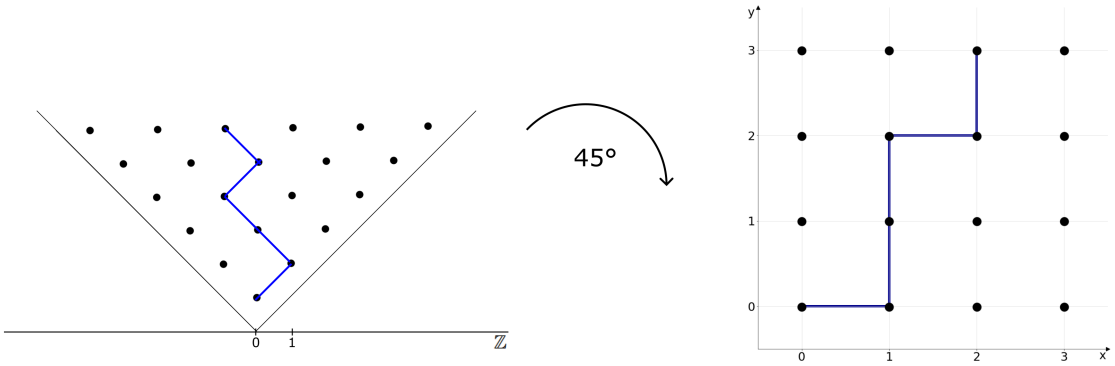


Figure 3: Passage from the space-time wedge representation of a 1+1 directed polymer to a two-dimensional integer lattice representation.

This new representation of a 1+1 directed polymer is called an up-right random walk and is defined in a $\mathbb{Z}_+^2$ integer lattice. The "up-right" mention refers to the "directed" constraint of the polymer. Indeed, going left or bottom in this new representation would mean going back in the time direction in the space-time wedge representation. Formally, to go from the first representation with positions $(t, x(t)) \in [0, T] \cap \mathbb{Z} \times \mathbb{Z}$ to the second one with the positions given by $(m, n) \in \mathbb{Z}^2$, the following transformation can be done:

$$\begin{cases} t \\ x(t) \end{cases} \longmapsto \begin{cases} m_0 \coloneqq 0 \\ m_t \coloneqq \sum_{s=0}^{t} \delta_{x(s),x(s-1)+1} = m_{t-1} + \delta_{x(t),x(t-1)+1} \\ n_t \coloneqq t - m_t \end{cases} \tag{3}$$

where $\delta_{i,j}$ is the Kronecker delta.

In this thesis, the model studied is an up-right random walk with an endpoint associated with a site along the diagonal, that is, with the relation $m = n$.

Remark: A quick observation of this relation brought back to the space-time wedge representation indicates that only paths with even values of the discrete time t will be considered.

Inverse-gamma distribution

The random environment that is chosen to add a quenched disorder to the random walks is a set of weights extracted from an inverse-gamma distribution, which is defined and rigorously studied in the next chapter. The Figure 4 below gives an overview of it. The inverse-gamma distribution has heavy tails, which means it places a relatively high probability on large values of the positive

real axis $\mathbb{R}_{\geq 0}$. It ensures that the prior is not overly restrictive, allowing for greater variability in the data. This property can be particularly useful in models where large variance values are plausible, which, in this thesis, are related to one of the three established theorems for which a comparison with the numerical results is done in Chapter 3.

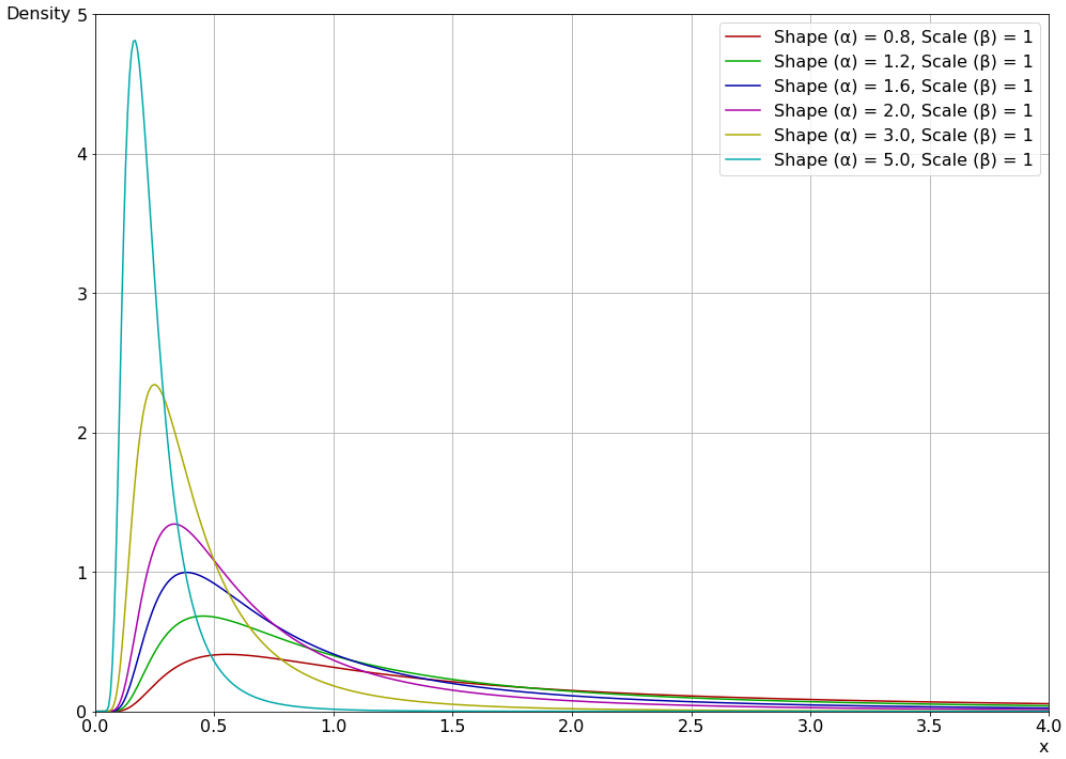


Figure 4: Inverse-gamma distribution for different shape parameter values.

The inverse-gamma distribution has two parameters:

- the shape parameter α: it controls the tail behaviour and concentration of the distribution. A higher value of this parameter results in a higher peak in density with a higher probability to obtain values near to it ;
- the scale parameter β: it scales the distribution and reflects the expected value.

In the numerical results given in Chapter 3, the scale parameter β is set to one and the studied values of shape parameter α are exactly the ones in the Figure 4.

The interest of this model

These choices for the random walk and the random environment of the polymer were previously referred to as a "log-gamma directed polymer" model and are more recently called "inverse-gamma 1+1 directed polymer" with the dimensions ($t = 1$ and $d = 1$) sometimes omitted. This simpler model allows for exact solutions using methods from integrable systems, queuing theory with martingales processes and even from advanced mathematical techniques such as in [15] that use coordinate Bethe Ansatz.

The 1+1 directed polymer model is a fascinating and rich area of study in statistical physics, and it

is intricately linked to several other models and concepts in mathematical physics and probability theory. A long but quite incomplete list of its relation to these other concepts is given below:

- Kardar-Parisi-Zhang (KPZ) universality class: The KPZ universality class is related to the growth of interfaces between two regions where one invades the other. The so-called "KPZ equation" describes their evolution in mathematical terms and is written as:

$$\frac{\partial}{\partial t}h(x,t) = \nu\,\Delta h(x,t) + \frac{\lambda}{2}\left(\nabla h(x,t)\right)^2 + \sqrt{D}\,\eta(x,t) \tag{4}$$

  where $\nu, \lambda \in \mathbb{R}, D > 0$ are physical constants and $h(x,t)$ is the height of the growing interface at position x and time t, generally referred to as the "height function."
  The free energy of the directed polymer, which is defined in the next chapter, can be mapped onto the height function $h(x,t)$ in the KPZ equation (4), with the growth of the interface corresponding to the evolution of the free energy ;

- Random matrix theory: the theory of random matrices, which are matrix-valued random variables meaning that some or all of their entries are sampled randomly from a chosen probability distribution. The fluctuations of the largest eigenvalue of certain of these random matrices result in a Tracy-Widom distribution, detailed at the end of the next chapter, which is the same distribution of the fluctuations of the free energy associated with the inverse-gamma directed polymer studied in this thesis ;

- Last passage percolation (LPP): introduced by Hammersley and Welsh in [11] (1965) to make a model of a fluid passing through a random environment, the LPP is a well-known model [17,18], which could easily be numerically studied with a slight adaptation of the algorithm used in this thesis ;

- Burgers'equation: this equation describes the dynamics of a fluid with viscosity. The transformation between the KPZ height function and the Burgers' velocity field provides a direct link between the directed polymer model and fluid dynamics ;

- Spin Glasses: they are diluted magnetic material in which the magnetic moments are randomly interacting, with a huge number of metastable states which prevent reaching equilibrium. Techniques from spin glass theory, such as the replica method, have been adapted to study directed polymers and both systems exhibit complex energy landscapes with many local minima and are characterized by disorder ;

- Free fermions and integrable systems: certain exactly solvable models of directed polymers can be mapped onto problems involving free fermions [19] and integrable systems. These mappings allow the use of powerful mathematical techniques from integrable systems to derive exact results for the distribution of free energy and other quantities.

These connections help in understanding the properties of directed polymers and provide tools for solving complex problems. The first two of the list are interesting for this thesis because of their direct link to the numerical results, seen further in the chapter 3.

2. Model's definition and properties

This chapter is dedicated to the definition of the model and the properties that emerge from it, followed by the enunciation of the known results from the two papers [1,2] that are verified in this numerical study.

2.1. Definition of the model

In this section, all the mathematical objects that constitute the basis of this model are introduced in order to connect the known theorems to the numerical results obtained from them.

The gamma and inverse-gamma distributions

The gamma function is defined as the following:

$$\Gamma(x) \equiv \int_0^{+\infty} t^{x-1}\, e^{-t}\, dt \qquad \text{for } x > 0. \tag{5}$$

The digamma ψ_0 and trigamma ψ_1 functions are defined through the polygamma expression:

$$\psi_m(x) \equiv \left(\frac{d}{dx}\right)^{m+1} \log(\Gamma(x)) \qquad \text{for } x > 0. \tag{6}$$

Let $\alpha, \beta > 0$ be real parameters. A positive random variable X follows a gamma distribution with shape paramater α and rate parameter β, noted $X \sim \mathrm{Ga}(\alpha, \beta)$, if it has density function

$$f_X(x) = \frac{1}{\Gamma(\alpha)}\, x^{\alpha-1}\, \beta^{\alpha}\, e^{-\beta x} \qquad \text{for } x > 0 \tag{7}$$

such that

$$\mathbb{P}\left[X \leq s\right] = \int_0^s f_X(x)\, dx. \tag{8}$$

The main distribution used in this thesis is the inverse-gamma distribution:

Definition 1. [Inverse-gamma distribution]
A positive random variable Y has an inverse-gamma distribution with shape parameter α and rate parameter β, noted $Y \sim \mathrm{Ga}^{-1}(\alpha, \beta)$, if its reciprocal Y^{-1} follows a gamma distribution: $Y^{-1} \sim \mathrm{Ga}(\alpha, \beta)$ where $\alpha, \beta > 0$ are real parameters. Its associated probability density function is

$$f_Y(y) = \frac{1}{\Gamma(\alpha)}\, y^{-1-\alpha}\, \beta^{\alpha}\, e^{-\frac{\beta}{y}} \qquad \text{for } y > 0 \tag{9}$$

such that

$$\mathbb{P}\left[Y \leq s\right] = \int_0^s f_Y(y)\, dy. \tag{10}$$

Equation (9) is the result of the link between the cumulative density function F of two distributions X and Y, which are set as the inverse of each other $Y \coloneqq X^{-1}$, and for which the following relation holds:

$$F_Y(y) \equiv \mathbb{P}\left[Y \leq y\right] = \mathbb{P}\left[X \geq y^{-1}\right] = 1 - \mathbb{P}\left[X < y^{-1}\right] = 1 - F_X(y^{-1}) \tag{11}$$

which implies that

$$f_Y(y) \equiv \frac{d}{dy}\left[F_Y(y)\right] \;=\; \frac{d}{dy}\left[1 - F_X(y^{-1})\right] \;=\; -f_X(y^{-1})\,\frac{\partial}{\partial y}(y^{-1}) \;=\; \frac{1}{y^2}\,f_X(y^{-1}) \tag{12}$$

and therefore, for the specific case of $X \sim \mathrm{Ga}(\alpha, \beta)$ for which f_X is given by the equation (7), its inverse distribution is indeed given by the equation (9):

$$f_Y(y) = \frac{1}{y^2}\,f_X(y^{-1}) \;=\; \frac{1}{\Gamma(\alpha)}\,(y^{-2})\,(y^{-1})^{\alpha-1}\,\beta^{\alpha}\,e^{-\beta y^{-1}} \;=\; \frac{1}{\Gamma(\alpha)}\,y^{-1-\alpha}\,\beta^{\alpha}\,e^{-\frac{\beta}{y}}\,. \tag{13}$$

Proposition 1. [Expectation value of the logarithm of an inverse-gamma distribution]
For an inverse-gamma distribution $Y \sim \mathrm{Ga}^{-1}(\alpha, \beta)$ with the rate parameter set to one $\beta \coloneqq 1$, the expectation value of the random variable $\log(Y)$ is given by the opposite value of the digamma function:

$$\mathbb{E}\left[\log(Y)\right] = -\psi_0(\alpha). \tag{14}$$

Proof. The value of the digamma function given by the relation (6) for $m \coloneqq 0$ is:

$$\psi_0(\alpha) \equiv \frac{d}{d\alpha}\left[\log(\Gamma(\alpha))\right] \;=\; \frac{1}{\Gamma(\alpha)}\,\frac{d}{d\alpha}\left[\Gamma(\alpha)\right] \;=\; \frac{1}{\Gamma(\alpha)}\int_0^{+\infty} y^{\alpha-1}\,e^{-y}\,\log(y)\,dy\,. \tag{15}$$

The expectation value of the random variable $\log(Y)$ with $Y \sim \mathrm{Ga}^{-1}(\alpha, \beta)$ is:

$$\begin{aligned}
\mathbb{E}\left[\log(Y)\right] &\equiv \int_0^{+\infty} \log(y)\,f_Y(y)\,dy \\
&= \int_0^{+\infty} \log(y)\,\frac{1}{\Gamma(\alpha)}\,y^{-1-\alpha}\,\beta^{\alpha}\,e^{-\frac{\beta}{y}}\,dy \\
&= \frac{\beta^{\alpha}}{\Gamma(\alpha)}\int_0^{+\infty} y^{-1-\alpha}\,e^{-\frac{\beta}{y}}\,\log(y)\,dy\,.
\end{aligned}$$

Changing the measure through the transformation $y = x^{-1}$, $dy = -x^{-2}dx$ gives :

$$\begin{aligned}
\mathbb{E}\left[\log(Y)\right] &= \frac{\beta^{\alpha}}{\Gamma(\alpha)}\int_{+\infty}^{0} (x^{-1})^{-1-\alpha}\,e^{-\beta x}\,\log(x^{-1})\,(-x^{-2})dx \\
&= \frac{\beta^{\alpha}}{\Gamma(\alpha)}\int_0^{+\infty} x^{\alpha-1}\,e^{-\beta x}\,\left[(-1)\log(x)\right]dx \\
&= -\frac{\beta^{\alpha}}{\Gamma(\alpha)}\int_0^{+\infty} x^{\alpha-1}\,e^{-\beta x}\,\log(x)\,dx
\end{aligned}$$

which for $\beta \coloneqq 1$ concludes the proof by subtituting the relation (15):

$$\mathbb{E}\left[\log(Y)\right] \;=\; -\frac{1}{\Gamma(\alpha)}\int_0^{+\infty} x^{\alpha-1}\,e^{-x}\,\log(x) \;=\; -\psi_0(\alpha).$$

□

Proposition 2. [Variance of the logarithm of an inverse-gamma distribution]
For an inverse-gamma distribution $Y \sim \mathrm{Ga}^{-1}(\alpha, \beta)$ with the rate parameter set to one $\beta \coloneqq 1$, the variance of the random variable $\log(Y)$ is given by the value of the trigamma function:

$$\mathbb{V}\mathrm{ar}\left[\log(Y)\right] \;=\; \psi_1(\alpha). \tag{16}$$

This proposition is proved by using the following lemma:

Lemma 1. [n-th order derivative of the gamma function]
Let $x > 0$ be a real parameter. For $n \in \mathbb{Z}_+$, if Γ is the gamma function defined in equation (5), then

$$\frac{d^n}{dx^n}\left[\Gamma(x)\right] \;=\; \int_0^{+\infty} t^{x-1}\, e^{-t}\, (\log(t))^n\, dt \tag{17}$$

Remark: For the two following proofs, both standard notations of the derivative of a function f will be used interchangeably:

$$\frac{d^m}{dx^m}\, f(x) \;=\; f^{(m)}(x)$$

and in particular, the prime and double prime symbols will be used for the first ($m \coloneqq 1$) and second derivative ($m \coloneqq 2$) respectively:

$$\frac{d}{dx}\, f(x) \;=\; f'(x) \quad ; \quad \frac{d^2}{dx^2}\, f(x) \;=\; f''(x)\,.$$

Proof. The proof is done by induction. The case $n \coloneqq 0$ is true since it is exactly the definition of the gamma function (5). Let suppose equation (17) is true for n, then

$$\begin{aligned}
\frac{d^{n+1}}{dx^{n+1}}\left[\Gamma(x)\right] \;&=\; \frac{d}{dx}\left[\Gamma^{(n)}(\alpha)\right] \\
&=\; \frac{d}{dx}\left[\int_0^{+\infty} t^{x-1}\, e^{-t}\, (\log(t))^n\, dt\right].
\end{aligned}$$

By the dominated convergence theorem, the differentiation operator can pass under the integral sign from where, by using differentiation rules:

$$\begin{aligned}
\frac{d^{n+1}}{dx^{n+1}}\left[\Gamma(x)\right] \;&=\; \int_0^{+\infty} \frac{\partial}{\partial x}\left[t^{x-1}\, e^{-t}\, (\log(t))^n\, dt\right] \\
&=\; \int_0^{+\infty} \frac{\partial}{\partial x}\left[t^x\right]\, t^{-1}\, e^{-t}\, (\log(t))^n\, dt \\
&=\; \int_0^{+\infty} \left[t^x\, log(t)\right]\, t^{-1}\, e^{-t}\, (\log(t))^n\, dt \\
&=\; \int_0^{+\infty} t^{x-1}\, e^{-t}\, (\log(t))^{n+1}\, dt\,,
\end{aligned}$$

which completes the lemma's proof by induction for all $n \in \mathbb{Z}_+$.

□

Proof of proposition 2. Since $Y \sim \mathrm{Ga}^{-1}(\alpha, \beta)$ with $\beta \coloneqq 1$, the proposition 1 can be used. Therefore, by definition of the variance and by using the relation (14):

$$\begin{aligned}
\mathbb{V}\mathrm{ar}\left[\log(Y)\right] &\equiv \mathbb{E}\left[(\log(Y))^2\right] - \left(\mathbb{E}\left[\log(Y)\right]\right)^2 \\
&= \int_0^{+\infty} (\log(y))^2 \, f_Y(y) \, dy - (-\psi_0(\alpha))^2 \\
&= \frac{1}{\Gamma(\alpha)} \int_0^{+\infty} y^{\alpha-1} \, e^{-y} \, (\log(y))^2 \, dy - \psi_0^2(\alpha) \, .
\end{aligned}$$

Using the equation (17) of the lemma for the first term and the relation (15) of the definition of the digamma function for the second term lead to the final result:

$$\begin{aligned}
\mathbb{V}\mathrm{ar}\left[\log(Y)\right] &= \frac{1}{\Gamma(\alpha)}\Gamma''(\alpha) - \left(\frac{\Gamma'(\alpha)}{\Gamma(\alpha)}\right)^2 \\
&= \frac{d}{d\alpha}\left[\frac{\Gamma'(\alpha)}{\Gamma(\alpha)}\right] = \frac{d^2}{d\alpha^2}\left[\log(\Gamma(\alpha))\right] = \psi_1(\alpha)
\end{aligned}$$

where the last equality is given by the definition of the trigamma function in equation (6).

□

The two propositions above were a key component in the proofs of the theorems in [1], which are studied numerically in Chapter 3. For that reason, the rate parameter β is set to one in the inverse-gamma distribution throughout the rest of this thesis to benefit from these results. The notation:

$$\mathrm{Ga}^{-1}(\alpha) = \mathrm{Ga}^{-1}(\alpha, \beta \coloneqq 1)$$

will be used throughout the rest of this thesis to express this fact.

Lattice and polymer

The lattice is finite and defined through the rectangle $\Lambda_{m,n} \equiv \{0, \dots, m\} \times \{0, \dots, n\}$, which is embedded in the non-negative integer plan $\mathbb{Z}_+^2$.

Remark: Note that by convention $\mathbb{Z}_+ \equiv \{0, 1, 2, \dots\}$ and $\mathbb{N} \equiv \{1, 2, \dots\}$.

Definition 2. [Directed polymer]
A directed polymer evolving inside $\Lambda_{m,n}$ is the sequence $x_\cdot \equiv (x_k)_{0 \le k \le m+n}$, for $x_k \in \mathbb{Z}_+^2$, with its starting point always fixed at $x_0 \coloneqq (0,0)$ and its ending point set at $x_{m+n} \coloneqq (m,n)$. Two successive values can only differ by $e_1 \coloneqq (1,0)$ or $e_2 \coloneqq (0,1)$, meaning that $|x_{k+1} - x_k|_1 = 1$ for all $k \in \{0, 1, \dots, m+n-1\}$.

In other words, a directed polymer encodes an up-right path connecting $(0,0)$ to (m,n) with m displacements to the right and n displacements to the top. For that reason, the word "path" will be used interchangeably with the words "directed polymer" throughout this thesis. The polymer is said to be directed because it can only go up and right but never go back down or left, see Figure 5, as already mentioned in Chapter 1.

