Karl-Franzens University Graz

Bachelor Thesis (BSc)

An algorithmic approach to calculate geodesics in quantum gravity

David Kneidinger
11828451

supervised by

Univ.-Prof. Dipl.-Phys. Dr.rer.nat. Axel Maas

November 21, 2021

Contents

1 Abstract

Since space-time in quantum gravity is assumed to be isotropic and homogenous, physical observables can be expressed by just one scalar quantity- for example the geodesic distance between two events. The aim of this work is to find these geodesic distances, in a lattice of gaussian distributed space-time events. The approach in the following is to use the A*-algorithm, a shortest path algorithm, to find the geodesic distance by maximizing the proper time along the path. With the FMS mechanism the invariant distance is split into a classical part and a quantum correction part. With the used algorithm, the quantum correction part is calculated for flat Minkowski-space.

2 Introduction

In the beginning of the last century, two new theories- general relativity (GR) and quantum mechanics (QM), were born. Both have greatly widened our understanding of the physical world, led to great discoveries and opened up the exploration of new branches in physics. QM for instance led to atomic physics, particle physics, condensed matter physics, lasers , computers and so on. GR on the other hand led to relativistic astrophysics, cosmology, GPS technology and many more. While both theories produce excellent results with an astounding degree of accuracy, when applied inside their framework, they where both formulated in terms of asumptions contradicted by the other theory [2]. However, already from a pragmatic point of view the question arises if an overlapping domain of quantum gravitational phenomena exists and if so, how we could describe and observe it. It is argued that an interface of both frameworks is needed to provide a satisfying description of the microstructure of spacetime together with matter at the so-called Planck scale. This is the natural scale where effects of quantum gravity are expected to occur [3]. In order to deal with such phenomena a theory of both GR and QM is needed - a theory of quantum gravity.
Such a theory should be able to make accurate predictions to be checked against observation. Until now there is no experiment or observation known, which definetely needs a quantum theory of gravity for its explanation. However, just as classical mechanics recovers from quantum mechanics for large quantum numbers according to Bohr's correspondence principle and GR reduces to Newtonian Gravity for weak gravitational fields. Hence it's plausible to expect, that a theory of quantum gravity reproduces the results of QFT and GR in the according limits. Therefore 'correspondence principle' is often used as a guide and a test for a potentially successful quantum theory of gravity [4].
A possibilty to do so is to construct an object which is invariant under all gauge symmetries, and therefore becomes a physical observable [5] [1], which can be measured, for example at the LHC.
In chapter 3, a brief review of an approach where flat-space-time quantum field theory emerges as a diffeomorphism-invariant limit of quantum gravity and then a way to make it systematically possible to calculate the measurable mass of the elementary Higgs particle, is made [1].
The aim of this thesis is to evaluate geodesics, by using the A*-Algorithm on a lattice of space-time-events in order to construct such a diffeomorphism invariant quantity, depending only on the distance between two events x and y and thus making the above mentioned calculation possible.
How the space-time-events are arranged on the lattice and how the neighbouring relations between these events look like is described in chapter 4.
The algorithm used for these calculations is described in chapter 5.
The result of the calculations can be seen in chapter 6, followed by a final summary and an outlook on what can be pursued in future theses.

[1] Note that Gauge-invariant quantities are not necessarily observable, but all observables must be gauge-invariant. Such observables are called Dirac obserables

3 Theoretical Background

The following chapter should give a brief theoretical background on how observables may be constructed in quantum gravity and the motivation behind the attempt of finding geodesic connections with the A*-Algorithm. For further information see [1].

3.1 The problem of observability in quantum gravity

By merging GR and QM, in quantum gravity space-time itself becomes quantized and therefore undergoes quantum fluctuations at small scales. In classical physics space-time points are determined by particles, but particles themselves are subjects of QM and therefore a subject of quantum fluctuations. So in this context the concept of space-time points becomes 'fuzzy' [5]. So how can physical observables be constructed? On the one hand, the expectation value of a gauge variant operator vanishes [2] [1]. Only allowing for the trivial value of zero, making the expectation values in a sense invariant under gauge transformations. But since this value vanishes, it can't represent a physically meaningful quantity. Hence gauge variant operators can't represent physical observables.
In GR the gauge group is the diffeomorphism [3] group of the underlying manifold [7]. So for an observable to be physical it must be invariant under both diffeomorphism and gauge transformations [1].

3.2 Constructing an observable

[4]

The simplest example to construct physical objects as mentioned above are operators, which are completely scalar. For example O_1 describing the physical Higgs particle, depending on the event x:

$$O_1(x) = \phi^\dagger(x)\phi(x) \tag{1}$$

The fundamental quantity describing the particle is the propagator:

$$D(x,y) = \left\langle O(y)^\dagger O(x) \right\rangle . \tag{2}$$

Where the points x and y denote the events, not the coordinates. Such a coordinate independence is diffeomorphism-invariant and therefore physical.