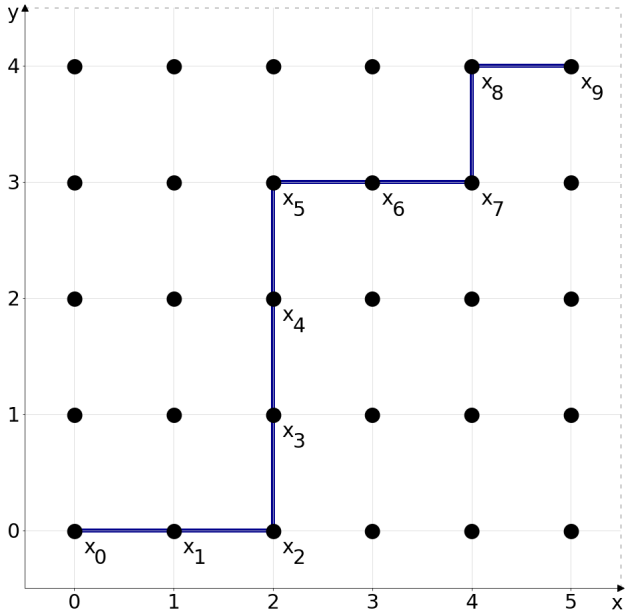


Figure 5: Example of an up-right path from $(0, 0)$ to $(5, 4)$ in $\mathbb{Z}^2_+$.

Weights and partition function

The weights $\{\omega_x : x \in \mathbb{Z}^2_+\}$ that are used within the lattice are i.i.d. random variables coming from an inverse-gamma distribution with a chosen shape parameter α. In other words, for each site's location $x \in \mathbb{Z}^2_+$, a random value is picked from the inverse-gamma distribution.

Once the endpoints are fixed, we can define the weight of a polymer $x.$, which is the product of all the weights it crosses along its path in the lattice:

$$H^{\omega}_{m,n} \equiv \prod_{k=0}^{m+n} \omega_{x_k} \tag{18}$$

with

$$\omega_{i,j}(\alpha) \equiv \omega_{(i,j)}(\alpha), \quad (i,j) \in \Lambda_{m,n} \tag{19}$$

The weight at the origin could be left out of this definition since it contributes to all paths' weights indistinctly. It is kept in this model because it fits how indices are used in most programming languages, moreover, it can be used as a parameter to globally rescale those values and observe the change in the behaviour of the asymptotic limits. Until the numerical results of section 3.2., it can be set to one: $\omega_{0,0} \coloneqq 1$.

Definition 3. [Partition function]
The partition function comprises all possible paths between the endpoints of a lattice $\Lambda_{m,n}$. It is the sum of the weight of all the paths:

$$Z^{\omega}_{m,n}(\alpha) \equiv \sum_{x. \in \Pi_{m,n}} H^{\omega}_{m,n}(x.\,;\alpha) = \sum_{x. \in \Pi_{m,n}} \prod_{k=0}^{m+n} \omega_{x_k}(\alpha) \tag{20}$$

where $\Pi_{m,n}$ denotes the collection of all possible configurations of the polymer $x.$ with endpoints x_0 and x_{m+n} inside the lattice $\Lambda_{m,n}$.

It is worth noting that by this definition, the partition function is a random variable. Through the rest of this thesis, the dependence on ω will be dropped in the notation of the partition function: $Z_{m,n} = Z^{\omega}_{m,n}$.

Remark: Through this definition, it is more visible that the first and last weights, ω_{x_0} and $\omega_{x_{m+n}}$ respectively, play a special role within the calculations since they both contribute in all paths $x_\cdot \in \Pi_{m,n}$. The assignation of a specific value to one of them can rescale all the paths' weights $H_{m,n}$ and therefore the value of $Z_{m,n}$ but with an effect depending on how impactful the value is chosen in comparison to the total number of weights $m+n+1$ that are accounted for the result.

Proposition 3. [Partition function value by iteration]
Let $m, n \in \mathbb{Z}_+$ be fixed. For a polymer $x_\cdot$ on a lattice $\Lambda_{m,n}$ filled with fixed weights $\{\omega_x\}$ from a given distribution, the partition function Z defined as in equation (20) has the following property:

$$Z_{m+1,n+1} = (Z_{m,n+1} + Z_{m+1,n}) \cdot \omega_{m+1,n+1} \tag{21}$$

Proof. Starting from the relation (20) of the partition function's definition, the sum can be split into two parts, one containing the point (m, n+1) and one the point (m+1, n):

$$\begin{aligned} Z_{m+1,n+1} &\equiv \sum_{x_k \in \Pi_{m+1,n+1}} \prod_{k=0}^{m+n+2} \omega_{x_k} \\ &= \underbrace{\omega_{0,0} \dots \omega_{m,n+1}\, \omega_{m+1,n+1} \;+\; \dots \;+\; \omega_{0,0} \dots \omega_{m,n+1}\, \omega_{m+1,n+1}}_{\text{all paths passing through } (m,n+1)} \\ &\quad + \underbrace{\omega_{0,0} \dots \omega_{m+1,n}\, \omega_{m+1,n+1} \;+\; \dots \;+\; \omega_{0,0} \dots \omega_{m+1,n}\, \omega_{m+1,n+1}}_{\text{all paths passing through } (m+1,n)} . \end{aligned}$$

By factorizing the common weight of the endpoint (m+1, n+1) and reuse the relation (20) of the partition function's definition, the equation (21) is obtained:

$$\begin{aligned} Z_{m+1,n+1} &= (\omega_{0,0} \dots \omega_{m,n+1} \;+\; \dots \;+\; \omega_{0,0} \dots \omega_{m,n+1} \\ &\qquad + \omega_{0,0} \dots \omega_{m+1,n} \;+\; \dots \;+\; \omega_{0,0} \dots \omega_{m+1,n}) \cdot \omega_{m+1,n+1} \\ &= (Z_{m,n+1} + Z_{m+1,n}) \cdot \omega_{m+1,n+1} . \end{aligned}$$

□

Binary path representation and growth rate

The previous property will be useful to compute the partition function numerically since the total number of paths $|\Pi_{m,n}|$ from x_0 to x_{m+n} inside a lattice $\Lambda_{m,n}$ grows exponentially. To see this, it is practical to work with a different representation of an up-right path by observing that there is a one-to-one correspondence between a chosen polymer $x_\cdot \in \Pi_{m,n}$ and a (m+n)-uplet of binary elements $\{b_1, b_2\}$, where one of these elements appears m times and the other one n times.

The construction of this bijection is done as follows: by the definition 2 of a directed polymer $x_\cdot \in \Pi_{m,n}$ with $m, n \in \mathbb{Z}_+$ fixed, two successive elements $x_k,\ x_{k+1},\ k \in \{0, \dots, m+n\}$ of the sequence can only differ either by $e_1 := (1,0)$ or by $e_2 := (0,1)$. Since the starting point x_0 is always

the origin, the sequence $(i_k)_{1\leq k\leq m+n}$ with $i_k \in \{1,2\}$ allows the following iterative construction for any directed polymer:

$$\begin{aligned} x_1 &= x_0 + e_{i_1} &&\doteq (b_{i_1}) \\ x_2 &= x_1 + e_{i_2} &&\doteq (b_{i_1}, b_{i_2}) \\ x_3 &= x_2 + e_{i_3} &&\doteq (b_{i_1}, b_{i_2}, b_{i_3}) \\ &\vdots && \vdots \\ x_{m+n} &= x_{m+n-1} + e_{i_{m+n}} &&\doteq (b_{i_1}, b_{i_2}, b_{i_3}, \ldots, b_{i_{m+n}}) \end{aligned}$$

where the symbol " $\doteq$ " means an equality between two different representations.

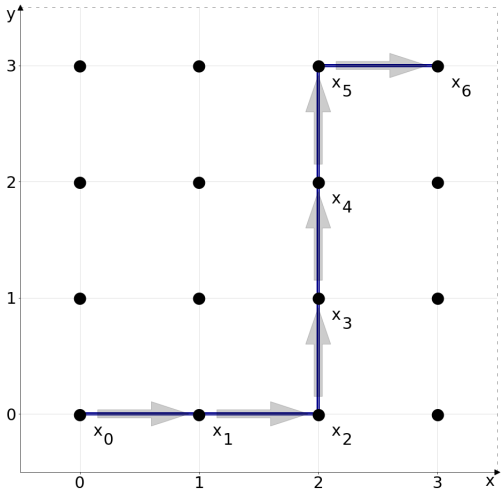
This creates a one-to-one correspondence between all the e_{i_k} and the b_{i_k}, which confirms the bijection between a directed polymer $x_\cdot \in \Pi_{m,n}$ and an $(m+n)$-uplet of binary elements.

Examples: Denoting the binary elements as right and up arrows $\{b_1 := \rightarrow,\ b_2 := \uparrow\}$, it is therefore possible to write:

$$x_\cdot \doteq (\underbrace{\rightarrow, \ldots, \rightarrow}_{m \text{ times}}, \underbrace{\uparrow, \ldots, \uparrow}_{n \text{ times}})$$

for a path passing through the bottom right corner of the lattice.

This can be done for any path $x_\cdot \in \Pi_{m,n}$:



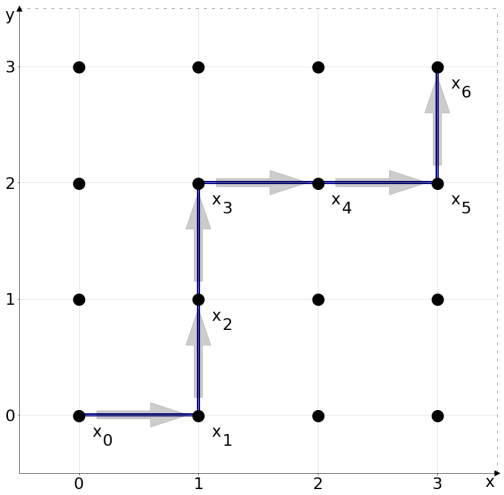
(a) a possible path from $(0,0)$ to $(3,3)$.

This path $x_\cdot^{(a)}$ on the left is given by:

$$\begin{aligned} x_\cdot^{(a)} &= ((0,0),\ (1,0),\ (2,0), \\ &\qquad (2,1),\ (2,2),\ (2,3),\ (3,3)) \\ &\doteq (\rightarrow, \rightarrow, \uparrow, \uparrow, \uparrow, \rightarrow) \end{aligned}$$

and the path $x_\cdot^{(b)}$ on the right by:

$$\begin{aligned} x_\cdot^{(b)} &= ((0,0),\ (1,0),\ (1,1), \\ &\qquad (1,2),\ (2,2),\ (3,2),\ (3,3)) \\ &\doteq (\rightarrow, \uparrow, \uparrow, \rightarrow, \rightarrow, \uparrow)\,. \end{aligned}$$



(b) another path from $(0,0)$ to $(3,3)$.

This binary representation of a directed polymer is useful to prove the following proposition:

Proposition 4. [Total number of paths in a lattice]
Let $m, n \in \mathbb{Z}_+$ be fixed. The total number of paths $|\Pi_{m,n}|$ in a lattice $\Lambda_{m,n}$ is given by the relation:

$$|\Pi_{m,n}| = \binom{m+n}{m} = \frac{(m+n)!}{m! \cdot n!} = \binom{m+n}{n} \tag{22}$$

Proof. For $m, n \in \mathbb{Z}_+$ fixed and by the previous bijection, all possible paths in the lattice $\Lambda_{m,n}$ can be represented by $(m+n)$-uplets of binary elements. With an illustrative choice of $b_1 \coloneqq \rightarrow$, $b_2 \coloneqq \uparrow$ for these binary elements and an empty path denoted as:

$$x_{\cdot}^{\star} \doteq (\underbrace{\star, \star, \ldots, \star}_{m+n \text{ times}}), \tag{23}$$

all the paths $x_{\cdot} \in \Pi_{m,n}$ in the lattice $\Lambda_{m,n}$ can be obtained by converting m of the $(m+n)$ empty $\star$ values to right arrows $\rightarrow$ and the remaining n empty $\star$ values to up arrows $\uparrow$.
Therefore, the total number of paths is:

$$|\Pi_{m,n}| = \binom{m+n}{m}$$

and since

$$\binom{m+n}{m} \equiv \frac{(m+n)!}{m! \cdot (m+n-m)!} = \frac{(m+n)!}{m! \cdot n!} = \frac{(m+n)!}{(m+n-n)! \cdot n!} \equiv \binom{m+n}{n}$$

the proof is complete. Alternatively, it is possible to start with the empty path $x_{\cdot}^{\star}$ and first convert n of the $(m+n)$ empty $\star$ values to up arrows $\uparrow$ and then change the remaining m empty $\star$ values to right arrows $\rightarrow$, therefore giving:

$$|\Pi_{m,n}| = \binom{m+n}{n}$$

which is coherent with the previous results.

□

Corollary 1. In the particular case of an endpoint along the diagonal $m \coloneqq n$, the asymptotic limit of the ratio of the total number of paths for an expanding lattice is given by:

$$\lim_{n \to \infty} \frac{|\Pi_{n+1,n+1}|}{|\Pi_{n,n}|} \;=\; 4 \tag{24}$$

Proof. Using the proposition 4, the ratio of the total number of paths for an expanding square lattice is:

$$\frac{|\Pi_{n+1,n+1}|}{|\Pi_{n,n}|} \;=\; \frac{(2n+2)!}{[(n+1)!]^2} \cdot \frac{(n!)^2}{(2n)!} \;=\; \frac{(2n+2)(2n+1)}{(n+1)^2} \;=\; \frac{4n^2+6n+2}{n^2+2n+1}\,.$$

which leads to the equation (24) by taking the limit of $n \longrightarrow \infty$.

□

This indicates that the total number of paths tends to grow exponentially with a growth rate of 4 along the diagonal in the limit when n grows larger, see Figure 6. This information is essential computation-wise since it is important to program an algorithm that takes the expansion into account.

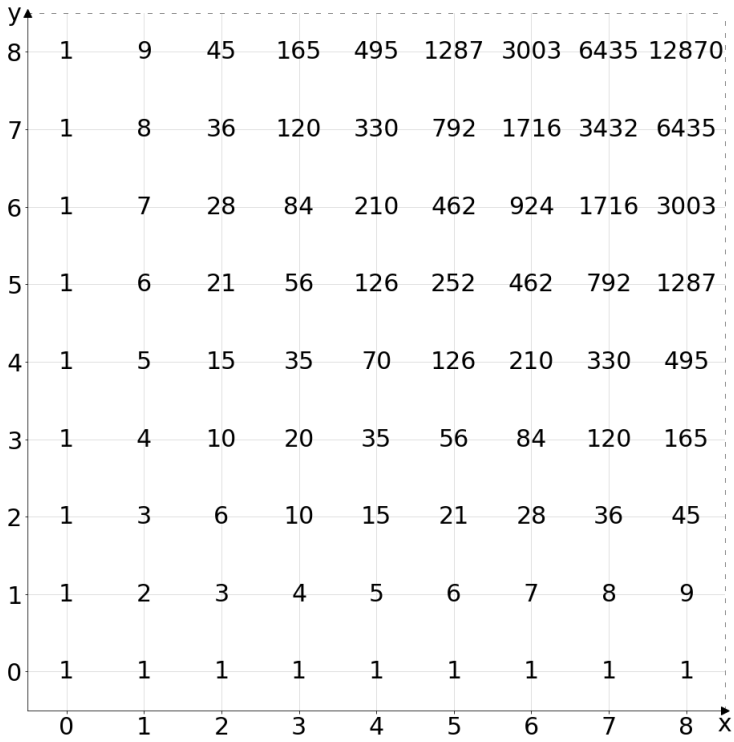


Figure 6: The total number of paths in a square lattice as a function of the endpoint $(m, n) \in \mathbb{Z}_+^2$.

Two corollaries are added below because of their link with the algorithm design discussed in the next chapter.

Corollary 2. The total number of paths is symmetric under the inversion of m and n:

$$|\Pi_{m,n}| \;=\; |\Pi_{n,m}| \tag{25}$$

Proof. This follows immediately from the relation (22).

□

Corollary 3. In the particular case of an endpoint along the diagonal $m \coloneqq n$ with $n \in \mathbb{N}$, the total number of paths is even and respects the following relation:

$$|\Pi_{n,n}| \;=\; |\Pi_{n-1,n}| \;+\; |\Pi_{n,n-1}| \;=\; 2\,|\Pi_{n-1,n}| \tag{26}$$

Proof. Since the total number of paths in the lattice are binomial values, this property seems obvious by knowing the link between those and Pascal's triangle. Nonetheless, it is not a lot of work to demonstrate it rigorously. By the relations (25) and (22) with $m \coloneqq n$, it follows that

$$
\begin{aligned}
|\Pi_{n-1,n}| \;+\; |\Pi_{n,n-1}| \;&=\; |\Pi_{n-1,n}| \;+\; |\Pi_{n-1,n}| \\
&=\; 2\,|\Pi_{n-1,n}| \\
&=\; 2\binom{2n-1}{n} \\
&=\; 2\,\frac{(2n-1)!}{(n-1)!\,n!} \\
&=\; 2\,\frac{n\,(2n-1)!}{n\,(n-1)!\,n!} \\
&=\; \frac{(2n)!}{(n!)^2} \\
&=\; \binom{2n}{n} \\
&=\; |\Pi_{n,n}|
\end{aligned}
$$

where the relation with the second line proves that the total number of paths $|\Pi_{n,n}|$ is even.

□

Height function and free energy

As discussed in the introduction, the directed polymer model is linked to other random growth processes through its definition of the height function, which here is given by $h(m,n) = \log(Z_{m,n})$, that is, the equivalent of (minus) the free energy in statistical mechanics with the inverse temperature β set to one, see the relation (2). All the results obtained before are therefore related to the height function, which is linked to the Kardar-Parisi-Zhang-universality class by the theorems of the following section.

2.2. Known asymptotic results of the model

The objective of this thesis is to observe numerically two of the theorems that were demonstrated in [1] and restated in [2], which are a description of probability distribution of the free energy. Also studied in this thesis, the result given by the first theorem demonstrated in [2], which is the strongest result of the three because it encompasses the results from the two previous ones by describing the resulting distribution of the fluctuations of the free energy at a large scale.

Theorems 1 and 2

This first theorem corresponds to the law of large numbers of the free energy (Theorem 2.4. [1]). The proof used the stationary boundary conditions defined in the previous section, then the result was proven almost surely equivalent in the law of large numbers to the model without boundaries.

This theorem describes how the asymptotic behaviour of the scaled free energy converges almost surely in probability for a 1+1 directed polymer $x.$ on a lattice $\Lambda_{n,n}$, $n \in \mathbb{N}$, with i.i.d. random variables coming from an inverse-gamma distribution of shape parameter α: $\{\omega_{i,j}\} \sim Ga^{-1}(\alpha)$. It is written here under in its simpler notation form used in [2]:

Theorem 1. [Convergence of the scaled free energy in the thermodynamic limit]

$$\lim_{n\to\infty} \frac{\log(Z_{n,n})}{n} = -2\,\psi_0\left(\frac{\alpha}{2}\right) \qquad \mathbb{P}\text{-a.s.} \tag{27}$$

The second theorem gives an upperbound to the scaled variance of the free energy in the thermodynamic limit:

Theorem 2. [Scaled variance upper-bound in the thermodynamic limit]
There exists a constant $\mathrm{C} > 0$ such that

$$\limsup_{n\to\infty} \left\{ \frac{\mathbb{V}\mathrm{ar}\left[\log(Z_{n,n})\right]}{n^{2/3}} \right\} \leq \mathrm{C}\,. \tag{28}$$

Theorem 3 and the Tracy-Widom distribution

The Tracy-Widom distribution is a probability distribution that describes the statistical behaviour of the largest eigenvalue of certain types of random matrices, which are mostly categorized into the three following ensembles:

- Gaussian Orthogonal Ensemble [GOE] ($\beta = 1$), which are real symmetric matrices with i.i.d. samples from the standard normal distribution ;
- Gaussian Unitary Ensemble [GUE] ($\beta = 2$), which are complex Hermitian matrices with complex Gaussian entries ;
- Gaussian Symplectic Ensemble [GSE] ($\beta = 4$), which are Hermitian quaternionic matrices.

The β value is their corresponding Dyson index, which counts the number of real components per matrix element.

This distribution, which is shown in Figure 7, is an asymmetric probability distribution introduced by Craig Tracy and Harold Widom (1993) in the context of random matrix theory and it plays a fundamental role in describing universal properties of stochastic processes and disordered systems. It is the crossover function between the two phases of weakly versus strongly coupled components in a system and appears in various contexts but notably in the large-scale statistics of the Kardar-Parisi-Zhang equation (4) mentioned in Chapter 1.

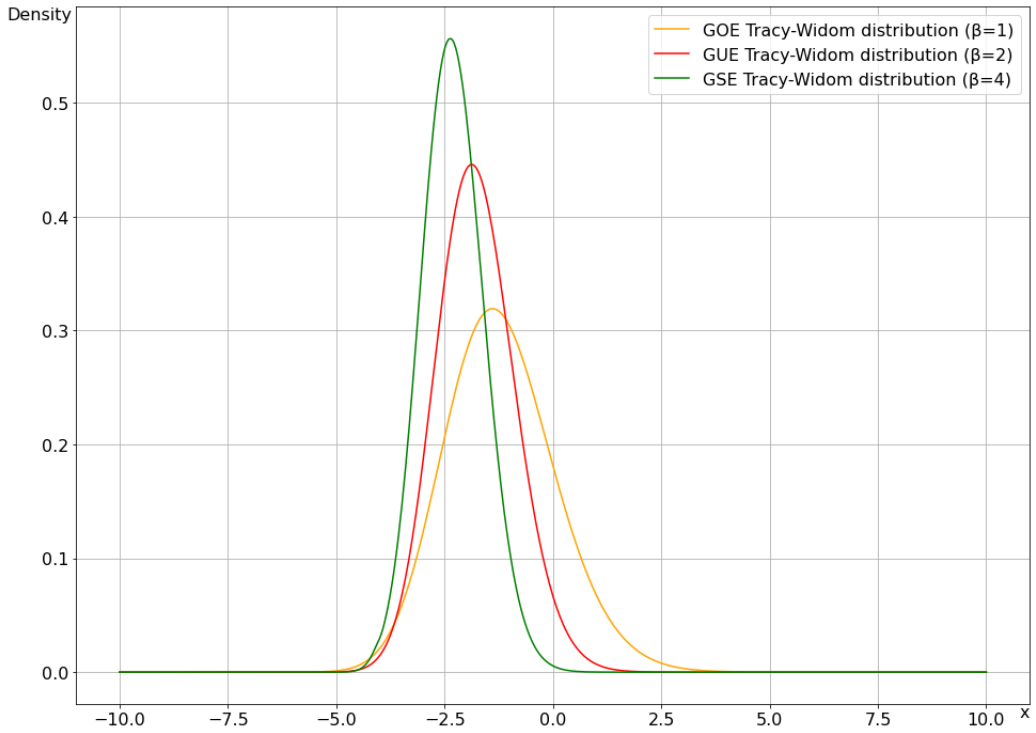


Figure 7: Tracy-Widom distribution for β=1, 2 and 4.

The cumulative distribution function [CDF] F_β of the Tracy–Widom distribution for a given β is seen as a law of large numbers, similar to the central limit theorem:

$$F_{\text{GUE}}(s) = F_2(s) = \lim_{N\to\infty} \mathbb{P}\left[(\lambda_{N,\max} - \sqrt{4N})N^{1/6} \leq s\right] \tag{29}$$

for the example of $N \times N$ Hermitian matrices with off-diagonal variance 1, where $\lambda_{N,\max}$ denotes the largest eigenvalue of the random matrix. A more formal definition of F_{GUE} is given by the expression:

$$F_2(s) \equiv e^{-\int_s^\infty (u-s)\, q^2(u)\, du} \tag{30}$$

where the function q is called a "Painlevé transcendent" and is the unique solution to the following Painlevé II equation:

$$q''(x) \;=\; xq(x) + 2q^3(x) \;\text{ with }\; \begin{cases} q(x) \;\sim\; -\frac{x^{-\frac{1}{4}}}{2\sqrt{\pi}}\, e^{-\frac{2}{3}x^{\frac{3}{2}}} & \text{when } x \to +\infty \\ q(x) \;\sim\; \sqrt{\frac{|x|}{2}} & \text{when } x \to -\infty \end{cases}, \tag{31}$$

which makes the equation (30) a distribution since:

$$F_2(-\infty) = 0, \quad F_2(+\infty) = 1, \quad F'_2(s) > 0 \text{ for } s \in \mathbb{R}\,. \tag{32}$$

Similar formulations exist for the two other ensembles and the reader eager to know more about the mathematical aspects of this distribution may find the article [16] interesting.

The Tracy-Widom distribution has profound implications in various fields and is particularly relevant in this context of the 1+1 directed polymer model as it characterizes the fluctuations of the free energy in the KPZ universality class, as the next theorem shows:

Theorem 3. [Fluctuations of the free energy]

$$\lim_{n\to\infty} \mathbb{P}\left[\frac{\log(Z_{n,n}) + 2n\,\psi_0\left(\frac{\alpha}{2}\right)}{n^{1/3}} \leq r\right] = F_{\text{GUE}}\left(\left(-\psi_2\left(\frac{\alpha}{2}\right)\right)^{-\frac{1}{3}} \cdot\, r\right) \tag{33}$$

This third theorem is the strongest result since it encompasses both the behaviour of the average variance and the fluctuations. It shows that the fluctuations distribution of the polymer's free energy is given by the Tracy-Widom distribution in the asymptotic limit, which can be thought of an equivalent of the central limit theorem.

In order to use this last theorem numerically, it is required to define:

$$s = \left(-\psi_2\left(\frac{\alpha}{2}\right)\right)^{-\frac{1}{3}} \cdot\, r \tag{34}$$

and rewrite the relation (33) as

$$\lim_{n\to\infty} \mathbb{P}\left[\frac{\log(Z_{n,n}) + 2n\,\psi_0\left(\frac{\alpha}{2}\right)}{\left(-\psi_2\left(\frac{\alpha}{2}\right)\, n\right)^{\frac{1}{3}}} \leq s\right] = F_{\text{GUE}}\,(s) \tag{35}$$

Remark: The sign of the polygamma function defined in (6) is given by $\epsilon(\psi_m) = (-1)^{m+1}$.

The rigorous proofs of these theorems are given in the references mentioned at the beginning of the section and aren't incorporated in this thesis.

Studied functions

The three independent functions T_n^1, T_n^2 and T_n^3 related to these theorems that need be studied numerically are therefore:

$$T_n^1(\alpha) \;=\; \frac{\log(Z_{n,n}(\alpha))}{n} \tag{36}$$

$$T_n^2(\alpha) \;=\; \frac{\mathbb{V}\text{ar}\left[\log(Z_{n,n}(\alpha))\right]}{n^{2/3}} \tag{37}$$

$$T_n^3(\alpha) \;=\; \frac{\log(Z_{n,n}(\alpha)) + 2n\,\psi_0\left(\frac{\alpha}{2}\right)}{\left(-\psi_2\left(\frac{\alpha}{2}\right)\, n\right)^{\frac{1}{3}}} \tag{38}$$

Remark: It is worth noting that while Theorem 2 is deterministic, Theorems 1 and 3 describe a probabilistic behaviour at a large scale. Therefore the asymptotic limit $n \to \infty$ of the functions T_n^i cannot be seen as a replacement of these results. They are the studied functions in order to relate the numerical outcomes to the analytic ones.

Slight adaptation of the theorems

Since the previous results given by these three theorems correspond to an equivalent model that starts the path from the position $(1, 1)$ instead of $(0, 0)$, therefore performing $m + n - 2$ displacements, it is necessary to compensate for that change within the algorithm. This is done directly in the code by the transformation $m \mapsto m + 1$ for any $m \in \mathbb{Z}_+$ during the use of the indices. The algorithm is given at the end of this document and the reader can verify that correction.

Now that the mathematical aspects of the model are established, the numerical portion of this study can be presented.

3. Numerical Study

In this chapter, a more technical aspect of the creation of an algorithm capable of confronting the theorems given in the previous chapter is described before showing the numerical results obtained.

In the first section, two algorithms are presented because, at the start of this thesis, a design choice made me go towards an approach that was seemingly more interesting in terms of details offered but was in the end not sufficient to illustrate the theorems because of a slow convergence rate. This is why a second algorithm was developed to analyse the veracity of the theorems through numerical results, but at the cost of a great loss of information about the paths themselves, e.g. how they contribute individually in comparison to the partition function.
The first algorithm is nevertheless a contribution of my work and therefore presented in this thesis although it would probably be more suitable for other study cases.

The second section shows the numerical results obtained with the second algorithm. Along with the figures generated by the code, observations are made in order to discuss the numerical results obtained and compare them to the analytical ones.

3.1. Implementation of the algorithms

This section is a technical one for which the link with the logic of computer programs is noticeable. It is possible to skip it entirely and still understand the following numerical results in the next section.

Choice between parallel and iterative constructions

The objective is to calculate the value of the partition function $Z_{n,n}$ in a square lattice $\Lambda_{n,n}$ of size n. To do that, two distinct approaches are possible:

1. Through a direct computation of the partition function via its definition (20):

$$Z_{m,n} \equiv \sum_{x_\cdot \in \Pi_{m,n}} H_{m,n}(x_\cdot) = \sum_{x_\cdot \in \Pi_{m,n}} \prod_{k=0}^{m+n} w_{x_k}$$

2. Or with the recurrence property given in equation (21):

$$Z_{m+1,n+1} = (Z_{m,n+1} + Z_{m+1,n}) \cdot w_{m+1,n+1}$$

The first choice is a parallel approach. It computes explicitly all the paths from x_0 to x_{2n} independently from one another and sums them up to obtain the partition function value. This approach keeps track of all of the information given by the paths, which means that it would be possible to look at the path that contributed the most, take the maximum value between all of these paths to make a relation to the corner growth model, or even draw a map over the lattice that would show the privileged trajectory of the directed polymer as introduced by [1] in the terms of characteristic direction.

The second approach is an iterative one with the relation (21) as the central piece of the algorithm that performs the calculations. It is simpler to code but at the cost of all the details within the paths' weights $\{H_{n,n}(x.) : x. \in \Pi_{n,n}\}$. The evaluation of the partition function $Z_{n,n}$ relies on the previous values of it in the lattice but those cannot be kept as a result in the numerical study since all these values are linked to the same set of distributed weights during the iterative calculation, they are therefore not independent random variables in regards of each other.

Usually, when given the choice between the possibility to work with a parallel design over an iterative one for which the algorithm relies on subsequent values, parallelism is chosen because it means that calculations can easily be ported to the GPU (Graphics Processing Unit), which is faster than the CPU (Central Processing Unit) in most cases.
Unfortunately for me, I made this design choice even if I already knew that the total number of paths in a lattice grows exponentially with rate 4, thanks to the result demonstrated by the relation (24) because if the convergence related to the first theorem was to be quick enough relative to the value of n, this would have been the best choice in order to perform a lot of iterations for a fixed lattice size thanks to a GPU port in addition to preserve all of the information contained within the paths.
Sciences are also about trial and error and it is important to share what could profit someone else, even if the result wasn't the one expected. I present the first algorithm design for which I worked for a couple of months, then I describe the iterative approach that was more reliable for this study case because of the necessity to expand the lattice at a large size ($n \sim 200$).

First algorithm: a parallel approach

Thanks to the partition function defined in (20), which is the sum of all the paths' weights $H_{n,n}$ evaluated separately, the advantage of this design is that once the paths for a fixed value of n are known, they don't need to be calculated anymore if they are stored. Hence, for each iteration with a different set of distributed weights, the calculations of the weights of all paths are done with a speed proportional to the total number of paths and are performed independently from one another. Therefore, the most crucial aspect of the construction of this algorithm is how to find and register all these possible paths.
I like to think that I succeeded in creating this important part of the algorithm and to prevent anyone from devoting the same time I did, this is described below.

Two important aspects need to be thought of and built jointly:

1. How to move from the current position that multiplied the respective weight to the next position?

2. How to store the path and how to read it afterwards?

To do so, the binary representation described in section 2.1. is really useful.
By setting $b_1 \coloneqq 0$ and $b_2 \coloneqq 1$ with the interpretation of zero to be False and one to be True, any path can be stored as an array of boolean values as, for example:

$$x. \doteq (0, 1, 0, 0, \dots, 1, 0, 1, 1) \tag{39}$$

and then be used in the algorithm to decide between going right or upwards within the lattice:

$$\text{Reading Logic: } \begin{cases} \text{IF } 0 : \text{ go right} \\ \text{ELSE: go up} \end{cases} \tag{40}$$

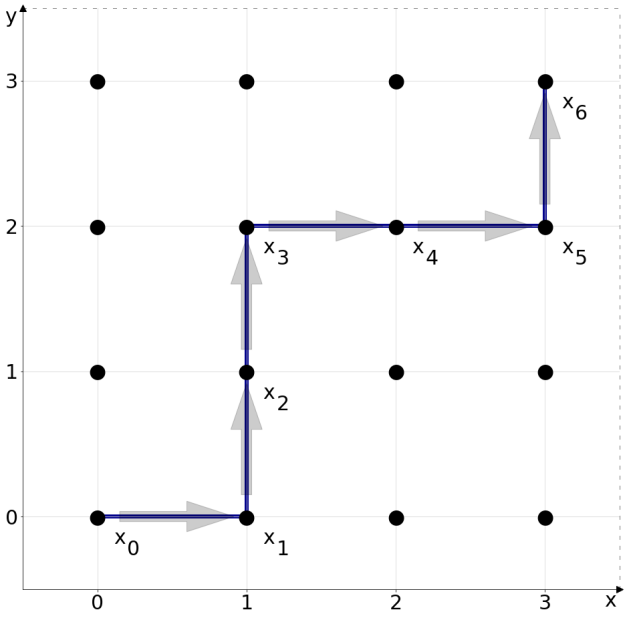
Therefore, the two questions can be answered by using an array of boolean values. This is already sufficient enough to allow the evaluation of all the paths' weights $(H_{i,j})_{0\leq i,j\leq n}$ and therefore the partition function $Z_{n,n}$.

Before describing the more difficult part, which is the construction of those arrays, some improvements can be made to help with the data storage under the assumption that only endpoints on the diagonal are considered.
Since the total number of paths ending along the diagonal is always an even number, thanks to the property (26) for $m \coloneqq n$, there is the same number of zeros in the binary array as the number of ones in it. Therefore, it is possible to represent all the paths from x_0 to x_{2n} in a square lattice $\Lambda_{n,n}$ by integer values and vice versa. To do so, it suffices to take the values of the binary array as coefficients in a series of power of twos:

$$x. \;\doteq\; (b_{i_1},\, b_{i_2},\, b_{i_3},\, \ldots,\, b_{i_{2n}}) \;\doteq\; \sum_{k=1}^{2n} b_{i_k}\, 2^{2n-k}$$

with $i_k \in \{1, 2\}$ for $k \in \{1, 2, \ldots, 2n\}$ and $\{b_1 \coloneqq 0,\, b_2 \coloneqq 1\}$.

For example, consider the following path from $(0, 0)$ to $(3, 3)$ for $n \coloneqq 3$ on the adjacent image:



This path corresponds to:

$$x. \;=\; ((0,0),\,(1,0),\,(1,1),\,(1,2),\,(2,2),\,(3,2),\,(3,3))$$

$$\doteq\; (\rightarrow, \uparrow, \uparrow, \rightarrow, \rightarrow, \uparrow) \qquad \text{with } b_1 \coloneqq \rightarrow,\; b_2 \coloneqq \uparrow$$

$$\doteq\; (0,\,1,\,1,\,0,\,0,\,1) \qquad \text{with } b_1 \coloneqq 0,\; b_2 \coloneqq 1$$

$$\doteq\; 0 \cdot 2^5 \,+\, 1 \cdot 2^4 \,+\, 1 \cdot 2^3 \,+\, 0 \cdot 2^2 \,+\, 0 \cdot 2^1 \,+\, 1 \cdot 2^0 \;=\; 25$$

The words "integer representation of a path" will be shortened to "integer path" throughout the rest of this thesis.

To go from an integer value to the binary representation of a path, it is enough to express that integer into a series of power of twos, retrieve their coefficients into an array, and then complete the missing zeros on the left of the array if they are strictly less than the number of ones. For example:

$$11 \;=\; 1 \cdot 2^3 \;+\; 0 \cdot 2^2 \;+\; 1 \cdot 2^1 \;+\; 1 \cdot 2^0$$

gives (1,0,1,1) with $n = 3$ ones and $m = 1$ zero, and is completed to the array (0,0,1,0,1,1) to match a path ending along the diagonal in $\Lambda_{3,3}$. The binary representation equivalent to the integer path of value 11 is therefore complete and unique. The uniqueness is obtained by the fact that no two integers have the same series expansion of power of twos, and by the construction that only adds zeros on the left, leaving the integer series expansion intact, until their number matches the numbers of ones, meaning that no two paths can emerge from a given series of power of twos.

Remark: The correspondence of this integer representation of paths with the binary representation is only true for integers from a restricted subset of $\mathbb{N}$. Indeed, some integers have no valid binary representation of a path by this construction because they start with the number of zeros strictly greater than the number of ones. For example:

$$8 \;=\; 1 \cdot 2^3 \;+\; 0 \cdot 2^2 \;+\; 0 \cdot 2^1 \;+\; 0 \cdot 2^0$$

produces the array (1,0,0,0), which cannot be expanded to a path with an endpoint on the diagonal without adding ones, hence changing the corresponding value of the integer that is given.

Instead of storing long binary arrays of $2n$ elements, it is therefore possible to store integer values which are more (human) readable and take less space. Another way to improve the data storage is to notice that only half of the total number of paths are needed to make the algorithm work for an endpoint along the diagonal.
It is indeed noticeable with a close look at the two following paths:

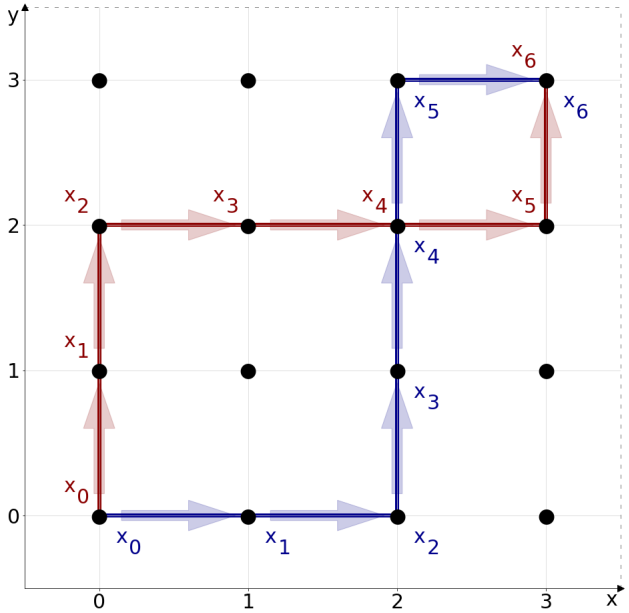


Figure 8: Two conjugate paths from $(0, 0)$ to $(3, 3)$ for $n \coloneqq 3$.

and their binary representations:

$$\text{blue path} \doteq (0,0,1,1,1,0) \qquad ; \qquad \text{red path} \doteq (1,1,0,0,0,1).$$

The blue path is the unique path obtained by the interchange of 0 and 1 values in the red path's binary representation. Two paths related to this transformation will be said to be "conjugate paths" of each other throughout the rest of this thesis.
By their integer representations, it is also noticeable that two conjugate paths are always of different parity because the conjugation flips the value of the last binary element $b_{i_{2n}}$, which represents the parity of an integer path.

This, nonetheless, can only be true along the diagonal when $m \coloneqq n$ because conjugate paths are the result of both equations (25) and (26), rewritten here:

$$\begin{aligned} |\Pi_{m,n}| &= |\Pi_{n,m}|\,, & n,m \in \mathbb{Z}_+ \\ |\Pi_{n,n}| &= |\Pi_{n-1,n}| + |\Pi_{n,n-1}|\,, & n \in \mathbb{N} \end{aligned}$$

meaning that the total number of paths in a square lattice $\Lambda_{n,n}$ with $n \in \mathbb{N}$ is evenly distributed between the total number of paths passing by $x_{2n-1} = (n-1,n)$ from the left of the endpoint, and the total number of the paths passing by $x_{2n-1} = (n,n-1)$ from the bottom of the endpoint, both equal in size.
These two relations therefore allow the notion of conjugate paths between paths from the two distinct subsets $\Pi_{n-1,n}$ and $\Pi_{n,n-1}$ for which the separation:

$$\left[(b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}\,,\,0) \in \Pi_{n,n} \iff x_{2n-1} = (n-1,n)\right] \implies (b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}) \in \Pi_{n-1,n}$$

$$\left[(b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}\,,\,1) \in \Pi_{n,n} \iff x_{2n-1} = (n,n-1)\right] \implies (b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}) \in \Pi_{n,n-1}$$

is clear in term of the last element $b_{i_{2n}}$ in the binary array representations.

In summary, the data storage can be reduced to stock only odd integer paths, that is the paths coming from the bottom of the endpoint with a binary representation ending with 1, and the logic (40) can be read twice in the algorithm where the second reading inverse the logic output, meaning that the conjugate path of the one given is treated.

$$\text{First Reading Logic: } (b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}\,,\,1) \longrightarrow \begin{cases} \text{IF } 0: \text{ go right} \\ \text{ELSE: go up} \end{cases}$$

$$\text{Second Reading Logic: } (b_{i_1}\,,\,b_{i_2}\,,\,\ldots\,,\,b_{i_{2n-1}}\,,\,1) \longrightarrow \begin{cases} \text{IF } 0: \text{ go up} \\ \text{ELSE: go right} \end{cases}$$

Now that the use of binary arrays as indicators of the path to take and the storage of those in terms of integers are set up, the next question is how to produce all the possible paths for a fixed square lattice $\Lambda_{n,n}$ in order to obtain and sum up all paths' weights $(H_{i,j})_{0\leq i,j\leq n}$.
The obvious way would be to perform all permutations given a starting binary representation of a

path. Unfortunately, this is computationally heavy and even with the best tools designed for that purpose only, it cannot be done on most computers starting at values greater than $n \coloneqq 7$ (binary array of 14 elements), which is a strong limitation. In the end, based on what was defined above, I managed to construct all possible paths with my second idea, which is described further.