[2] The path integral is essentially a sum over all possible metrics, where each metric has the same weight, since the weight factor, i.e. the action, is invariant. This means that no metric is distinct. An integration over a non-invariant quantity can then be seen as the integration of a vector over a sphere. With no direction distinguished, the integration will yield a vanishing value. [6]

[3] Diffeomorphism invariance refers to the form invariance of tensor(-equations)s under diffeomorphisms. A diffeomorphism Φ is a one-to-one mapping of a differentiable manifold M (or an open subset) onto another differentiable manifold N.

[4] [1] Chapter 2, 3

In flat-space quantum field theory, space-time is static and therefore the propagator (2) depends on the distance between the two points. In quantum gravity, space-time is expected to be homogeneous and isotropic on average as well. Therefore only one quantity is needed for a diffeomorphism-invariant characterization of the relation between two events x and y. Thus the distance between the two events x and y itself becomes an expectation value.

So for every arrangement of events there should be a unique [5] geodesic distance of the two events x and y. Therefore a uniquely defined expectation value for an invariant length r can be defined as:

$$r(x,y) = \left\langle \min_{z(t)} \int_x^y dt g^{\mu\nu} \frac{dz_\mu(t)}{dt} \frac{dz_\nu(t)}{dt} \right\rangle . \tag{3}$$

In this the minimization over the path $z(t)$ connecting x and y should state to find the geodesic length. With (3) the propagator (2) can be expressed as a function of the invariant distance $r(x,y)$, $D(r(x,y))$. With this an invariant under all gauge symmetries, both local and space-time, is created and hence a physical object.

In this work a numerical approach is made to find the expectation value of such a geodesic connection (3).

[5] Note that uniqueness is not necessarily fulfilled, depending on the space-time and the chosen points to connect.

3.3 Geodesics in flat Minkowski-space

With $\eta_{\mu\nu}$, the flat Minkowski metric[6], the line element for timelike paths $z^\nu(\tau)$ can be written as:

$$d\tau^2 = \eta_{\mu\nu} dz^\mu(\tau) dz^\nu(\tau) \tag{4}$$

(4) yields an invariant expression under general coordinate transformations, where the world-line $z(\tau)$ is parameterized by an affine parameter such as proper time τ [7]. This leads to the coordinate-invariant Lagrangian for timelike paths:

$$S[x] = \int d\tau = \int \sqrt{\eta_{\mu\nu} dz^\mu(\tau) dz^\nu(\tau)} = \int \sqrt{\eta_{\mu\nu} \frac{dz^\mu(\tau)}{d\tau} \frac{dz^\nu(\tau)}{d\tau}} d\tau \tag{5}$$

Note that in (5) $d\tau$ is used instead of the squared quantitiy $d\tau^2$, this makes no difference and both lead to the same geodesic equations, but for later the expression with the square root is of advantage. Varying the functional leads to the well know geodesic equations. It can be shown that timelike Geodesics are of maximum proper time [8] and represent straight lines in the Minkowski-diagram. Since all the connections between two neighbours are in this case straight lines and therefore local geodesics, the search for the minimal distance for timelike paths does not yield a correct result. But as stated above, timelike geodesics are paths of maximum proper time and this enables the search for the timelike path that has the longest proper time and is therefore the best approximation for the geodesic line.
The above stated maximization of proper time can easily be seen by the reversed triangular equation valid for timelike vectors $\mathbf{v}$ and $\mathbf{w}$ with the same time orientation (i.e., $\mathbf{v} \cdot \mathbf{w} < 0$).

$$\|\mathbf{u} + \mathbf{w}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\| \tag{6}$$

[9]

[6] in this work the notation $(+,-,-,-)$ is used

[7] If one affine parameter λ works well, then the geodesic equations are also satisfied for any other affine parameter $\lambda' = a\lambda + b$, where a and b are constants

3.4 FMS mechanism and splitting the metric

[8]

In order to calculate the above mentioned observables, a metric split and the FMS mechanism is used. A metric split, splits the metric in two parts- a classical part $g^c_{\mu\nu}$ and a fluctuation part $\gamma_{\mu\nu}$, as follows:

$$g_{\mu\nu} = g^c_{\mu\nu} + \gamma_{\mu\nu} \tag{7}$$

In order for such a split to be valid, the classical part has to be dominant, which means as much as, the fluctuations must be small in comparison. Before performing the split (7), it is necessary to fix a gauge [1]. The exact procedure will not be encountered here. However it should be mentioned, that after gauge-fixing the expectation value of $\langle g_{\mu\nu}(x) \rangle$ is just the classical metric:

$$\langle g_{\mu\nu}(x) \rangle = g^c_{\mu\nu}(x) \tag{8}$$

The idea of the FMS mechanism is to take an operator (just like (1)) and insert the split (7) into its expression. If the fluctuations around the classical metric are small, then this can be used in quantum gravity [1].

Applying the expansion yields for the invariant distance r:

$$r = \min_{z(t)} \int_x^y dt g^{\mu\nu}_c \frac{dz_\mu(t)}{dt} \frac{dz_\nu(t)}{dt} + \left\langle \min_{z(t)} \int_x^y dt \gamma^{\mu\nu} \frac{dz_\mu(t)}{dt} \frac{dz_\nu(t)}{dt} \right\rangle = r^c + \rho \tag{9}$$

Where r^c is the geodesic distance of the classical metric $g^c_{\mu\nu}$, to which a quantum correction ρ is added. This can also be used to check the usefulness of the expansion ($|\rho/r^c| \ll 1$, for r^c not lightlike). It should be noted that $\gamma_{\mu\nu}$ may not be small on individual configurations and can fluctuate locally widely, but all these fluctuations compensate on average[1]. For $g^c_{\mu\nu} = \eta_{\mu\nu}$ flat-space-time quantum field theory recovers at leading order, resulting in the corresponding flat-space propagator.

$$D(x,y) = D(r) = \left\langle O(y)^\dagger O(x) \right\rangle_{\eta^c_{\mu\nu}} (r^c) + \mathcal{O}(\gamma_{\mu\nu}) \tag{10}$$

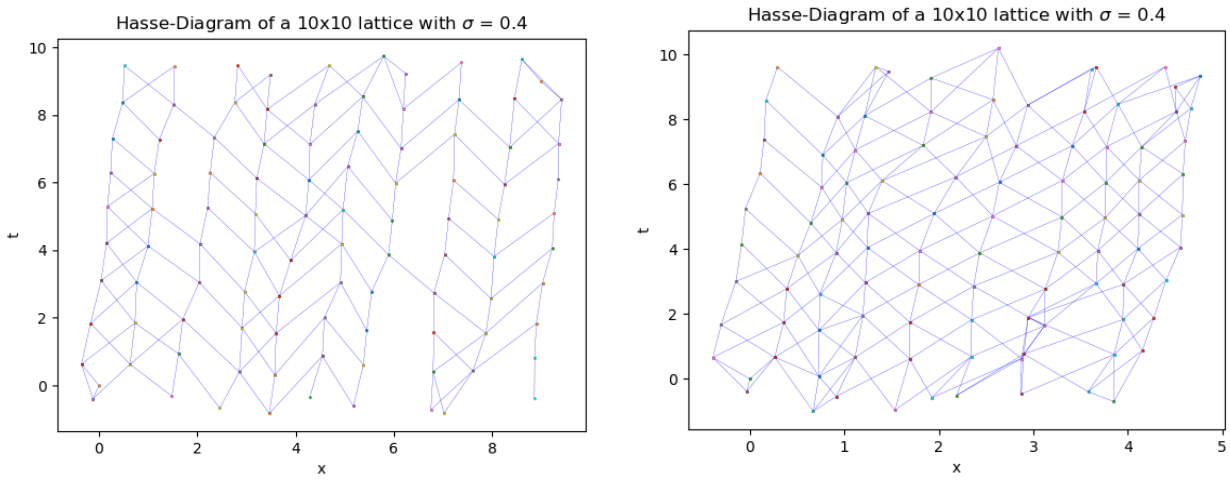
[8] [1] Chapter 4

4 Setup

In this chapter the setup needed for calculating the geodesic distances is given.

4.1 Lattice

In the following space-time is considered as a collection of events, which can be enumerated (t_i, x_i) and has definite neighboring relations. For the sake of simplicity the dimension of the lattice is set to two, in order to decrease the complexity of the calculations and reduce computing time. The configuration of the lattice looks as follows: The events [9] are located in a square lattice with a spacing, which is Gaussian distributed in both the $\hat{\mathbf{t}}$- and the $\hat{\mathbf{x}}$-direction. In order to ensure a flat space and not fix it to a certain topology, open boundary conditions need to be used. Each lattice point has eight next neighbours, which gives the lattice a triangular shape. The choice of the triangular structure above the simplier rectangluar structur where each lattice point has four next neighbours, is only due to the larger selection of resulting paths in the lattice [10] and as the calculation is of a purely technical nature, this is a possible course of action. So in order to get the best approximation for timelike paths, with as small grid sizes as possible, a different arrangement should be used, as in the Hasse-Diagramm there are regions with hardly any connections (fig.1 (a)). To fix this, the x-axis is scaled accordingly (see fig.1 (b)). So the connection from the start node (t_0, x_0) to the goal event (t_g, x_g) lies on the diagonal connections of the lattice, resulting in the analytical value for the calculation of the geodesic paths, for $\sigma = 0$ with no fluctuations.