My first attempt was indeed a naive approach of finding all the paths independently of each other by using their correspondence with an integer value. For any given n in a square lattice, the minimal and maximal integer paths can be found by the binary representations of the bottom-right corner path and the top-left corner path respectively:

$$\text{bottom-right corner path} \doteq (\underbrace{0,\, 0,\, \ldots,\, 0}_{n \text{ times}},\, \underbrace{1,\, 1,\, \ldots,\, 1}_{n \text{ times}}) \doteq \sum_{k=1}^{n} 2^{n-k}$$

$$\text{top-left corner path} \doteq (\underbrace{1,\, 1,\, \ldots,\, 1}_{n \text{ times}},\, \underbrace{0,\, 0,\, \ldots,\, 0}_{n \text{ times}}) \doteq \sum_{k=1}^{n} 2^{2n-k}\,.$$

The idea was to check for the validity of the binary representations coming from integer values between these bounds and store the correct paths that have the same number of zeros as ones. The problem was the number of integers to check:

$$\left|\left[\sum_{k=1}^{n} 2^{n-k}\,,\, \sum_{k=1}^{n} 2^{2n-k}\right]\right| \;=\; \sum_{k=1}^{n} \left(2^{2n-k} - 2^{n-k}\right) \;=\; 2^{2n} \sum_{k=1}^{n} \left(2^{-k} - 2^{-n-k}\right),$$

which grew too fast in comparison to the total number of paths $\Pi_{n,n}$ along the diagonal, given by equation (22), as a quick code implementation showed: see Figure 9 below.

```
For a path of length 2x1 the 2 searched values are to be found in |[1, 2]| = 2 integers.
For a path of length 2x2 the 6 searched values are to be found in |[3, 12]| = 10 integers.
For a path of length 2x3 the 20 searched values are to be found in |[7, 56]| = 50 integers.
For a path of length 2x4 the 70 searched values are to be found in |[15, 240]| = 226 integers.
For a path of length 2x5 the 252 searched values are to be found in |[31, 992]| = 962 integers.
For a path of length 2x6 the 924 searched values are to be found in |[63, 4032]| = 3970 integers.
For a path of length 2x7 the 3432 searched values are to be found in |[127, 16256]| = 16130 integers.
For a path of length 2x8 the 12870 searched values are to be found in |[255, 65280]| = 65026 integers.
For a path of length 2x9 the 48620 searched values are to be found in |[511, 261632]| = 261122 integers.
For a path of length 2x10 the 184756 searched values are to be found in |[1023, 1047552]| = 1046530 integers.
For a path of length 2x11 the 705432 searched values are to be found in |[2047, 4192256]| = 4190210 integers.
For a path of length 2x12 the 2704156 searched values are to be found in |[4095, 16773120]| = 16769026 integers.
For a path of length 2x13 the 10400600 searched values are to be found in |[8191, 67100672]| = 67092482 integers.
For a path of length 2x14 the 40116600 searched values are to be found in |[16383, 268419072]| = 268402690 integers.
```

Figure 9: The fast raising of integer's number between the minimal and maximal bounds for a given n.

Even if the search was to be done only once, this method, or a variant one using only odd integers, seemed unreliable for square lattices of really large sizes. I therefore came up with a better way of finding these odd integer paths by a more targeted search:

Since the total number of paths given a position (m,n) is the sum of the total number of paths from the bottom and left positions, see Figure 6, the idea to use integer paths from the previous square lattice size came to my mind and, after a few weeks of work, I developed an iterative process based on that. Let $\mathcal{I}_n^{\text{odd}}$ denote the ensemble of odd integer paths, for example:

$$\mathcal{I}_2^{\text{odd}} = \{3,\, 5,\, 9\} \doteq \{(0,0,1,1),\, (0,1,0,1),\, (1,0,0,1)\}$$

is the ensemble of all paths coming from the bottom position of $(2,2)$ in a square lattice $\Lambda_{2,2}$. The iterative process is in two steps and is demonstrated here for the construction of

$$\mathcal{I}_3^{\text{odd}} = \{7,\ 11,\ 13,\ 19,\ 21,\ 25,\ 35,\ 37,\ 41,\ 49\}$$

through the reading of $\mathcal{I}_2^{\text{odd}}$.

Step 1: (00)-expansion. To go from the set of paths $\Pi_{n,n}$ to the next one $\Pi_{n+1,n+1}$, two entries need to be added in the binary array, a 0 and a 1. By the expansion of the binary representation of the previous integer paths, it is possible to obtain the new integer paths at a low calculation cost. The first step consists of inserting two zeros at the left of the binary representation of the previous integer paths and then selecting as a result, almost all possible ways to convert a zero to a one:

$$3 \doteq (0,0,1,1) \overset{(00)}{\longmapsto} (0,0,0,0,1,1) \overset{0\mapsto 1}{\longmapsto} \begin{bmatrix} (1,0,0,0,1,1) \doteq 35 \\ (0,1,0,0,1,1) \doteq 19 \\ (0,0,1,0,1,1) \doteq 11 \\ (0,0,0,1,1,1) \doteq 7 \end{bmatrix}$$

$$5 \doteq (0,1,0,1) \overset{(00)}{\longmapsto} (0,0,0,1,0,1) \overset{0\mapsto 1}{\longmapsto} \begin{bmatrix} (1,0,0,1,0,1) \doteq 37 \\ (0,1,0,1,0,1) \doteq 21 \\ (0,0,1,1,0,1) \doteq 13 \\ (0,0,0,1,1,1) \doteq 7 \end{bmatrix}$$

$$9 \doteq (1,0,0,1) \overset{(00)}{\longmapsto} (0,0,1,0,0,1) \overset{0\mapsto 1}{\longmapsto} \begin{bmatrix} (1,0,1,0,0,1) \doteq 41 \\ (0,1,1,0,0,1) \doteq 25 \\ (0,0,1,1,0,1) \doteq 13 \\ (0,0,1,0,1,1) \doteq 11 \end{bmatrix}$$

One can see that only zeros on the left side of the most-left 1 value in the binary array need to be transformed in order to prevent multiple appearances of the same number, here denoted in orange. Most importantly, this step can be performed directly on integers by simply adding powers of two to the initial integer path accordingly, that is, only powers strictly greater than the highest contained in that integer and strictly lower than $2n$:

$$\begin{aligned} 3 + 2^2 = 7 \ ;\ 3 + 2^3 = 11 \ ;\ 3 + 2^4 = 19 \ ;\ 3 + 2^5 = 35 \\ 5 + 2^3 = 13 \ ;\ 5 + 2^4 = 21 \ ;\ 5 + 2^5 = 37 \\ 9 + 2^4 = 25 \ ;\ 9 + 2^5 = 41 \end{aligned}$$

This step produces $3 \times \binom{2n-3}{n-1}$ odd integer paths of those contained in $\mathcal{I}_n^{\text{odd}}$ for a square lattice $\Lambda_{n,n}$ of size $n \geq 2$.

Step 2: (11)-expansion. There is still one missing element in comparison to $\mathcal{I}_3^{\text{odd}}$. It comes from the fact that all the paths starting with two up displacements weren't considered by the (00)-expansion since converting only one zero after that expansion cannot allow that type of path. This is why this second step completes the ensemble of odd integer paths but also why it cannot originate from the previous integer paths, because one of the 1 values in the binary representation of those would need to be shifted to one of the two first entries, therefore changing the initial integer

value, to allow the construction of paths with an equal amount of up and right displacements. This step starts by creating the binary representation of the integer path of value 1 with $2n-2$ elements. To that array, the (11)-expansion is performed on the left then, but only if there are more zeros than ones, a conversion of zeros into ones is done to generate all missing odd integer paths. From the previous example, this looks like the following:

$$\underbrace{(1,1)}_{\text{expansion}} \; + \; \underbrace{(0,0,0,1)}_{\substack{\text{integer path of value 1} \\ \text{with } 2n-2 \text{ elements}}} \; \longmapsto \; (1,1,0,0,0,1) \doteq 49\,.$$

Since the number of zeros equals the number of ones, the path is immediately obtained without any conversion of 0 into 1. In higher lattice dimensions, it is necessary to use a recursive function in order to generate all the possible paths. Here is an explicit case of this second step for the missing numbers of $\mathcal{I}_5^{\text{odd}}$ after the (00)-expansion:

$$(1,1) \; + \; (0,0,0,0,0,0,0,1) \; \longmapsto \; (1,1,0,0,0,0,0,0,0,1)\,.$$

More than one 0 needs to be converted into a 1, therefore a recursive function is used where a portion of the binary array is fixed after a transformation :

$$(1,1,0,0,0,0,0,0,0,1) \overset{0\mapsto 1}{\longmapsto} (1,1,\underbrace{1}_{\text{fixed}},0,0,0,0,0,0,1) \longmapsto \left[\begin{array}{l} (1,1,1,1,0,0,0,0,0,1) \\ (1,1,1,0,1,0,0,0,0,1) \\ (1,1,1,0,0,1,0,0,0,1) \\ (1,1,1,0,0,0,1,0,0,1) \\ (1,1,1,0,0,0,0,1,0,1) \\ (1,1,1,0,0,0,0,0,1,1) \end{array}\right.$$

$$(1,1,0,0,0,0,0,0,0,1) \overset{0\mapsto 1}{\longmapsto} (1,1,\underbrace{0,1}_{\text{fixed}},0,0,0,0,0,1) \longmapsto \left[\begin{array}{l} (1,1,0,1,1,0,0,0,0,1) \\ (1,1,0,1,0,1,0,0,0,1) \\ (1,1,0,1,0,0,1,0,0,1) \\ (1,1,0,1,0,0,0,1,0,1) \\ (1,1,0,1,0,0,0,0,1,1) \end{array}\right.$$

and so on until reaching the last combination:

$$(1,1,0,0,0,0,0,0,0,1) \overset{0\mapsto 1}{\longmapsto} (1,1,\underbrace{0,0,0,0,0,1}_{\text{fixed}},0,1) \longmapsto \left[\begin{array}{l} (1,1,0,0,0,0,0,1,1,1) \end{array}\right.$$

This step produces all the $\binom{2n-3}{n-3} = \binom{2n-3}{n}$ numbers missing from step one in order to equate the set $\mathcal{I}_n^{\text{odd}}$ of all odd integer paths in a square lattice $\Lambda_{n,n}$ of at least size $n \geq 3$.

Remark: One could think that the second step needs to start at a square lattice of size $n = 2$ but this expansion treats the missing paths where two successive up movements can be done at the start while still coming from the bottom of the endpoint, which is impossible before $n = 3$.

A verification of the count of the number of elements produced by these two steps shows no more missing elements in comparison to the total number of odd integer paths $|\mathcal{I}_n^{\text{odd}}|$:

# elements	1	2	3	4	5	...	n
$\lvert\mathcal{I}_n^{\text{odd}}\rvert$	1	3	10	35	126	...	$\frac{1}{2}\binom{2n}{n} = \frac{1}{2}\lvert\Pi_{n,n}\rvert$
produced by (00)-exp	/	3	9	30	105	...	$3\binom{2n-3}{n-1}$
produced by (11)-exp	/	/	1	5	21	...	$\binom{2n-3}{n}$

where the number of paths produced by the two expansions coincidently relates to a visible property of Pascal's triangle described in Figure 10 below,

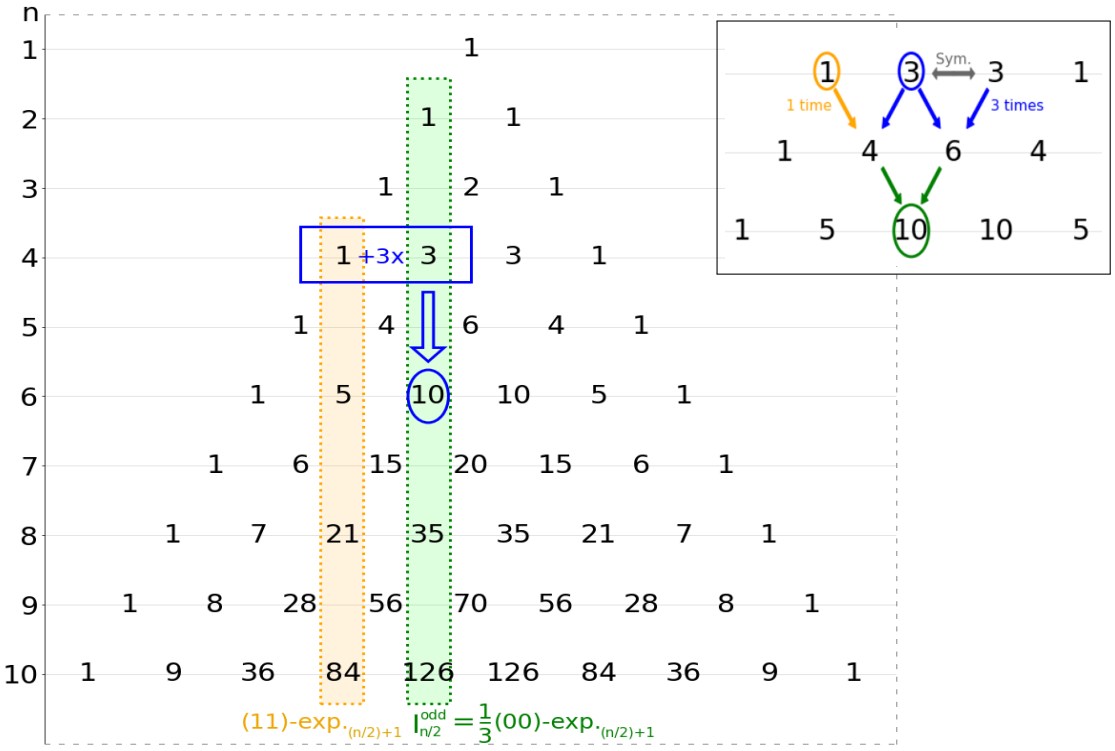


Figure 10: The two expansions of the algorithm relate to a property of the Pascal's triangle.

and is mathematically verified:

$$
\begin{aligned}
\binom{2n-3}{n} + 3\binom{2n-3}{n-1} &= \frac{(2n-3)!}{n!\,(n-3)!} + \frac{3\,(2n-3)!}{(n-1)!\,(n-2)!} \\
&= \frac{(2n)!}{(n!)^2}\left[\frac{n\,(n-1)(n-2)}{2n(2n-1)(2n-2)} + \frac{3n^2\,(n-1)}{2n(2n-1)(2n-2)}\right] \\
&= \binom{2n}{n}\left[\frac{n-2+3n}{4(2n-1)}\right] \\
&= \frac{1}{2}\binom{2n}{n} \\
&= \lvert\mathcal{I}_n^{\text{odd}}\rvert\,.
\end{aligned}
$$

It is therefore visible that the low-calculation cost (00)-expansion will contribute the most to generating the next odd integer paths ensemble in comparison to the more time-consuming (11)-expansion.

Given that procedure to generate all odd integer paths ensemble $\mathcal{I}_n^{\text{odd}}$, the code can be focused on the reading of these numbers to move through the lattice and multiply weights along the way. Therefore the computation time is linear, that is, proportional to the total number of odd integer paths $|\mathcal{I}_n^{\text{odd}}|$, which tends to grow exponentially by a factor of 4, as a corollary of the relation (24). Likewise, the data storage of these ensembles also grows by the same factor, see Figure 11, where the last file contains all the 77.558.760 odd integer paths for $\Lambda_{15,15}$.

odd_integer_ensemble_v2_N_2.txt	19-12-23 18:26	Document texte	1 Ko
odd_integer_ensemble_v2_N_3.txt	19-12-23 18:27	Document texte	1 Ko
odd_integer_ensemble_v2_N_4.txt	30-01-24 13:32	Document texte	1 Ko
odd_integer_ensemble_v2_N_5.txt	31-01-24 14:40	Document texte	1 Ko
odd_integer_ensemble_v2_N_6.txt	19-12-23 18:27	Document texte	3 Ko
odd_integer_ensemble_v2_N_7.txt	19-12-23 18:27	Document texte	11 Ko
odd_integer_ensemble_v2_N_8.txt	31-01-24 14:39	Document texte	44 Ko
odd_integer_ensemble_v2_N_9.txt	19-12-23 18:27	Document texte	181 Ko
odd_integer_ensemble_v2_N_10.txt	31-01-24 14:40	Document texte	719 Ko
odd_integer_ensemble_v2_N_11.txt	19-12-23 18:30	Document texte	3.022 Ko
odd_integer_ensemble_v2_N_12.txt	31-01-24 14:41	Document texte	12.356 Ko
odd_integer_ensemble_v2_N_13.txt	31-01-24 14:52	Document texte	50.193 Ko
odd_integer_ensemble_v2_N_14.txt	30-01-24 13:37	Document texte	207.968 Ko
odd_integer_ensemble_v2_N_15.txt	30-01-24 13:59	Document texte	832.452 Ko

Figure 11: The exponential growth of the data storage of odd integer paths ensembles.

The core structure of this algorithm design is now sufficiently detailed so that anyone can understand it. Its advantage is the possibility to run each iteration of the calculations in parallel thanks to a GPU port or thread management, and it also allows to keep all the information within the paths' weights, which could be useful in other numerical studies.
Unfortunately for this one, the use of this paths generator cannot fully contribute since the convergence of the first theorem wasn't as fast as it was first thought it would be at the start of this thesis, meaning that the storage of the odd integer paths would be tremendous before even thinking of porting the computation on the GPU. It was nevertheless used to confort values given by the second algorithm for the same lattice sizes.

Second algorithm: an iterative design

The central piece of the second algorithm takes root from the relation (21), which is rewritten below:

$$Z_{m+1,n+1} = (Z_{m,n+1} + Z_{m+1,n}) \cdot \omega_{m+1,n+1}$$

As already mentioned, even if this relation seems to offer the value of all the partition functions along the diagonal, only the last one can be kept as valid since the weights distributed on the square lattice $\Lambda_{n,n}$ are fixed during their calculations, meaning that they are not independent random variables anymore.

The relation (21) works well within the algorithm but can cause overflow errors since the weights $\{\omega_{i,j}\} \sim Ga^{-1}(\alpha)$ can be really large depending on the shape parameter α. To compensate for that, a change by taking the logarithm of the partition function is recommended:

$$\begin{aligned}
\log\left(Z_{m+1,n+1}\right) &= \log\left(\left(Z_{m,n+1} + Z_{m+1,n}\right) \cdot \omega_{m+1,n+1}\right) \\
&= \log\left(Z_{m,n+1} \cdot \left(1 + \frac{Z_{m+1,n}}{Z_{m,n+1}}\right) \cdot \omega_{m+1,n+1}\right) \\
&= \log\left(Z_{m,n+1}\right) + \log\left(1 + \frac{Z_{m+1,n}}{Z_{m,n+1}}\right) + \log\left(\omega_{m+1,n+1}\right) \\
&= \log\left(Z_{m,n+1}\right) + \log\left(1 + \frac{e^{\log(Z_{m+1,n})}}{e^{\log(Z_{m,n+1})}}\right) + \log\left(\omega_{m+1,n+1}\right) \\
&= \log\left(Z_{m,n+1}\right) + \log\left(1 + e^{\log(Z_{m+1,n}) - \log(Z_{m,n+1})}\right) + \log\left(\omega_{m+1,n+1}\right).
\end{aligned}$$

Let $m, n \in \mathbb{Z}_+$ and $Y_{m,n} \equiv \log\left(Z_{m,n}\right)$ be the random variable corresponding to the height function. The relation (21) can be written as:

$$Y_{m+1,n+1} = Y_{m,n+1} + \log\left(1 + e^{Y_{m+1,n} - Y_{m,n+1}}\right) + \log\left(\omega_{m+1,n+1}\right) \tag{41}$$

which no longer produces overflow errors during the computation.

This iterative version of the algorithm therefore allows one to calculate the free energy for a very large square lattice. Even with the loss of some details on the paths' weight, this was necessary to observe relevant numerical results.

3.2. Numerical results

This section covers the numerical results obtained by the second algorithm. It starts with the verification of the distribution of the weights from an inverse-gamma distribution $\mathrm{Ga}^{-1}(\alpha, 1)$ with shape parameter α and scale parameter set to one. After that the random environment is confirmed to fit the inverse-gamma distribution, the theorems are compared to the numerical evaluation of the functions (36-38) and observations are made about these results.

Verification of the inverse-gamma distribution

The first thing to do is to verify the modalities of the model, which is if the samples of the distributed weights correspond indeed to an inverse-gamma distribution.

With 1 iteration of random weights distributed inside a square lattice $\Lambda_{300,300}$ of size $n = 300$, the resulting histograms for $\alpha = 5.00,\ 2.00$ and 0.80 comply with the inverse-gamma distribution as the figures below indicate.

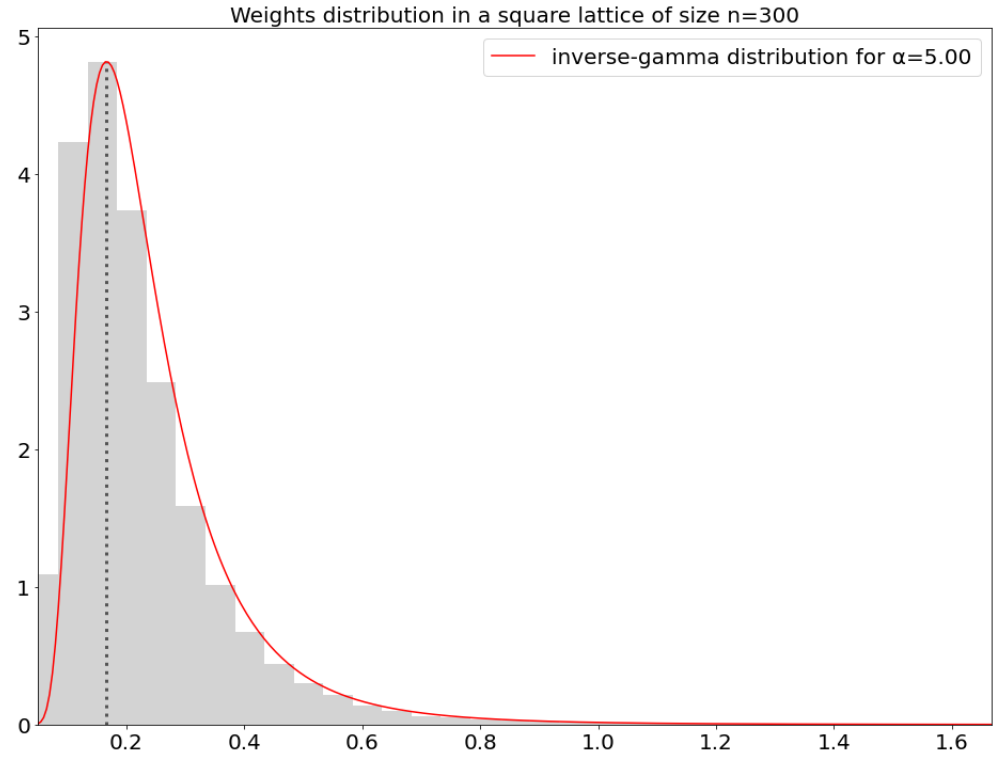


Figure 12: Histogram of the 90.000 distributed weights with the inverse-gamma distribution ($\alpha = 5.00$).