(a) equaly spaced t- and x-axis

(b) scaled x-axis to provide more timelike paths

Figure 1: Hasse Diagramm of a 10x10 Lattice with $\sigma = 0.4$

In Fig. 1, the space-time events in a 10x10 lattice with a standard deviation $\sigma = 0.4$ are shown. The lines between the events represent the causal connections between the events.

[9] The events or lattice points are in this work sometimes denoted as *nodes*, all three are equivalent names

[10] more on this in the later chapter, actually through additional neighbouring conditions the structure of the lattice becomes a tree shape, because not all next neighbours can be visited

5 Algorithm

To find the above stated geodesics, an implementation of A*-algorithm is used. The following general description of the used algorithm and how the lattice of space-time events with the required neighbour relationships, defines a graph, follows closely [10].

5.1 A*-Algorithm

A graph G is defined to be a set $\{n_i\}$ of elements called nodes in this case a set of space-time-events enumerated with $n_i = (t_i, x_i)$ and a set $\{e_{ij}\}$ of directed line segments called arcs. If e_{pq} is an element of the set $\{e_{ij}\}$, then there is an arc from node n_p to node n_q, and that n_q is a successor of n_j. One can assigne a cost to every arc, namely the space-time interval between the events associated with them. The cost of arc e_{ij} is represented by $d(i,j)$ and calculated with (4). Note that the square root in (4) is important to get $d\tau$ instead of $d\tau^2$, because only these sum up to the full length:

$$d(i,j) + d(j,k) = d\tau(i,j) + d\tau(j,k) = d(i,k) \tag{11}$$

A path from n_1 to n_k is an ordered set of nodes $(n_1, n_2, ..., n_k)$ with each n_{i+1} a successor of n_i. There exists a path from n_i to n_j if and only if n_j is accessible from n_i. Since there are lattice configurations where there are no timelike paths from event x to event y it is possible that there are no existing timelike paths in the graph, which does not rule out the fact that there are no paths from another start node to the goal node.

Every path has a cost which is obtained by adding the individual costs of each arc, $d(i, i+1)$, in the path. An optimal path from n_i to n_j is a path having the smallest cost [11] over the set of all paths from n_i to n_j.

The minimum cost path from s to each node encountered can be tracked as follows. Each time a node is expanded, with each successor node n both the cost of getting to n by the lowest cost path found thus far, and a pointer to the predecessor of n along that path, are stored. Eventually the algorithm terminates at some goal node p, and no more nodes are expanded. From that, a minimum cost path from n to p known at the time of termination can be reconstructed, simply by chaining back from t to s through the pointers. An algorithm is called admissible if it is guaranteed to find an optimal path from s to a preferred goal node of s for any graph, if there exists such a path. In order to do so, an evaluation function f is calculated:

$$f(n) = g(n) + h(n) \tag{12}$$

Where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

[11] in this case the biggest maximum proper time, therefore the proper time will be negated

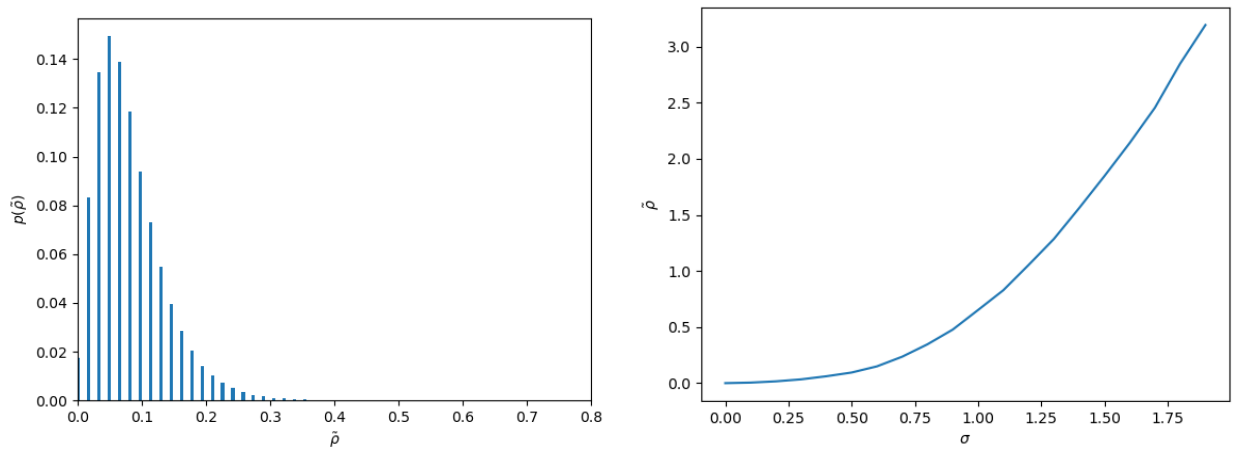
5.2 Properties and drawbacks of A*-Algorithm

The A*-algorithm is guaranteed to find the optimal solution (shortest path), if there exists one in the graph, if there is more than one optimal solution it's just a matter of favoring a solution. For lightlike paths for instance, the one with the least amount of hops between events could be favoured. Also there is no other algorithm to find the solution faster than A*, with the same heuristic function.

Although the stated arguments let the algorithm seem to be flawless, there are of course a few downsides to it as well. As the limiting factor may not be the calculation time, the amount of storage may become an issue. Since all known nodes have to be stored, the amount of data in the memory can quickly exceed the capacity. So for bigger simulations, different approaches have to be made.

5.3 Testing the algorithm in Euclidean space

For Euclidean space the heuristic function $h(n)$ becomes trivial as it should of course be the Euclidean distance from the node n to the goal node. Since each neighbouring node of n represents a possible continuation of the path, the above stated triangular structure of the lattice configuration can be used. The algorithm therefore delivers, in the limit of $\lim_{N\to\infty}$ with N the number of nodes in the lattice, the expected value of the straight line connection the start and end point according the Pythagorean theorem. But since the geodesic here is again a straight line for every start and goal node, the y and x-axes can be scaled such that for $\sigma = 0$ the algorithm gives the exact same value expected from the Pythagorean theorem.



(a) $\sigma = 0.3$ (b) $\tilde{\rho}(\sigma)$

Figure 2: relative frequency of $\tilde{\rho}$ with $\sigma = 0.3$ and $\tilde{\rho}(\sigma)$
with $\tilde{\rho} = \tilde{r} - \tilde{r_c}$ and $\tilde{r}$ the calculated geodesic distance and $\tilde{r_c}$ the straight line
10x10 lattice from the start node $(t_0, x_0) = (0, 0)$ to the goal node $(t_g, x_g) = (9, 9)$

In fig.2 (a) distributions of the difference between the calculated shortest distance and the expected one, is shown for $\sigma = 0.3$ In fig.2 (b) the difference $\tilde{\rho}$[12] of the calculated geodesic distance $\tilde{r}(x, y)$ and the straight line $\tilde{r_c}(x, y)$ is plotted for different values of σ.

[12]the tilde $\sim$ refers to the quantities calculated with every run, whereas the quantities without the tilde symbol refer to the mean value or the expectation value respectively

5.4 Problems and solution with an indefinite metric

In order to get a physical result, only purely timelike paths are considered for now. To ensure that only timelike connections are available in the lattice, only neigbours of a node n are considered which have a timelike connection ($d\tau > 0$) to n. In practice this should be done beforehand, otherwise a dynamical version of A* (called D*) would provide a shorter time complexity. For the sake of simplicity, it has not been done in that way, still A* was used.

Getting a valid heuristic function $h(n)$ is necessary for the algorithm to work. It turned out that the negative space-time interval, particularly the negative proper time for timelike paths is a good choice.
A heuristic has to fulfill two properties in order to be valid. First it has to be admissible, which means it must not overestimate the distance from the current node to the goal. Which means the heuristic h should always give a smaller or equal value, than the actual distance from a current node to the finish node in the graph. By looking at the reversed triangular equation (6), it can be seen that the admissibility is given for timelike paths and the negative proper time interval to the goal node as a heuristic function. If the heuristic function satisfies $h(n) \leq d(n,k) + h(k)$ for every arc e_{nk}, then h is called monotone or consistent and A* guaranteed to find an optimal path without processing any node more than once. Which is again fulfilled for the negative proper time interval.

In order to find the timelike geodesics, the above stated maximization of the proper time τ has to be done. This leads to the problem of finding longest paths. Altough shortest path algorithms and longest path algorithms are in general of completely different structure, the A*-Algorithm can still be used to find the longest path by negating the edge costs $d(i,j)$ because the following statement does hold:

$$\sum_i (-a_i) \geq \sum_j (-b_j) \Longrightarrow \sum_i a_i \leq \sum_j b_j \qquad a_i,\, b_j \in \mathbb{R} \tag{13}$$