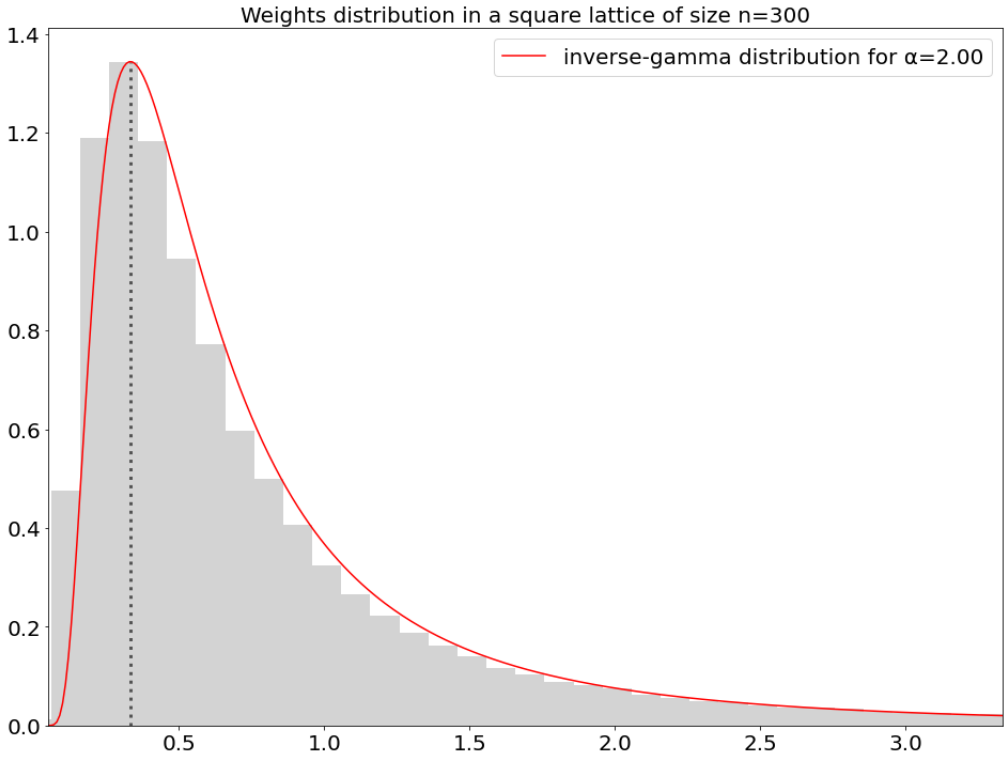


Figure 13: Histogram of the 90.000 distributed weights with the inverse-gamma distribution ($\alpha = 2.00$).

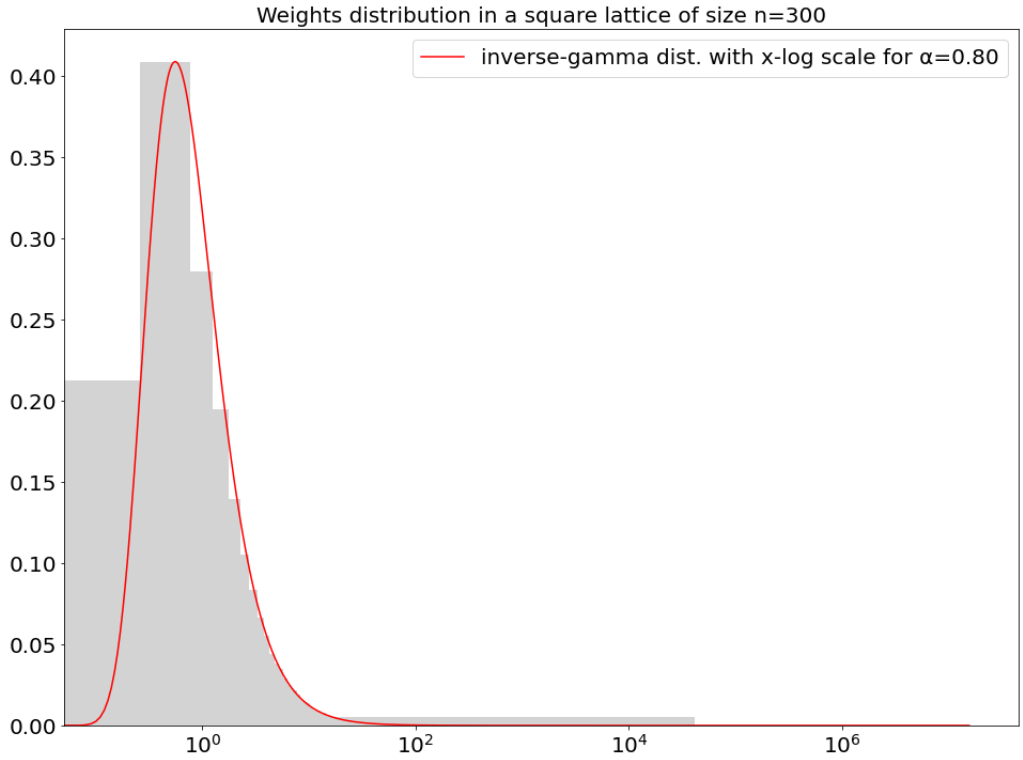


Figure 14: Histogram of the 90.000 distributed weights with the inverse-gamma distribution ($\alpha = 0.80$) in x-log scale.

Larger values of the shape parameter allow the rendering of a simple fit of the distributed weights with the inverse-gamma distribution but an x-log scale is required for low shape parameters such as $\alpha = 0.80$ because larger values are more probable due to the definition of the inverse-gamma distribution.

For the first and second theorems, 40 iterations per square lattice size n are performed meaning that these results ensure that for $n = 48$ and higher, the numerical outputs are with a correct distribution of weights. Similar results can be obtained with other shape parameters and lower square lattice size and with more iteration.

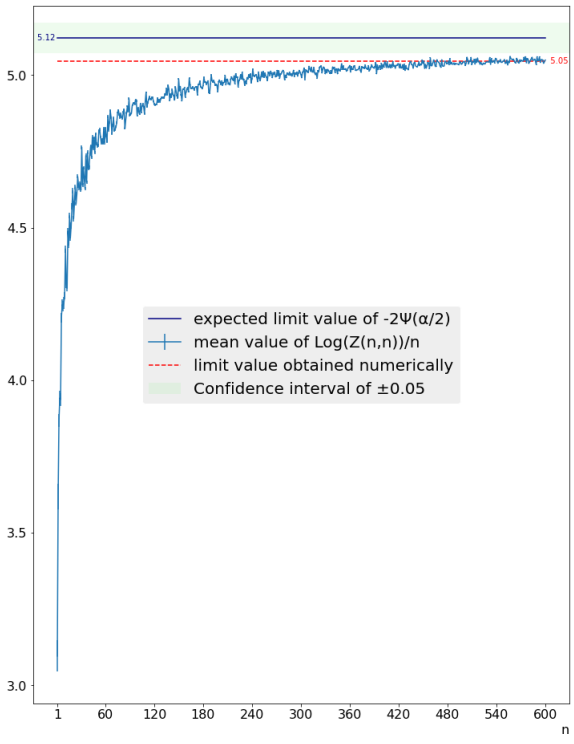
Theorem 1: Convergence of the scaled free energy

The theorem 1 is given by the equation (27) and studied by looking at the asymptotic behaviour of the function T_n^1 defined by (36), rewritten below:

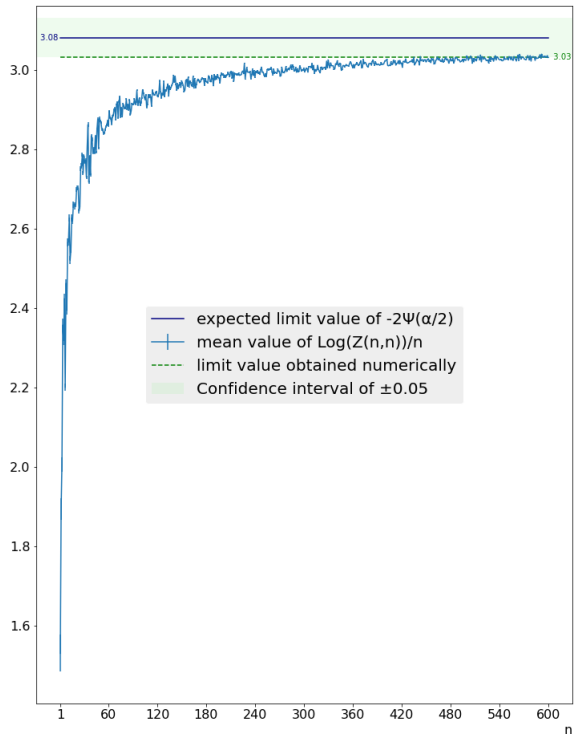
$$T_n^1(\alpha) \;=\; \frac{\log(Z_{n,n}(\alpha))}{n}.$$

It implies that the scaled free energy should converge almost surely in probability to a limit value. Since T_n^1 represents the rescaled height function h from the KPZ universality class, the convergence is done towards its limit shape. The height function h satisfies the KPZ equation (4), which is a heat equation. It means that the convergence rate is expected to slow down as the free energy approaches this limit due to the difference in the order of the derivatives : $\partial_t h(t,x) \propto \partial_x^2 h(t,x)$, which is here described in terms of t and x before any adaptation of the model such as the transformation (3). This is true for all models where the height function is linked to the KPZ universality class. For that reason, a confidence interval is subjectively defined around the expected limit value within which the convergence seems close enough. In this study, the confidence interval is set as the range of ± 0.05 of the expected limit value.

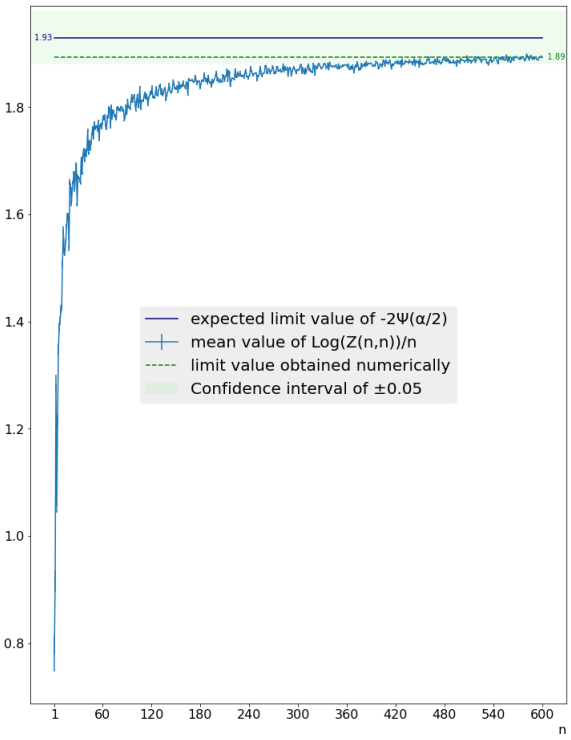
The following figures show the value of the scaled free energy T_n^1 against the lattice size n for shape parameter values $\alpha := 0.80$, 1.20, 1.60, 2.00, 3.00, and 5.00 from left to right and top to bottom respectively, with 40 iterations per value of n.



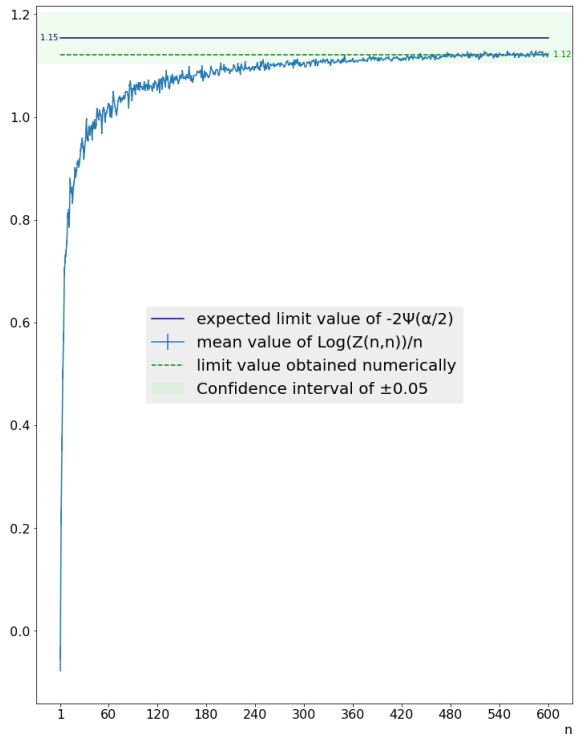
Convergence of T_n^1 for $\alpha = 0.80$ from $n = 1$ to $n = 600$.



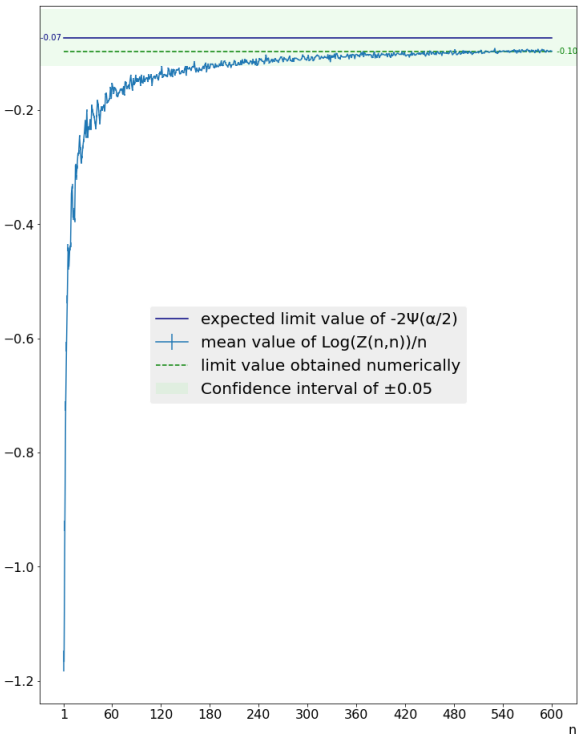
Convergence of T_n^1 for $\alpha = 1.20$ from $n = 1$ to $n = 600$.



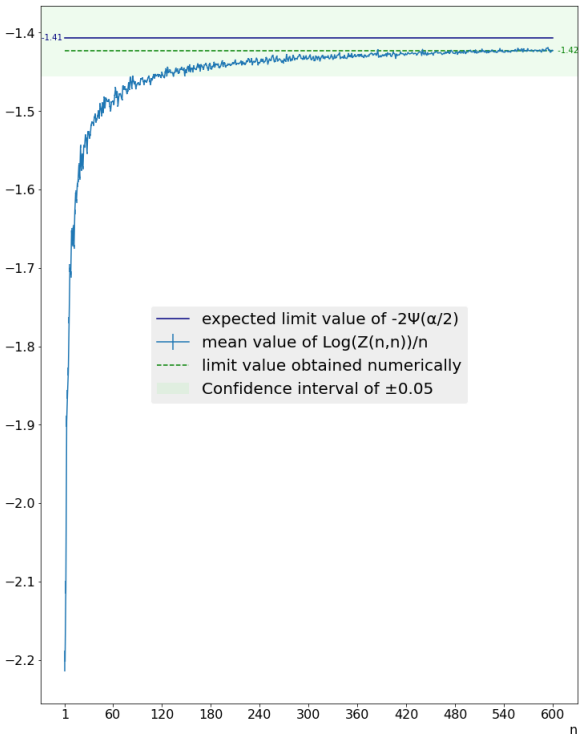
Convergence of T_n^1 for $\alpha = 1.60$ from $n = 1$ to $n = 600$.



Convergence of T_n^1 for $\alpha = 2.00$ from $n = 1$ to $n = 600$.



Convergence of T_n^1 for $\alpha = 3.00$ from $n = 1$ to $n = 600$.



Convergence of T_n^1 for $\alpha = 5.00$ from $n = 1$ to $n = 600$.

It is easy to see that indeed the function T_n^1 converges at large values of the square lattice size n but it is not clear nonetheless if the limit value obtained converges really to the one expected by

theorem 1. By going further in the lattice size expansion, the function approaches a bit more to the expected value but at a slow rate. Two possible explanations for these results are that:

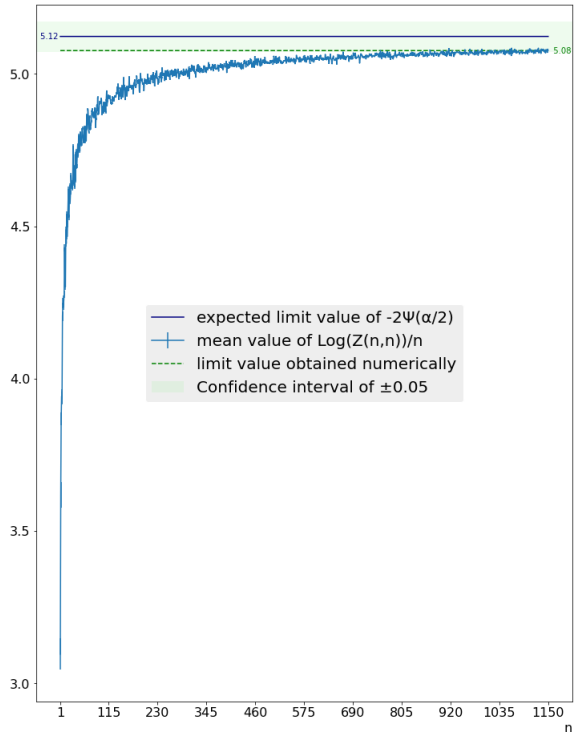
1. There could have rounding errors due to the limitation of the significand (also called the mantissa), which is the first (left) part of a number in the scientific notation related to the floating-point representation. In other words, when two numbers of different magnitudes are added or subtracted, some digits in the result are truncated because of the finite size of the significand. These rounding errors could happen in the relation (41) rewritten below:

$$Y_{m+1,n+1} \;=\; Y_{m,n+1} \;+\; \log\left(1 + e^{Y_{m+1,n} - Y_{m,n+1}}\right) \;+\; \log\left(\omega_{m+1,n+1}\right)$$

   where one or even two of the terms of the r.h.s. could contribute less than expected because of a significant difference in the magnitude, meaning that the contributions to the value of T_n^1 are not correctly done even with an expanding lattice. A quick verification of the magnitude during the calculation shows that at most 3 digits are truncated during the summation of the three terms, in favour of the first term, which has the highest value of them.
   Another source of rounding errors could happen during the subtraction in the exponential of the second term. This is called "catastrophic cancellation" and occurs with terms of the same order or magnitude, which could easily be the case here between $Y_{m+1,n}$ and $Y_{m,n+1}$. This possible rounding error hasn't been checked.

2. A second explanation of this result could be due to a finite-scale effect. It relates to the chosen stopping point in the square lattice size that may be too "short" to represent the infinite size given analytically. Therefore, a proper convergence would necessitate higher scales of the thermodynamical variable to reach its expected limit value.

This last point seems to be confirmed by looking at how it is required to go almost twice as large as before (from $n = 600$ to $n = 1150$) with the square lattice size to simply reach the confidence interval for $\alpha = 0.80$, see the figure on the right.



Convergence of T_n^1 for $\alpha = 0.80$
from $n = 1$ to $n = 1150$.

Observation 1. The convergence rate is slower for lower shape parameters. This could seem counterintuitive at first, knowing that a lower shape parameter means a higher magnitude in the values given by the probability density function of the inverse-gamma distribution, see Figure 4, but it also means a higher variance of the free energy. This higher variance slows the convergence. See the following Figure 15 for a comparison:

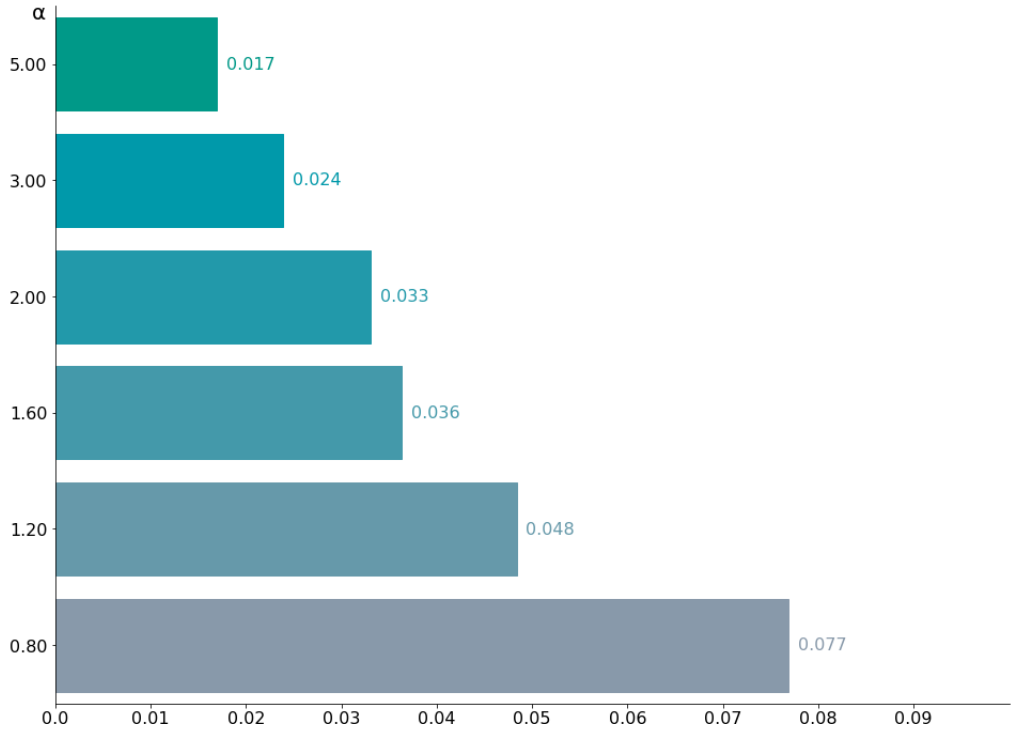


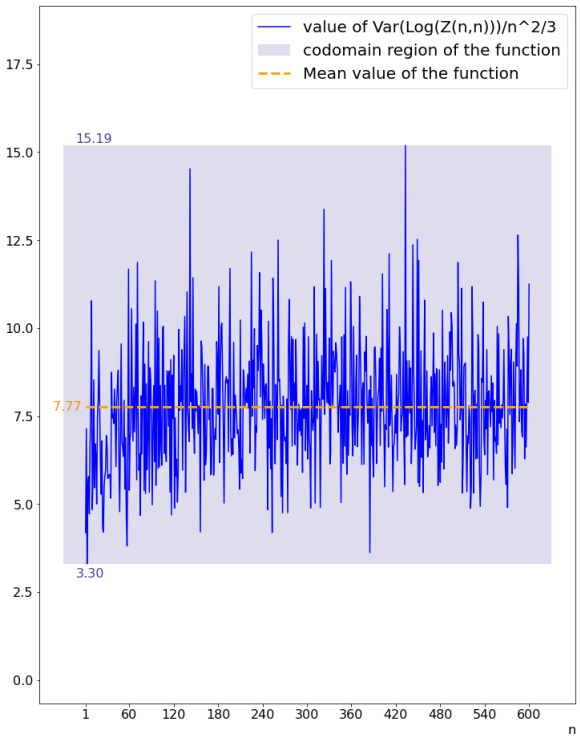
Figure 15: Difference between the expected limit value and the numerical limit value obtained for different shape parameters and $n = 600$.