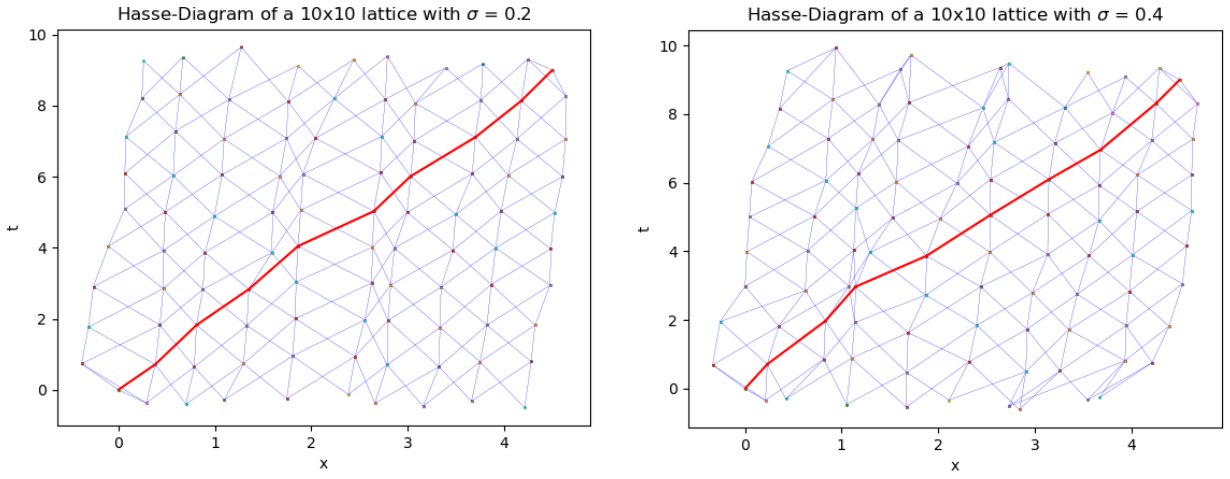
Since the above stated tree shape of the path does not allow for negative cycles [13] A*-Algorithm can still be used with negative edge costs. If the distinction between timelike and spacelike edges were not be made, one would have to switch to another algorithm which recognizes negative cycles such as the Bellman-Ford algorithm.

[13] If a graph contains a *negative cycle* (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no shortest path in the graph since any path that has a point on the negative cycle can be made shorter by one more walk around the negative cycle

6 Results

In the following, all results refer to a 10x10 lattice, with σ the standard deviation of the lattice points. The x-axis is scaled by a factor of $\frac{1}{2}$ compared to the time axis. The geodesics are calculated from the start node $(t_0, x_0) = (0, 0)$ to the goal node $(t_g, x_g) = (9, 9)$ with scaled x-axis so node $(9, 9) \rightarrow (9, 4.5)$, where the left side corresponds to the enumartion (t_i, x_i) and the right side to the coordinates. This is chosen for simplicity and does not affect the results in any way. The calculation could be performed for any other combination of timelike connected events x and y. The relatively small lattice size of 10x10 events is due to much shorter computing time compared to bigger lattices, which comes quite handy, when generating statistics.

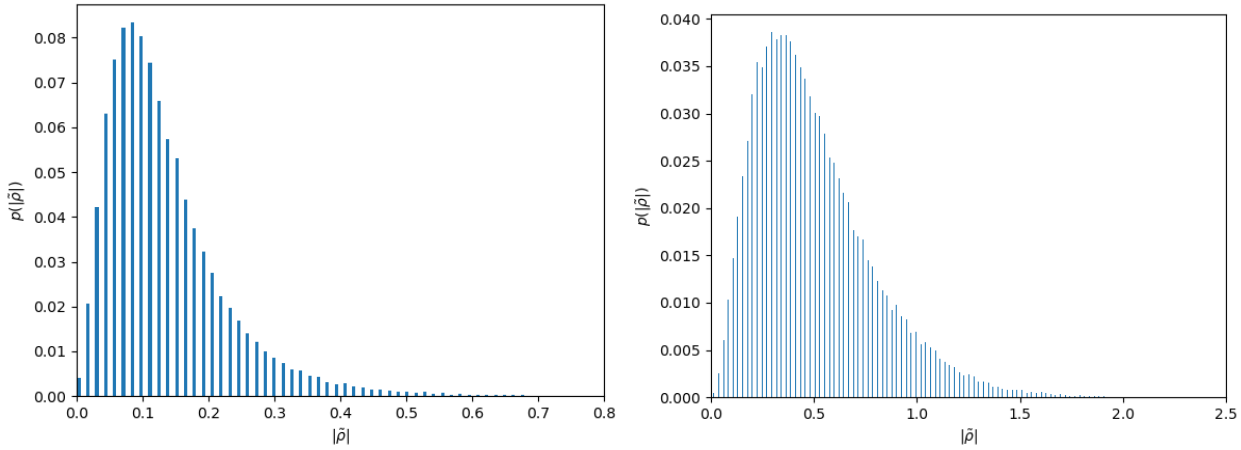
In fig.3 the timelike geodesics for $\sigma = 0.2$ and $\sigma = 0.4$ are shown. It can be seen that for the bigger σ (bigger fluctuations) the geodesic differs more from the straight line.