Theorem 2: Scaled variance upper-bound of the free energy

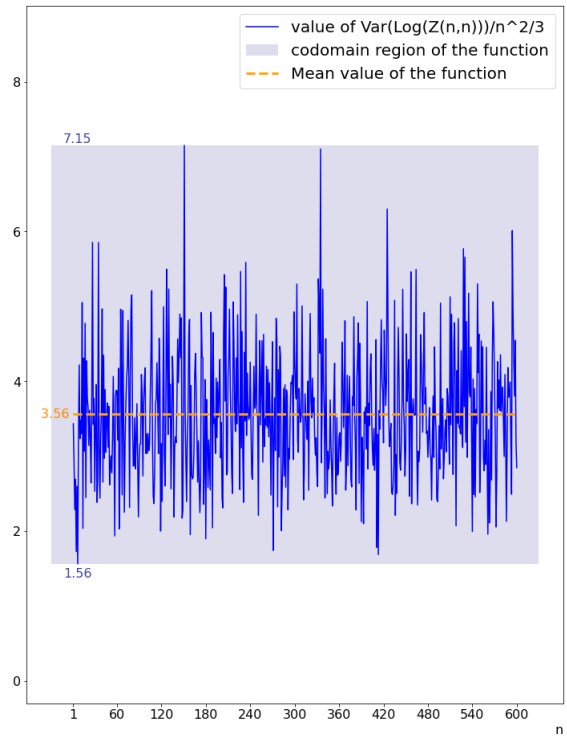
The second theorem (28) is studied through the function given by (37):

$$T_n^2(\alpha) \;=\; \frac{\mathbb{V}\mathrm{ar}\left[\log(Z_{n,n}(\alpha))\right]}{n^{2/3}}\,.$$

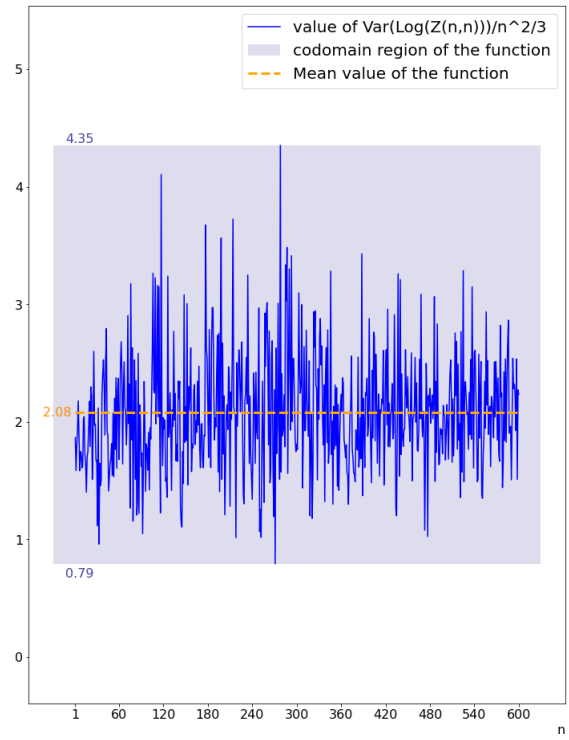
This theorem states that the codomain of T_n^2 is bounded. This is indeed what is observed in the following figures for which the values of the shape parameter and the square lattice size are the same as before.



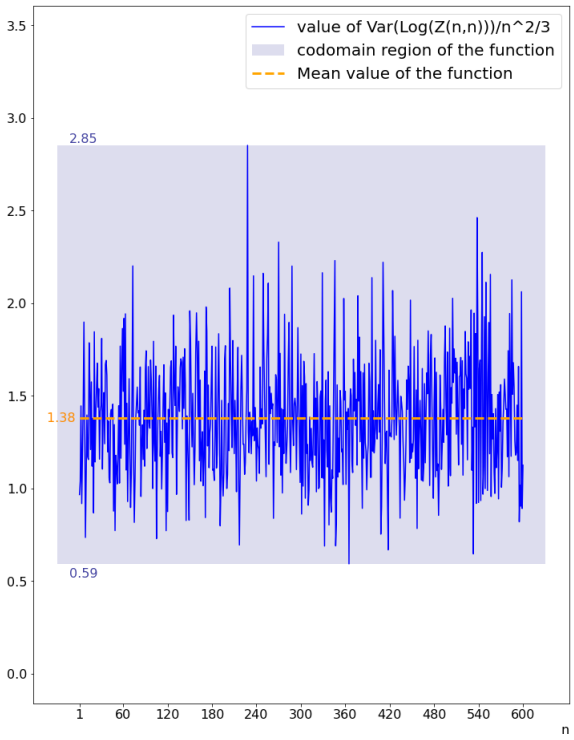
Codomain region of T_n^2 for $\alpha = 0.80$ from $n = 1$ to $n = 600$.



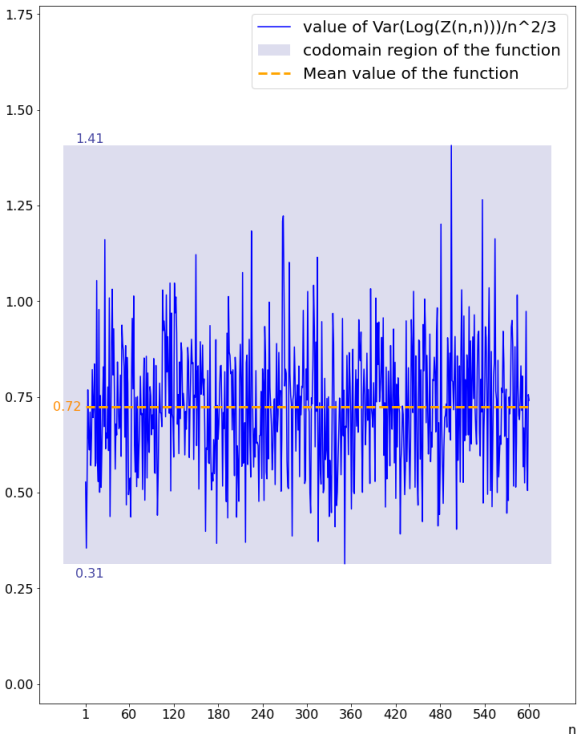
Codomain region of T_n^2 for $\alpha = 1.20$ from $n = 1$ to $n = 600$.



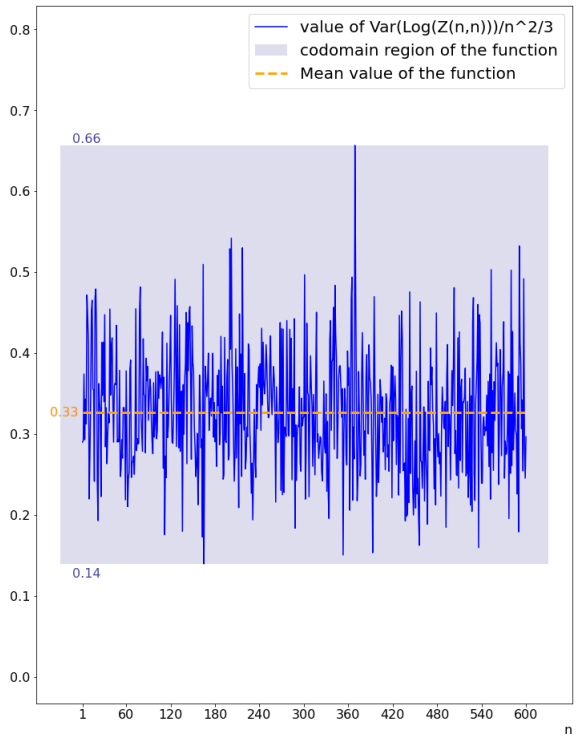
Codomain region of T_n^2 for $\alpha = 1.60$ from $n = 1$ to $n = 600$.



Codomain region of T_n^2 for $\alpha = 2.00$ from $n = 1$ to $n = 600$.



Codomain region of T_n^2 for $\alpha = 3.00$ from $n = 1$ to $n = 600$.



Codomain region of T_n^2 for $\alpha = 5.00$ from $n = 1$ to $n = 600$.

The numerical results of the second theorem are conclusive.

Observation 2. The range of the codomain is larger for small shape parameter values. This observation goes along with the supposition that the convergence rate is negatively affected by the higher variance of the inverse-gamma distribution for lower shape parameter values.

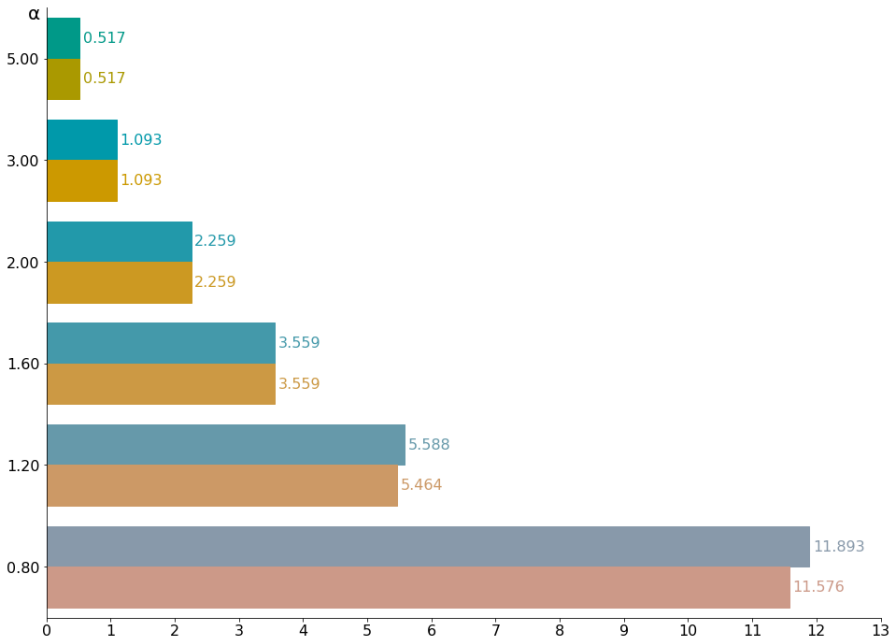


Figure 16: Codomain ranges of T_n^2 from $n = 0$ to $n = 600$ (in shades of blue) and from $n = 51$ to $n = 600$ (in shades of orange) for different shape parameters.

In Figure 16 above, the removal of the first 50 values of the square lattice sizes n is done to compare the impact of the starting weight on the variance for low shape parameters.
A recapitulative table of the information about the codomain values of T_n^2 for the different shape parameters for square lattice size from $n = 51$ to $n = 600$ is given below:

Shape parameter $(/\alpha)$	5.00	3.00	2.00	1.60	1.20	0.80
Upper bound value	0.656	1.407	2.851	4.351	7.150	15.193
Lower bound value	0.139	0.314	0.592	0.792	1.686	3.617
Range between lower and upper bounds	0.517	1.093	2.259	3.558	5.463	11.576
Mean value	0.324	0.725	1.385	2.100	3.576	7.877

Theorem 3: Fluctuations of the free energy as a Tracy-Widom distribution

The last theorem is the most important one. It makes the connection with the Tracy-Widom distribution and therefore the KPZ-universality class. The results of the relation (33) are studied through the resulting distribution of

$$T_n^3(\alpha) \;=\; \frac{\log(Z_{n,n}(\alpha)) + 2n\,\psi_0\left(\frac{\alpha}{2}\right)}{\left(-\psi_2\left(\frac{\alpha}{2}\right)\,n\right)^{\frac{1}{3}}}$$

which is given numerically below for the same shape parameters as before, and with two different numbers of bins in the histogram for each of them:

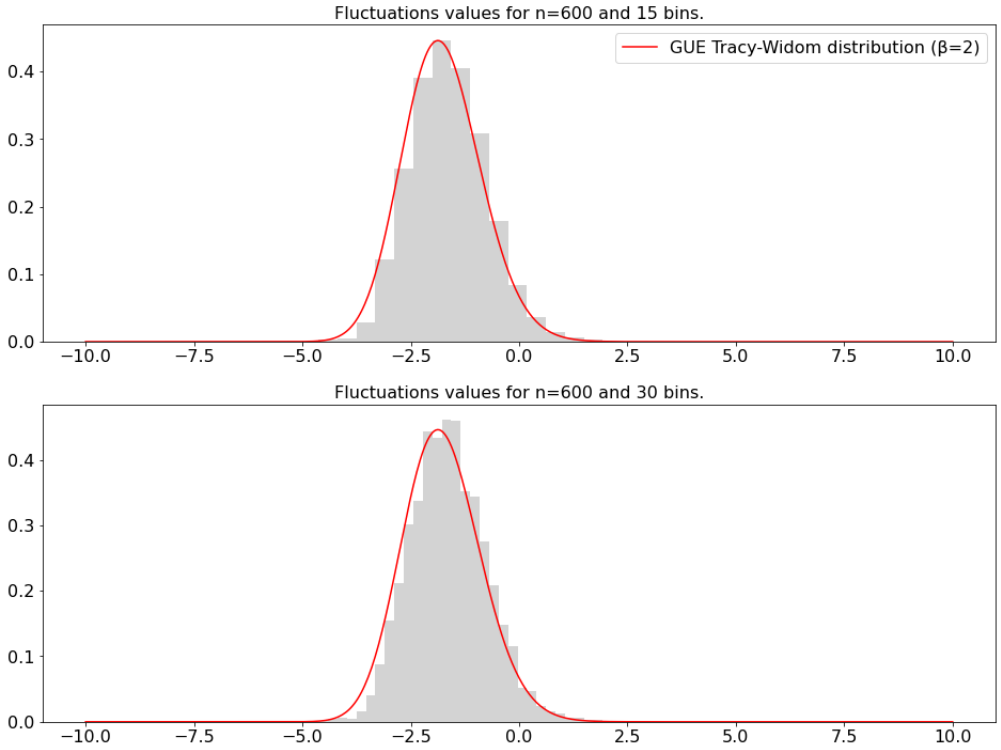


Figure 17: Distribution of T_{600}^3 for $\alpha = 0.80$ with 3000 iterations.

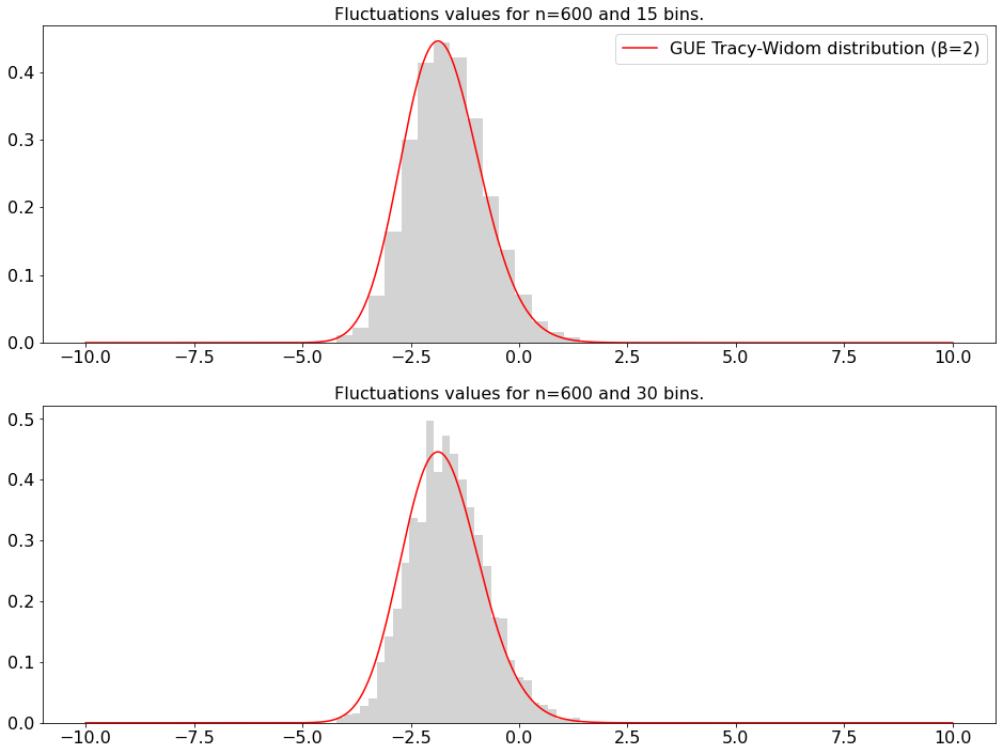


Figure 18: Distribution of T^3_{600} for $\alpha = 1.20$ with 3000 iterations.

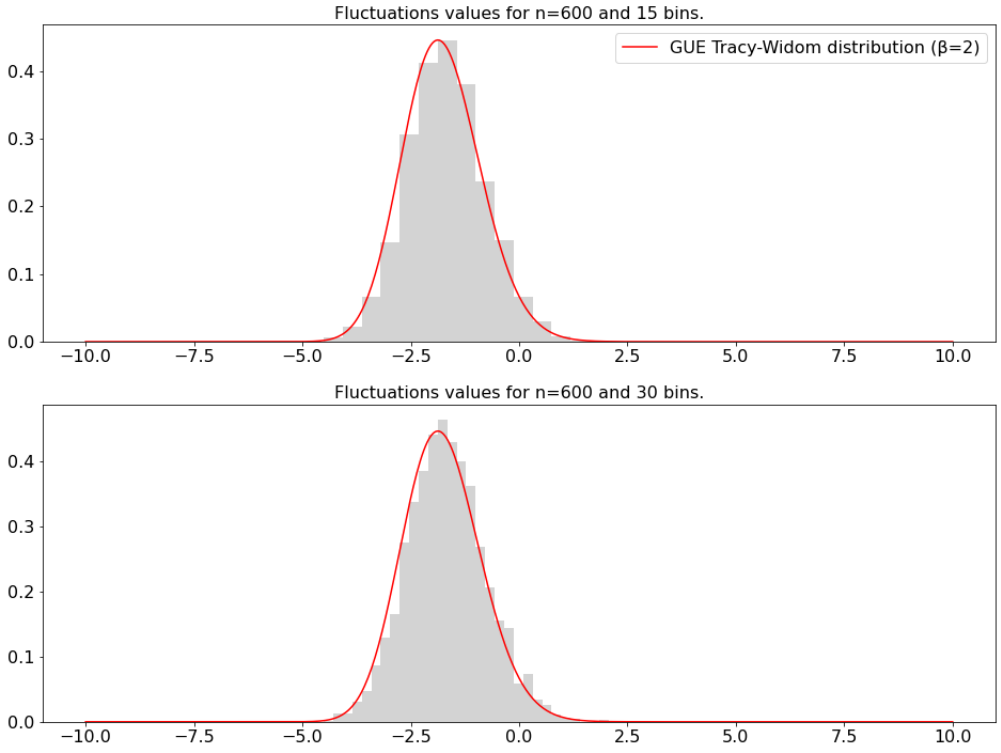


Figure 19: Distribution of T^3_{600} for $\alpha = 1.60$ with 3000 iterations.

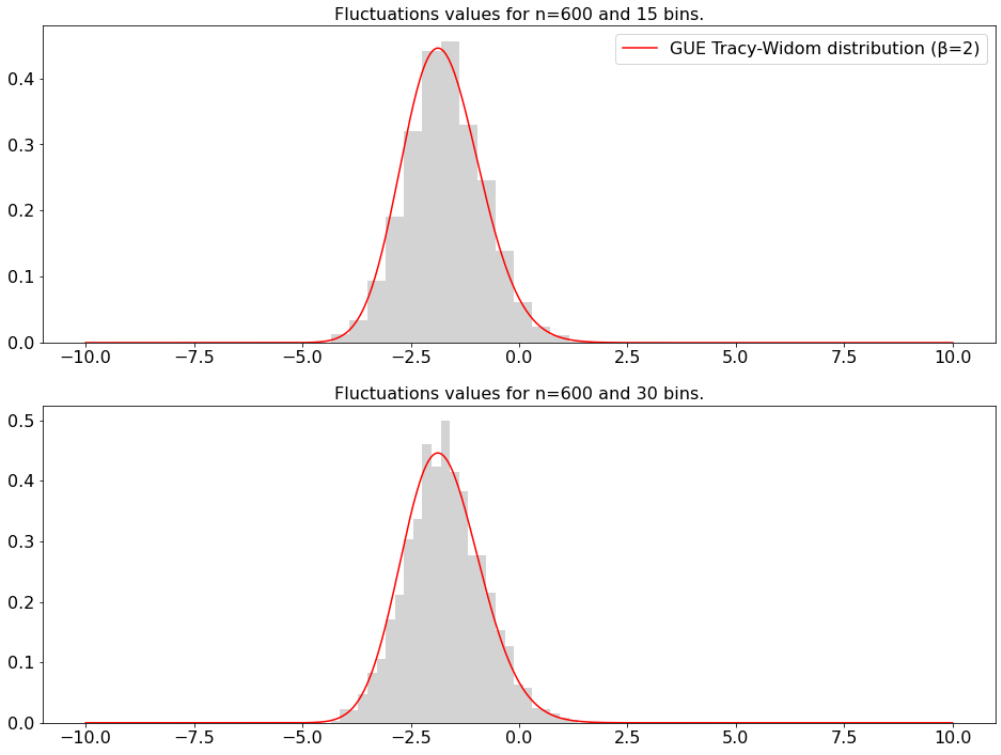


Figure 20: Distribution of T^3_{600} for $\alpha = 2.00$ with 3000 iterations.

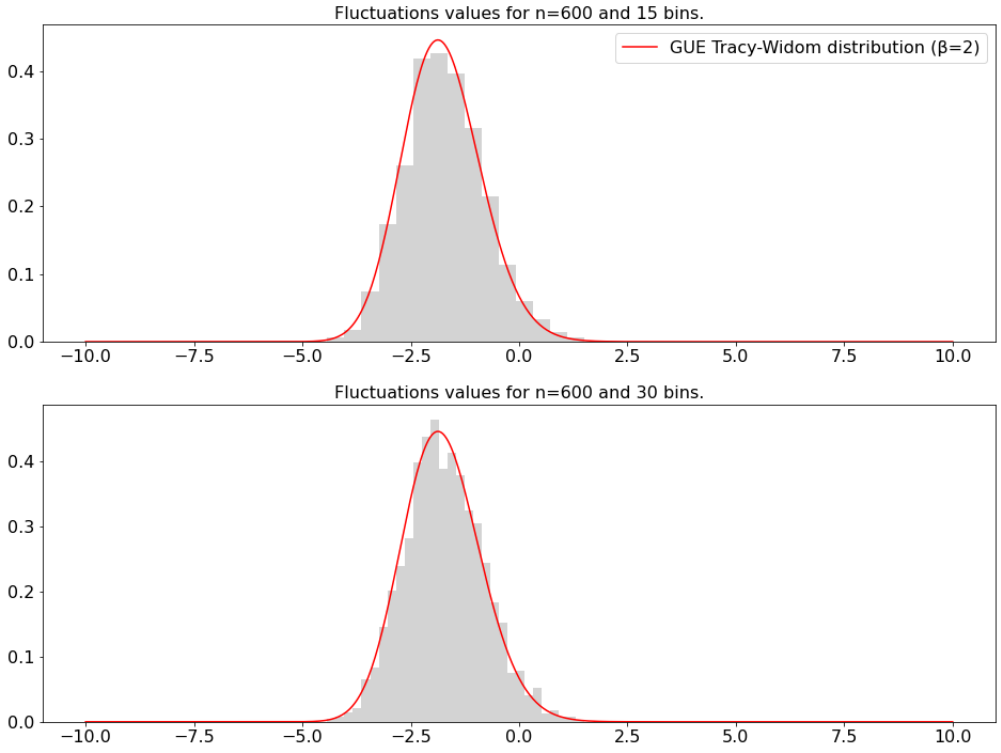


Figure 21: Distribution of T^3_{600} for $\alpha = 3.00$ with 3000 iterations.

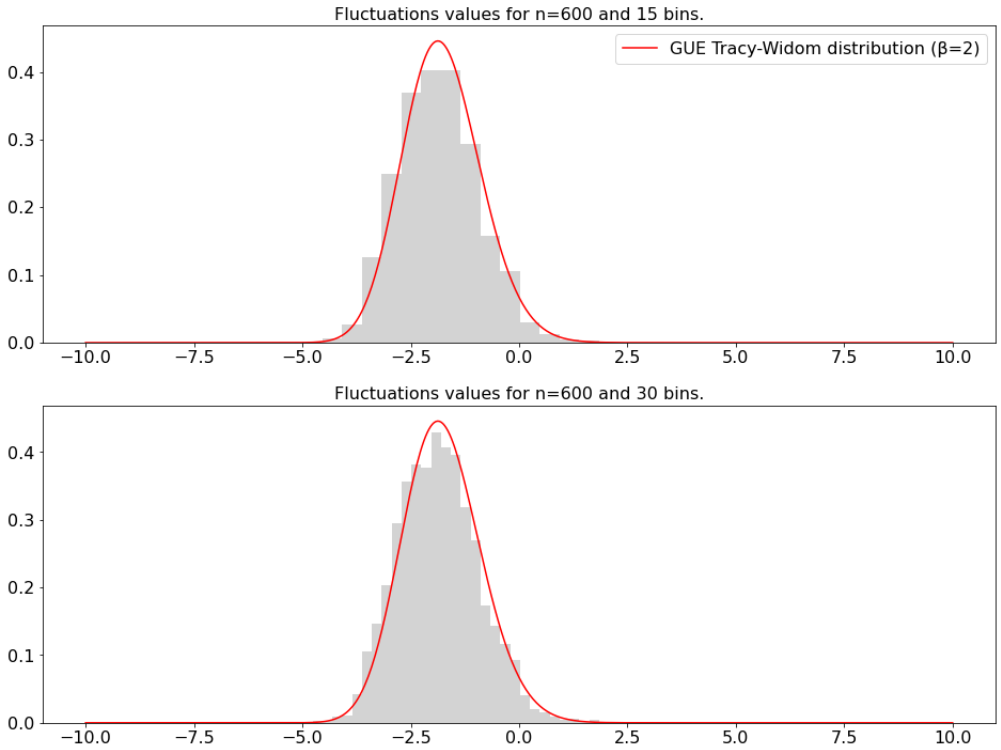


Figure 22: Distribution of T_{600}^3 for $\alpha = 5.00$ with 3000 iterations.

The distributions of the fluctuations value T_n^3 comply really well with the Tracy-Widom distribution for the 6 different values of α: 0.8, 1.2, 1.6, 2, 3 and 5, for a square lattice size of $n = 600$. The numerical results of the third theorem are therefore conclusive and agree with the analytical result.

Observation 3. One can see, and mostly on lower shape parameter values, that the bars of the histogram don't cross the curve in their middle. This means that the histogram is still not centred as the Tracy-Widom distribution. This shift in the resulting distribution goes along with the results observed with the first theorem since the expected limit value, which wasn't reached, is the value responsible for the recentering in Theorem 3.

In summary

The mathematical and the numerical models give results that are compatible with each other but with no precise match for the first theorem, which is still convincing even if "finite n effects" or numerical errors are clearly visible. The second and third theorems given by [1,2] on the other hand are verified flawlessly by the numerical results. Therefore the numerical study confirms the analytical results.

Conclusion

The Tracy-Widom distribution is an analog of the central limit theorem and the three theorems each plays a role in bringing the free energy of the inverse-gamma 1+1 directed polymer model to be distributed like it. Since the thermodynamical limit of the free energy exists (law of large number given by Theorem 1) and its variation is bounded (Theorem 2), both rescaled with a typical KPZ-scaling which leads to the limit distribution of the fluctuations as a Tracy-Widom distribution (Theorem 3).

The up-right random walk is a simpler model yet rich in both applications and observations. The use of an inverse-gamma distribution as the quenched disorder (impurities) for the random environment allows for a solid analytical foundation against which the comparison of numerical outputs is made easier. Nonetheless, the most difficult aspect of working with this model, in the vision of a numerical study, is the slow convergence rate of the rescaled free energy. The fact that it is required to go higher than a square lattice size of one hundred (9×10^{58} possible paths) to approach the expected limit value makes the numerical implementation both limited and challenging.

The numerical results are consistent with the analytical ones and the third theorem, which is the most important, is confirmed. To complete this study, a proper analysis of the possible rounding errors should be integrated and also the investigation of the convergence rate.

To go further with that analysis, it would be interesting to strengthen the mathematical aspects around the height function within this specific model, which is lacking in this thesis, and verify the KPZ equation to reinforce the whole structure of the model. The use of an iterative algorithm forces one to lose a lot of information that would have probably led to interesting observations. It's a compromise that is maybe possible to get around by a combination of both the iterative and parallel designs where the parallel approach could be partial and seen as a refinement of information given by the iterative one. Furthermore, the present algorithm is easily adaptable to explore different possibilities such as: making a connection with the corner growth model or the last passage time from percolation models; adding different boundary conditions such as the stationary ones that were used in [1] or other sources of inhomogeneity; trying different distributions to extract the weights and see if similar results could be obtained with the necessary analytical adaptations.

Bibliography

[1] Timo Seppäläinen. "Scaling for a one-dimensional directed polymer with boundary conditions," The Annals of Probability, 40(1) 19-73 January 2012. https://doi.org/10.1214/10-AOP617

[2] A. Borodin, I. Corwin and D. Remenik. "Log-Gamma Polymer Free Energy Fluctuations via a Fredholm Determinant Identity," Commun. Math. Phys. 324, 215–232 (2013). https://doi.org/10.1007/s00220-013-1750-x

[3] Xiao Shen. "Independence property of the Busemann function in exactly solvable KPZ models," arXiv:2308.11347 [math.PR]. https://doi.org/10.48550/arXiv.2308.11347

[4] Ivan Corwin. "The Kardar-Parisi-Zhang equation and universality class," arXiv:1106.1596 [math.PR]. https://doi.org/10.48550/arXiv.1106.1596

[5] Jinho Baik. "KPZ limit theorems," arXiv:2206.14086 [math.PR]. https://doi.org/10.48550/arXiv.2206.14086

[6] L. C. G. Rogers and J. W. Pitman. "Markov Functions," Ann. Probab. 9:573-582 (1981). https://doi.org/10.1214/aop/1176994363

[7] I. Corwin, N. O'Connell, T. Seppäläinen and N. Zygouras. "Tropical Combinatorics and Whittaker functions," arXiv:1110.3489 [math.PR]. https://doi.org/10.48550/arXiv.1110.3489

[8] E. Bates, W-T.L. Fan and T. Seppäläinen. "Intertwining the Busemann process of the directed polymer model," arXiv:2307.10531 [math.PR]. https://doi.org/10.48550/arXiv.2307.10531

[9] D.A. Huse, C.L. Henley. "Pinning and roughening of domain wall in Ising systems due to random impurities," Phys. Rev. Lett. 54 (1985) 2708-2711. https://doi.org/10.1103/PhysRevLett.54.2708

[10] J.Z. Imbrie, T.Spencer. "Diffusion of directed polymer in a random environment," J. Stat. Phys. 52 3/4 (1988) 608-626. https://doi.org/10.1007/BF01019720

[11] J. Hammersley and D. Welsh. "First-passage percolation, subadditive processes, stochastic networks, and generalized renewal theory." In Bernoulli 1713 Bayes 1763 Laplace 1813, pages 61–110. Springer, 1965.

[12] L. Talon, A. A. Hennig, A. Hansen and A. Rosso. "Influence of the imposed flow rate boundary condition on the flow of Bingham fluid in porous media." arXiv:2312.16639 [physics.flu-dyn]. https://doi.org/10.48550/arXiv.2312.16639

[13] B. Derrida and H. Spohn. "Polymers on disordered trees, spin glasses, and traveling waves," Journal of Statistical Physics. 51. 817-840. (1988). https://doi.org/10.1007/BF01014886

[14] Francis Comets. "Directed Polymers in Random Environments, École d'Été de Probabilités de Saint-Flour XLVI – 2016" Lecture Notes in Mathematics, Springer Cham. https://doi.org/10.1007/978-3-319-50487-2

[15] T. Thiery and P. Le Doussal. "Log-gamma directed polymer with fixed endpoints via the replica Bethe Ansatz," Journal of Statistical Mechanics: Theory and Experiment P10018 (2014). http://doi.org/10.1088/1742-5468/2014/10/P10018

[16] Z. Su, Y. Lei and T. Shen. "Tracy-Widom distribution, Airy_2 process and its sample path properties," Applied Mathematics-A Journal of Chinese Universities. 36. 128-158 (2021). http://doi.org/10.1007/s11766-021-4251-2

[17] T. Alberts and E. Cator. "On the Passage Time Geometry of the Last Passage Percolation Problem," ALEA, Lat. Am. J. Probab. Math. Stat. 18, 211–247 (2021). https://doi.org/10.30757/ALEA.v18-10

[18] Alessandra Occelli. "KPZ universality for last passage percolation models," PhD Thesis in Rheinische Friedrich-Wilhelms-Universität Bonn (2019). https://hdl.handle.net/20.500.11811/8071

[19] J.S. Pallister, S.H. Pickering, D.M. Gangardt, A.G. Abanov. "Phase transitions in full counting statistics of free fermions and directed polymers," arXiv:2405.12651 [cond-mat.stat-mech]. https://doi.org/10.48550/arXiv.2405.12651

[20] K. Matetski, J. Quastel, and D. Remenik "The KPZ Fixed Point," arXiv:1701.00018 [math.PR]. https://doi.org/10.48550/arXiv.1701.00018

[21] Erwin Bolthausen. "A Note on the Diffusion of Directed Polymers in a Random Environment," Commun. Math. Phys. 123, 529-534 (1989). https://doi.org/10.1007/BF01218584

[22] M.S.T. Piza. "Directed Polymers in a Random Environment: Some Results on Fluctuations," Journal of Statistical Physics, Vol. 89, Nos. 3/4 (1997). https://doi.org/10.1007/BF02765537

[23] Marco Chiani. "Distribution of the largest eigenvalue for real Wishart and Gaussian random matrices and a simple approximation for the Tracy–Widom distribution," Journal of Multivariate Analysis 129 69–81 (2014). http://dx.doi.org/10.1016/j.jmva.2014.04.002

[24] P.L. Ferrari and H. Spohn. "Random Growth Models," arXiv:1003.0881 [math.PR]. https://doi.org/10.48550/arXiv.1003.0881

[25] S. Das and W. Zhu. "Short and Long-Time Path Tightness of the Continuum Directed Random Polymer," arXiv:2205.05670 [math.PR]. https://doi.org/10.48550/arXiv.2205.05670

[26] K. Khanin and L. Li. "On end-point distribution for directed polymers and related problems for randomly forced Burgers equation," Phil. Trans. R. Soc. A 380:20210081. https://doi.org/10.1098/rsta.2021.0081

[27] F. Comets and N. Yoshhida. "Directed Polymers in Random Environment are Diffusive at Weak Disorder," arXiv:math/0411223 [math.PR]. https://doi.org/10.48550/arXiv.math/0411223

[28] N. Zygouras. "Directed polymers in a random environment: a review of the phase transitions," arXiv:2401.01757 [math.PR]. https://doi.org/10.48550/arXiv.2401.01757

Code implementation

The code implementation in terms of Python 3 classes is presented below.

InverseGamma class

This class is used to generate the weights according to an inverse-gamma distribution of the shape parameter given in the parameter and a fixed scale parameter of one.

```
class InverseGamma():
    """
    InverseGamma class associated to the distribution of the same name with
    fixed scale parameter beta, which is set to one.

    Parameters
    ----------
    shape_parameter : FLOAT, optional
        The shape parameter of the inverse-gamma distribution.
        The default is 2.0.

    Example
    -------
    dist = InverseGamma()
    title = "Example of an inverse-gamma distribution"
    dist.plot_histogram(dist.get_random_weights(90000), False, title)
    dist.set_shape(0.8)
    title += " with xlog-scale"
    dist.plot_histogram(dist.get_random_weights(90000), True, title)
    print(np.array(dist.generate_random_weights((5,5))))

    """

    def __init__(self, shape_parameter=2.0):

        if shape_parameter <= 0:
            shape_parameter = 2.0
            print("The shape parameter value should be strictly positive." +\
                  "The default value (2.0) is set.")
        self.__shape = shape_parameter

    def get_shape(self):
        """
        Returns the shape parameter of the inverse-gamma distribution (FLOAT).
        """
        return self.__shape

    def set_shape(self, shape_parameter):
        """
        Set a new value for the shape parameter of the inverse-gamma dist.

        Parameters
        ----------
        shape_parameter : FLOAT
            New value of the shape parameter.

        """
```

```
        if shape_parameter > 0:
            self.__shape = shape_parameter
        else:
            Warning("The shape parameter value should be strictly positive.")

    def plot_histogram(self, weights, is_xlog_scale=False, suptitle="", line_steps=100000, fontsize=20, **kwargs):
        """
        Generate an adapted histogram where depending of the shape parameter,
        an x-log scale can be used while still visualizing bins correctly.

        Parameters
        ----------
        weights : ARRAY(FLOAT)
            An array of shape (1,_) with the distributed weights to plot.
        is_xlog_scale : BOOLEAN, optional
            The default is False.
        suptitle : STRING, optional
            Title of the plot. The default is "".
        line_steps : INT, optional
            Number of steps for the drawing of the PDF. The default is 100000.
        fontsize : INT, optional
            The default is 20.
        **kwargs : DICTIONARY
            Keyword arguments to pass into the function matplotlib.pyplot.bar.

        """
        fig = plt.figure(figsize=(16,12))
        ax = fig.add_subplot(1, 1, 1)

        ### Drawing the bars ###
        mode = 1/(1+self.__shape)
        upper_lim = 40 * mode
        x1 = np.linspace(0.01, upper_lim, line_steps, endpoint=False)
        y1 = [gengamma.pdf(x1[i], self.__shape, -1) for i in range(len(x1))]
        bins = np.linspace(0.01, upper_lim, 200, endpoint=False)
        width = []
        for i in range(len(bins) - 1):
            width.append(bins[i+1]-bins[i])
        width.append(0)

        count = [0]*len(bins)
        for i in range(len(width)):
            for j in range(len(weights)):
                if weights[j] >= bins[i] and weights[j] < (bins[i] + width[i]):
                    count[i] +=1
        density_ratio = gengamma.pdf(mode, self.__shape, -1)/max(count)
        height = density_ratio*np.array(count)
        ax.bar(bins, height=height, width=width, **kwargs)

        ### Drawing the PDF ###
        if is_xlog_scale:
            add_on = np.linspace(upper_lim, 3*upper_lim, line_steps, endpoint=False)
            x2 = np.concatenate((x1,add_on))
            y2 = y1 + [gengamma.pdf(add_on[i], self.__shape, -1) for i in range(len(add_on))]
            ax.set(xscale="log")
            ax_line, = ax.plot(x2, y2, color="red")
```

```
            ax.set_xlim(0.05, ax.get_xlim()[1])

        else:
            ax.vlines(1/(1+self.__shape), 0, max(y1), color='k', ls=":", lw=3,
    alpha=0.6)
            ax_line, = ax.plot(x1, y1, color="red")
            ax.set_xlim(0.05, upper_lim/3)

        ax_legend_order_set = [ax_line]
        ax_legend_label_set = ["inverse-gamma " + ("distribution" if not
    is_xlog_scale else "dist. with x-log scale") + " for   =%.2f"%(self.__shape)]
        ax.tick_params(axis='both', labelsize=fontsize)
        ax.legend(ax_legend_order_set, ax_legend_label_set, loc='upper right',
    fontsize=fontsize)
        ax.set_title(suptitle, fontsize=fontsize)
        plt.show()

    def get_random_weights(self, number_of_variates=1):
        """
        Returns random weights from this inverse-gamma distribution with scale
        parameter set to 1 (ARRAY(FLOAT) or FLOAT).

        Parameters
        ----------
        number_of_variates : INT, optional
            The number of weights to return. The default is 1.

        """
        return gengamma.rvs(self.__shape, -1, size=number_of_variates)

    def generate_random_weights(self, array_shape):
        """
        Returns an array of random weights from the inverse-gamma distribution
        in the given array shape (ARRAY(FLOAT)).

        Parameters
        ----------
        array_shape : TUPLE
            Size (m,n) of the array.

        """
        number_of_weights = array_shape[0]*array_shape[1]
        weights = np.reshape(self.get_random_weights(number_of_weights),
    array_shape)
        return weights
```