(a) $\sigma = 0.2$ (b) $\sigma = 0.4$

Figure 3: Geodiscs in two different lattices
with a scaled x-axis to provide more timelike paths

The geodesic distance $\tilde{r}$ was computed for a fixed start- and goal node, with a certain σ. For every computation of $\tilde{r}$, the lattice was created with the mentioned constrains and the difference of the geodesic connection $\tilde{r}$ and the classical solution $\tilde{r_c}$, $\tilde{r} - \tilde{r_c} = \tilde{\rho}$ (eq. (9)) was calculated. In fig.4 the relative frequency of $|\tilde{\rho}|$ is shown for $\sigma = 0.2$ and $\sigma = 0.4$. It can be seen that for bigger fluctuations, the density function $p(|\tilde{\rho}|)$ spreads out wider.



(a) $\sigma = 0.2$ (b) $\sigma = 0.4$

Figure 4: relative frequency of $|\tilde{\rho}|$
with $\tilde{\rho} = \tilde{r} - \tilde{r_c}$ and $\tilde{r}$ the calculated geodesic distance and $\tilde{r_c}$ the theoretical one (straight line)

As seen in fig.5, the distribution of $|\tilde{\rho}|$ is best described by a Gumbel distribution, which in the general form looks as follows:

$$f(x) = \frac{1}{\beta} e^{-e^{-\frac{1}{\beta}(x-\mu)}} \cdot e^{-\frac{1}{\beta}(x-\mu)}. \tag{14}$$

The expectation value of the Gumbel distribution is given as follows:

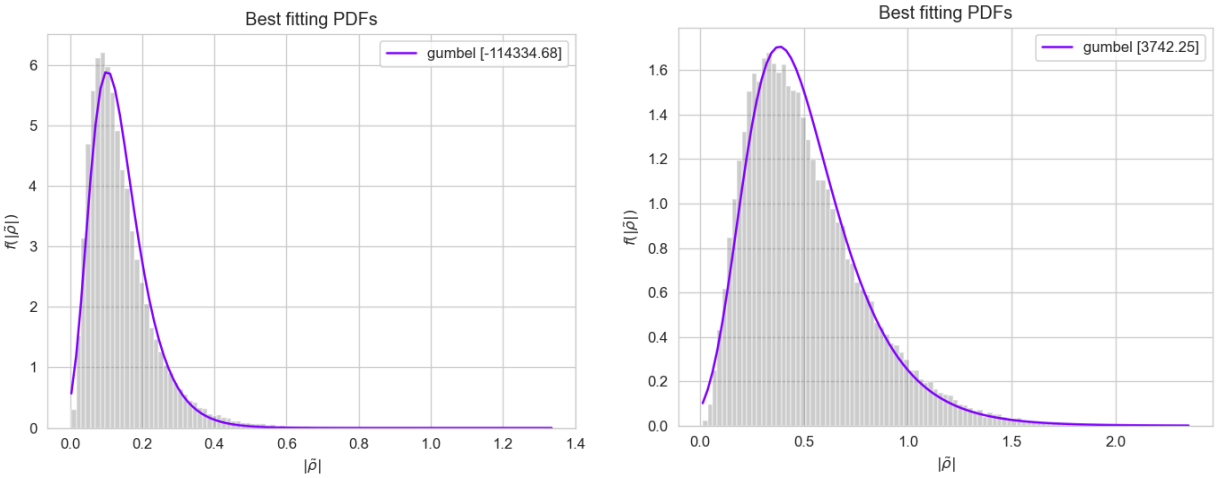
$$E(\tilde{\rho}) = \rho = \mu_\rho + \beta\gamma = \left\langle \min_{z(t)} \int_x^y dt \gamma^{\mu\nu} \frac{dz_\mu(t)}{dt} \frac{dz_\nu(t)}{dt} \right\rangle \tag{15}$$

with $\gamma \approx 0.5772$ the Euler–Mascheroni constant and μ_ρ the mean value of the Gumbel distribution. Thus ρ can be calculated accordingly.

Table 1: values for r and ρ for different fluctuations

σ: standard deviation of the gaussian distributed lattice points
r: calculated geodesic distance (eq 9)
r^c: classical solution for the geodesic distance
ρ: quantum correction due to the fluctuations

σ	0.0	0.1	0.2	0.3	0.4
r	7.794	7.777	7.736	7.678	7.622
r^c	7.794	7.794	7.794	7.794	7.794
ρ	0.000	-0.028	-0.122	-0.274	-0.438
$\lvert\frac{\rho}{r^c}\rvert$	0.0	3.6E-03	1.6E-02	3.5E-02	5.6E-02