Path class

This class is used within the first algorithm to generate and read the odd integer path ensembles given the square lattice size.

```
class Path():
    """
    Class acting as a reference of a diagonal displacement in a 2D up-right
    directed lattice.
    It makes the link between integers and arrow representation (-:0, |:1)
    which is a binary 2N-uplet of the coefficients of powers of 2 until the
    (2N-1)'s power, where N is the number of diagonal steps in the lattice.
```

```
8
9       Parameter:
10          diagonal_steps_dimension (int): this number is equal to both the number
11          of right displacement and up displacement on the lattice. Making this
12          the number of diagonal steps from the starting to the ending points.
13      """
14
15      def __init__(self, diagonal_steps_dimension = 2):
16          """Constructor of the class. Take the half the number of steps as
17          parameter."""
18          self.__lattice_dimension = diagonal_steps_dimension
19          self.__possible_paths_count = int(binom(2 * diagonal_steps_dimension,
20                                                  diagonal_steps_dimension))
21          self.__upper_bound = self.maximal_integer_value_range()
22          self.__lower_bound = self.minimal_integer_value_range()
23          self.__integer_path_list = []
24          self.__after_expansion_lower_bound = self.
    __get_after_expansion_minimal_integer_value_range()
25          self.__lower_value_range_max_exponent_value = self.
    __get_lower_value_range_max_exponent()
26
27      def __get_lower_value_range_max_exponent(self):
28          """
29          Returns the higher power of 2 that lower bound the minimal value range.
30          """
31          exponent = 0
32          is_bounded = False
33          while not is_bounded:
34              if (self.__lower_bound // 2**exponent) != 0:
35                  exponent += 1
36              else:
37                  is_bounded = True
38                  break
39          exponent -= 1
40          return exponent
41
42      def __get_after_expansion_minimal_integer_value_range(self):
43          """Returns the odd integer that correspond to the (11)
44          expansion of the previous ensemble dimension."""
45          exponent = 2 * self.__lattice_dimension - 1
46          return 2 ** exponent + 2 ** (exponent - 1) + 1
47
48
49      def get_dimension(self):
50          """
51          Returns the dimension of the path which is the number of diagonal
52          displacement from starting point to ending point.
53          """
54          return self.__lattice_dimension
55
56      def get_element_count(self):
57          """
58          Returns the number of the paths.
59          """
60          return self.__possible_paths_count
61
62      def get_arrow_path_list(self):
63          """
64          Returns the list of the paths in binary array (->: 0, |:1) representation.
```

```
        """
        return self.__integer_path_list

    def minimal_integer_value_range(self):
        """
        Returns the integer representing the lowest path for a 2N-steps path.
        """
        result = 0
        for i in range(self.__lattice_dimension):
            result += 2**i
        return int(result)

    def maximal_integer_value_range(self):
        """
        Returns the integer representing the highest path for a 2N-steps path.
        """
        result = 0
        for i in range(self.__lattice_dimension):
            result += 2**(self.__lattice_dimension + i)
        return int(result)

    def size_of_integer_searched_value_range(self): # OBSOLETE
        """
        Returns the subset of integer where all representations of a N
        dimensional path lies on."""
        return self.minimal_integer_value_range() * \
            (2**self.__lattice_dimension - 1) + 1

    def print_integer_value_range_infos(self): # OBSOLETE
        """
        Prints out the range of integer for the 2N-steps paths with the
        number of elements in it.
        """
        print("For a path of length 2x", self.__lattice_dimension , " the ",
              self.__possible_paths_count,
              " searched values are to be found in ", sep="", end="")
        print('|[',self.minimal_integer_value_range(),', ', sep="", end="")
        print(self.maximal_integer_value_range(),']| = ', sep="", end="")
        print(self.size_of_integer_searched_value_range(), 'integers.')

    def arrow_to_integer(arrow_path):
        """
        Returns the integer representing the path given in parameter.

        Parameter:
            arrow_path (list(int)): list of 0 (right) and 1 (up).
        """
        result = 0
        path_size = int(np.array(arrow_path).size)
        for i in range(path_size):
            result += arrow_path[i] * 2**(path_size - 1 - i)
        return result

    def integer_to_arrow(integer, starting_exponent = 0):
        """
        Returns a list of 0 (right) and 1 (up) associated with the given integer.

        Parameters:
            integer (int): the integer to convert into path
```

```
            starting_exponent (int): allows to find the power of 2 that bound
                the integer more quickly. Use that parameter precisely by
                giving the lowest possible value that (upper) bound the integer
        """
        exponent = starting_exponent
        is_bounded = False
        while not is_bounded:
            if (integer // 2**exponent) != 0:
                exponent += 1
            else:
                is_bounded = True
                break
        exponent -= 1 # Compensation of the While loop before exit.

        result = [1] + [0] * exponent
        # result = [True] + [False] * exponent     # Slower
        result_size = np.array(result).size
        # result_size = np.array(result, dtype='bool').size
        integer -= 2**exponent
        exponent -= 1
        index_position = 1

        if integer < 0:
            raise Exception("Integer became negative : ", integer)

        while integer > 0:
            if exponent < 0:
                raise Exception("Exponent became negative : exponent")
            if index_position >= result_size:
                raise Exception("Index outside list range : index_position")

            if integer // 2**exponent != 0:
                result[index_position] = 1
                integer -= 2**exponent
            exponent -= 1
            index_position += 1

        up_arrow_count = np.array(result).sum()
        diff = 2 * up_arrow_count - result_size
        if diff > 0:
            result = (diff * [0] + result)
        return result, np.array(result).size, up_arrow_count

    def __create_odd_integer_path_ensemble_iterator(self, chunk_size = 50):
        """Iterator for the integer paths creation."""

        def __get_higher_power_of_2(integer):
            """Returns the higher power of two included in this integer."""
            arrow_path = Path.integer_to_arrow(integer)[0]
            return len(arrow_path) - arrow_path.index(1) - 1

        def __get_missing_integer_count():
            if self.__lattice_dimension < 3:
                return 0
            return int(binom(2*self.__lattice_dimension - 3, self.
    __lattice_dimension - 3))

        def __recursive_filling_odd_integer_list(list_to_fill, list_to_check =
    [1,1,0,0,0,0,0,1], index_position = 2):
```

```
181             zero_to_convert_count = int(list_to_check.count(0) - len(list_to_check
    )//2)
182             if zero_to_convert_count == 0:
183                 list_to_fill.append(list_to_check)
184                 return 0
185 
186             while index_position < len(list_to_check):
187                 __li = list_to_check.copy()            # Possible memory flaw for
    large arrays
188                 __li[index_position] = 1
189                 while __recursive_filling_odd_integer_list(list_to_fill, __li,
    index_position + 1) != 0: # __li.copy() -> __li
190                     pass
191                 index_position += 1
192             return 0
193 
194         yield "#elements = " + str(self.get_element_count()//2) # Should come from
     counting elements directly
195 
196         # first expansion (00) of odd integer from the previous odd integer
    ensemble
197         odd_integer_list = []
198         for values in self.__read_integer_path_iterator(dimension = self.
    __lattice_dimension - 1):
199             for value in values:
200                 higher_power_of_two = __get_higher_power_of_2(value)
201                 for i in range(higher_power_of_two + 1, 2 * (self.
    __lattice_dimension)):
202                     odd_integer_list.append(value + 2**i)
203                 if len(odd_integer_list) > chunk_size:
204                     yield odd_integer_list[:chunk_size]
205                     odd_integer_list = odd_integer_list[chunk_size:]
206 
207         # second expansion (11) for a 2N-steps odd integer
208         if __get_missing_integer_count() != 0:
209             paths_to_convert_into_integers = []
210             starter_path = [1,1] + [0] * (2 *self.__lattice_dimension - 3) + [1]
211             while __recursive_filling_odd_integer_list(
    paths_to_convert_into_integers, starter_path) != 0:
212                 pass
213             for path in paths_to_convert_into_integers:
214                 odd_integer_list.append(Path.arrow_to_integer(path))
215                 if len(odd_integer_list) > chunk_size:
216                     yield odd_integer_list[:chunk_size]
217                     odd_integer_list = odd_integer_list[chunk_size:]
218 
219         # remaining integers to print after the last chunk
220         if len(odd_integer_list) > 0:
221             yield odd_integer_list
222 
223     def output_odd_integer_path(self, chunk_size=50, file_name_prefix="
    odd_integer_ensemble_v2_N_", file_name_extenstion="txt"):
224         """
225         Outputs a file with the list of odd integers representing a 2N-steps path.
226         """
227         filename = "ensembles\\" + file_name_prefix + "{0}." +
    file_name_extenstion
228         with open(filename.format(self.get_dimension()), 'w') as f:
```

```
            for line in self.__create_odd_integer_path_ensemble_iterator(chunk_size):
                f.write(str(line) + "\n")

    def __read_integer_path_iterator(self, dimension = 0, file_name_prefix="odd_integer_ensemble_v2_N_", file_name_extenstion="txt"):
        """Iterator that read the odd integer ensemble file and outputs list of integer paths."""
        if dimension <= 0:
            dimension = self.__lattice_dimension
        if dimension == 1:
            yield [1]
            return True
        filename = "ensembles\\" + file_name_prefix + str(dimension) + "." + file_name_extenstion
        with open(filename.format(dimension), 'r') as f:
            f.readline() # First line not used
            i = 0
            while i < 10 :
                values = f.readline()[1:-2].split(", ")
                if len(values) < 2 : # TO DO BETTER
                    break
                for j in range(len(values)):
                    values[j] = int(values[j])
                yield values

    def read_value_iterator(self):
        for value in self.__read_integer_path_iterator():
            yield value
```

Polymer class: fisrt algortihm

This is the first instance of the Polymer class with the use of explicit paths' weights.

```
class Polymer():
    """
    Class for the calcul of the log-gamma directed polymer partition function.
    """
    def __init__(self, square_lattice_dimension = 2):
        """
        Constructor of the class. Take the half the number of steps as
        parameter."""
        self.__LATTICE_DIMENSION = square_lattice_dimension
        self.partition_function_value = 0.0

    def partition_function(self, distributed_weights, path):
        """
        Returns the partition function value of two symmetrical paths.
        """
        total_steps = len(path)

        def _calculate_path_weight(direction_array, reverse = False):
            value = 1
            iteration = 0
            position = [0, 0]
            while iteration < total_steps:
                value *= distributed_weights[position[0]][position[1]]
                if direction_array[iteration] == (1 if not reverse else 0):
```

```
                    position[0] += (1 if position[0] != self.__LATTICE_DIMENSION
    else 0)
                if direction_array[iteration] == (0 if not reverse else 1):
                    position[1] += (1 if position[1] != self.__LATTICE_DIMENSION
    else 0)
                iteration += 1
            value *= distributed_weights[self.__LATTICE_DIMENSION][self.
    __LATTICE_DIMENSION]
            return value

        path_count = 0
        while path_count < 2:
            yield _calculate_path_weight(path, path_count == 1)
            path_count += 1  # second reading for the conjugate path
```

Polymer class: second algorithm

The second version of this class is the iterative version that allows a bigger extension of the lattice size than the first version.

```
class Polymer():
    def __init__(self, weights):
        """
        Constructor of the class Polymer given a distributed weights set.

        Parameter
        ---------
        weights : TYPE Array (matrix form)
            DESCRIPTION. The lattice's size is automatically deduced from the
    weights array.
        """
        self.__lattice_shape = weights.shape
        self.__partition_function_values = self.__partition_function_values_reset
    ()
        self.__part_funct_diag_values = []
        self.__calculate_partition_function_values(weights)

    def get_lattice_shape(self):
        return self.__lattice_shape

    def set_lattice_shape(self, lattice_shape):
        self.__lattice_shape = lattice_shape

    def __partition_function_values_reset(self):
        return np.zeros(self.__lattice_shape, dtype=np.float64)

    def get_part_func_values(self):
        return self.__partition_function_values

    def get_part_funct_value_from_weights(self, weights):
        self.set_lattice_shape(weights.shape)
        self.__part_funct_diag_values = []
        self.__partition_function_values = self.__partition_function_values_reset
    ()
        self.__calculate_partition_function_values(weights)
        return self.get_part_func_values()

    def __recursive_calculation(self, pos_indices, weights):
```

```
        """Calculation of all partition function values of the lattice sites (i,j)
    ."""
        try:
            if pos_indices[0] + pos_indices[1] == 0: # Origin (0,0)
                return math.log(weights[pos_indices])
            elif pos_indices[0] == 0 and pos_indices[1] > 0: # Left border (0,n),
    n>0
                return (self.__partition_function_values[0, pos_indices[1] - 1] +
    math.log(weights[pos_indices]))
            elif pos_indices[1] == 0 and pos_indices[0] > 0: # Bottom border (m,0)
    , m>0
                return self.__partition_function_values[pos_indices[0] - 1, 0] +
    math.log(weights[pos_indices])
            elif pos_indices[0] > 0 and pos_indices[1] > 0:  # In the bulk (m,n),
    m,n>0
                value = self.__partition_function_values[pos_indices[0],
    pos_indices[1] - 1]\
                    - self.__partition_function_values[pos_indices[0] - 1,
    pos_indices[1]]
                return self.__partition_function_values[pos_indices[0] - 1,
    pos_indices[1]] \
                    + math.log(weights[pos_indices]) + math.log(1 + math.exp(value
    ))
        except OverflowError:
            Warning("Overflow in partition function with position (%d,%d)"%(
    pos_indices[0],pos_indices[1]))
            return math.inf

     def __calculate_partition_function_values(self, weights):
        for i in range(self.__lattice_shape[0]): #lines
            for j in range(self.__lattice_shape[1]): #columns
                self.__partition_function_values[i, j] = self.
    __recursive_calculation((i,j), weights)
```

Tracy-Widom class

To draw the Tracy-Widom distribution, a snippet of code from https://github.com/yymao/TracyWidom under the MIT licence agreement was include within the project.

Master class

Here is the master class in the study of the 1+1 directed polymer. It makes use of the previous class: InverseGamma, Tracy-Widom and the second Polymer class.

```
import math
from scipy.special import digamma, polygamma
from scipy.stats import gengamma
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np

# add the classes InverseGamma, TracyWidom and Polymer (second)

if __name__ == "__main__":

    ### PARAMETERS ###
```

```
    ### ---------- ###
    SHAPE_PARAMETER = 0.80  # alpha
    MAX_LATTICE_SIZE = 600  # Lambda_{n,n} with n := n_max
    ITERATIONS = 40

    ### CALCULATIONS ###
    ### ------------ ###
    f_gamma = -2*digamma(SHAPE_PARAMETER/2)     # theorems 1, 2 and 3
    g_gamma_factor = -1*(polygamma(2, SHAPE_PARAMETER/2)) # theorem 3

    diag_log_values_dic = {}
    for i in range(len(diag_log_values_dic.keys()), MAX_LATTICE_SIZE):
        diag_log_values_dic[f"diag {i+1}"] = []

    lattice_size = 1
    end_lattice_size = 0

    dist = InverseGamma(SHAPE_PARAMETER)
    first_entry_in_confidence_interval_count = 0
    while lattice_size < MAX_LATTICE_SIZE + 1:
        weights = dist.generate_random_weights((lattice_size+1, lattice_size+1))
        polymer = Polymer(weights)
        iteration = 0
        while iteration < ITERATIONS:
            weights = dist.generate_random_weights((lattice_size+1, lattice_size
+1))
            diag_log_values_dic[f"diag {lattice_size}"].append(polymer.
get_part_funct_value_from_weights(weights)[-1,-1])
            iteration += 1
        value = np.sum(diag_log_values_dic[f"diag {lattice_size}"])/len(
diag_log_values_dic[f"diag {lattice_size}"])/lattice_size
        end_lattice_size = lattice_size
        lattice_size += 1

    # ### PREPARING DATA FOR CHECKING THE PROOFS ###
    # ### ------------------------------------ ###
    data_n = []
    data_log_Zn_mean = []
    data_log_Zn_std = []
    data_var_log_Zn23 = []
    histo_values = [] # for (log(Z)-f_gamma*N)/N^(1/3) distribution

    for i in range(0, MAX_LATTICE_SIZE):
        data = np.array(diag_log_values_dic[f"diag {i+1}"])
        data_n.append(i+1)
        data_log_Zn_mean.append((data.sum())/((i+1)*len(data)))
        data_log_Zn_std.append(np.std(data/((i+1)*len(data))))
        data_var_log_Zn23.append(np.var(data)/((i+1)**(2/3)))
        if i == MAX_LATTICE_SIZE-1:
            for j in range(len(data)):
                histo_values.append((data[j]-((i+1)*f_gamma))/((g_gamma_factor*(i
+1))**(1/3)))

    ### GRAPHIC PART ###
    ### ------------ ###
    FONTSIZE = 16

    limit_value_obtained = (np.array(data_log_Zn_mean[-5:]).sum()/len(
data_log_Zn_mean[-5:]))
```

```
67      print("Limits diff =", f_gamma - limit_value_obtained)
68      MIN = min(data_n)
69      MAX = max(data_n)
70      if MAX > 20 :
71          X_values_thicks = [i for i in range(1,MAX+1) if i == 1 or i%(MAX//10) ==
        0]
72      else :
73          X_values_thicks = data_n
74
75      fig = plt.figure(figsize=(12,16))
76      ax0 = fig.add_subplot(1, 1, 1)
77      line1, = ax0.plot(data_n, [f_gamma] * len(data_n), color="navy")
78      line2 = ax0.errorbar(data_n, data_log_Zn_mean, data_log_Zn_std)
79
80      limit_value_color = ("green" if (limit_value_obtained > f_gamma - 0.05 and
        limit_value_obtained < f_gamma + 0.05) else "red")
81      line3, = ax0.plot(data_n, [limit_value_obtained] * len(data_n), color=
        limit_value_color, linestyle='--')
82      ax0.set_xticks(X_values_thicks)
83      xlims = ax0.get_xlim()
84      ylims = ax0.get_ylim()
85      ax0.set_xlabel("n", loc='right', fontsize=FONTSIZE)
86      ax0.set_ylabel("T1", loc='top', fontsize=FONTSIZE)
87      ax0.tick_params(axis='both', labelsize=FONTSIZE)
88
89      xlims = ax0.get_xlim()
90      xy = (xlims[0], f_gamma - 0.05)
91      rect = mpatches.Rectangle(xy, xlims[1]-xlims[0], 0.10, color="#AAEEAA33", ec="
        none", zorder=-5)
92      region = ax0.add_patch(rect)
93      legend_order_set = [line1, line2, line3, region]
94      legend_label_set = ["expected limit value of -2  (  /2)", "mean value of Log(Z
        (n,n))/n",\
95                          "limit value obtained numerically", "Confidence interval
        of  0 .05"]
96      ax0.legend(legend_order_set, legend_label_set, loc="center", fontsize=(
        FONTSIZE+4), facecolor="#EEEEEE", edgecolor="#FFFFFF00", framealpha=1.0)
97
98      ax0.text(MAX/(MAX + 1) , f_gamma, "%.2f "%(f_gamma), color="navy",
        verticalalignment="center", horizontalalignment="right")
99      ax0.text(MAX, limit_value_obtained, "  %.2f"%(limit_value_obtained), color=
        limit_value_color, verticalalignment="center", horizontalalignment="left")
100     fig.suptitle("Shape parameter   =%.2f ; Iterations count=%d"%(SHAPE_PARAMETER,
         ITERATIONS), fontsize=FONTSIZE)
101
102     limit_var_obtained = (np.array(data_var_log_Zn23).sum()/len(data_var_log_Zn23)
        )
103
104     fig1 = plt.figure(figsize=(12,16))
105     ax1 = fig1.add_subplot(1, 1, 1)
106     line1, = ax1.plot(data_n, data_var_log_Zn23, color="blue")
107     ax1.set_xticks(X_values_thicks)
108     xlims = ax1.get_xlim()
109     max_var_value = np.max(data_var_log_Zn23)
110     min_var_value = np.min(data_var_log_Zn23)
111     codomain_range = max_var_value - min_var_value
112     ax1.set_ylim(min_var_value - codomain_range/3, max_var_value + codomain_range
        /3)
113     xy = (xlims[0], min_var_value)
```

```
114     rect = mpatches.Rectangle(xy, xlims[1]-xlims[0], max_var_value - xy[1], color="#00008022", ec="none", zorder=-5)
115     region = ax1.add_patch(rect)
116     ax1.text(xlims[0] + (xlims[1] - xlims[0])/40, max_var_value, "%.2f"%(max_var_value), fontsize=FONTSIZE, color="#000080BB", verticalalignment="bottom", horizontalalignment="left")
117     ax1.text(xlims[0] + (xlims[1] - xlims[0])/40, min_var_value-codomain_range/100, "%.2f"%(min_var_value), fontsize=FONTSIZE, color="#000080BB", verticalalignment="top", horizontalalignment="left")
118     mean_value, = ax1.plot(data_n, [limit_var_obtained] * len(data_n), color="orange", linestyle='--', linewidth=3, zorder=100)
119     ax1.text(MAX/(MAX + 1) , limit_var_obtained, "%.2f "%(limit_var_obtained), color="darkorange", verticalalignment="center", horizontalalignment="right", zorder=100, fontsize=FONTSIZE)
120     ax1.set_xlabel("n", fontsize=FONTSIZE, loc='right')
121     ax1.set_ylabel("T2", fontsize=FONTSIZE, loc='top')
122     ax1.tick_params(axis='both', labelsize=FONTSIZE)
123     legend_order_set = [line1, region, mean_value]
124     legend_label_set = ["value of Var(Log(Z(n,n)))/n^2/3", "codomain region of the function", "Mean value of the function"]
125     ax1.legend(legend_order_set, legend_label_set, loc='upper right', fontsize=(FONTSIZE+4))
126     fig1.suptitle("Boundary region of the free energy's variations:   =%.2f, iterations count=%d"%(SHAPE_PARAMETER, ITERATIONS), fontsize=FONTSIZE)
127
128     def plot_histo_TW(histo, beta=2):
129         x = np.linspace(-10, 10, 501)
130         tw = TracyWidom(beta)
131         pdf = tw.pdf(x)
132
133         if beta==2:
134             color = "red"
135         elif beta==1:
136             color = "orange"
137         elif beta==4:
138             color = "green"
139         else:
140             color = "blue"
141
142         fig2 = plt.figure(figsize=(16,12))
143         ax21 = fig2.add_subplot(2, 1, 1)
144         ax21.hist(histo, density=True, bins=15, color="lightgrey")
145         ax21.plot(x, pdf, color=color, label=f"GUE Tracy-Widom distribution (  ={beta})")
146         ax21.set_title("Fluctuations values for n=%d and 15 bins."%(end_lattice_size), fontsize=FONTSIZE)
147         ax21.tick_params(axis='both', labelsize=FONTSIZE)
148         ax21.legend(loc="upper right", fontsize=FONTSIZE)
149         ax22 = fig2.add_subplot(2, 1, 2)
150         ax22.hist(histo, density=True, bins=30, color="lightgrey") # TEMP
151         ax22.plot(x, pdf, color=color, label=f"GUE Tracy-Widom distribution (  ={beta})")
152         ax22.set_title("Fluctuations values for n=%d and 30 bins."%(end_lattice_size), fontsize=FONTSIZE)
153         ax22.tick_params(axis='both', labelsize=FONTSIZE)
154         fig2.suptitle("Histogram of fluctuations values for shape parameter   =%.2f, %d iterations."%(SHAPE_PARAMETER, len(data)), fontsize=FONTSIZE)
155         plt.show()
156     plot_histo_TW(histo_values)
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté des sciences
Place des sciences, 2 bte L6.06.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/sc**