(a) $\sigma = 0.2$ in the gaussian distributed lattice

(b) $\sigma = 0.4$ in the gaussian distributed lattice

Figure 5: fitted relative frequency of $|\tilde{\rho}|$
with $\tilde{\rho} = \tilde{r} - \tilde{r_c}$ and $\tilde{r}$ the calculated geodesic distance and $\tilde{r_c}$ the theoretical one

In fig.6 $\tilde{\rho} = \tilde{r} - \tilde{r_c}$ is plotted for different standard deviations σ of the Gaussian distributed lattice points. In contrast to the Euclidean case the value of $\tilde{\rho}$ does not get arbitrarily big for big σ, but instead stays constant at a certain value [14]. This is due to the fact that from a certain value of σ there are no timelike paths between the start- and the goal event in the lattice. But as these are not accounted in the averaging of the geodesic for every value of σ, the graph begins to flatten as less timelike paths can be found in the lattice. Until the point where the graph gets completely horizontal. This is where not a single timelike path can be found anymore.

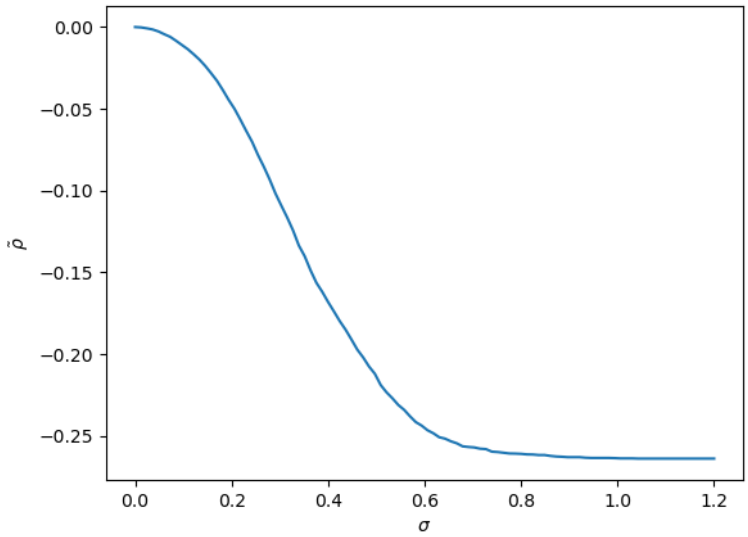


Figure 6: $\tilde{\rho}$ for different values of σ
with $\tilde{\rho} = r - r_c$ and r the calculated geodesic distance and r_c the theoretical one

[14] This value is different for different lattice setups

7 Summary and Outlook

In this thesis, an algorithmic approach for the search of geodesic connections between two events on a lattice has been made. It was shown that a shortest path algorithm, such as A*, can be used for that kind of problem. For the case of flat spacetime, geodesics were calculated on a 10x10 lattice for different values of σ. The used algorithm showed the expected values for no quantum fluctuations ($\sigma = 0$). Distributions of the quantum correction part of the metric split (9) are shown in fig.4. In order to get the expectation value ρ, the distribution was fitted via a Gumbel distribution and the according mean values were calculated.

As the use of the algorithm is not limited to static metrics, it may be developed for further calculations. One interesting aspect is the lattice itself, which in this thesis was just assumed to be Gaussian distributed in both directions. So one of the next steps could be to determine how such fluctuations may look like in reality and build on that. As mentioned in section 5 for bigger simulation, one might have to switch to a different algorithm.

References

[1] A. Maas. *The Fröhlich-Morchio-Strocchi mechanism and quantum gravity.* SciPost Physics **8** (2020).
doi:10.21468/scipostphys.8.4.051

[2] C. Rovelli. *Quantum Gravity.* Cambridge Monographs on Mathematical Physics. Cambridge University Press, 2004.
doi:10.1017/CBO9780511755804

[3] A. G. A. Pithis. Aspects of quantum gravity, 2019.

[4] K. Bradonjić. Quantum Gravity and the Correspondence Principle, 2011.

[5] C. Rovelli. *What is observable in classical and quantum gravity?* Classical and Quantum Gravity **8** (1999) 297.
doi:10.1088/0264-9381/8/2/011

[6] M. Markl. Black Holes as Quantum Phenomena-Master Thesis, 2020.

[7] J. Tambornino. *Relational Observables in Gravity: a Review.* Symmetry, Integrability and Geometry: Methods and Applications (2012).
doi:10.3842/sigma.2012.017

[8] *Relativity I: The Principle of Maximal Proper Time (Eigenzeit).* Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
doi:10.1007/978-3-540-36805-2_9

[9] *Geometrical Structure of M.* Springer New York Dordrecht Heidelberg London, 2010.

[10] P. E. Hart, N. J. Nilsson, B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics **4** (1968) 100.
doi:10.1109/TSSC.1968.300136

8 Appendix

8.1 Flowchart

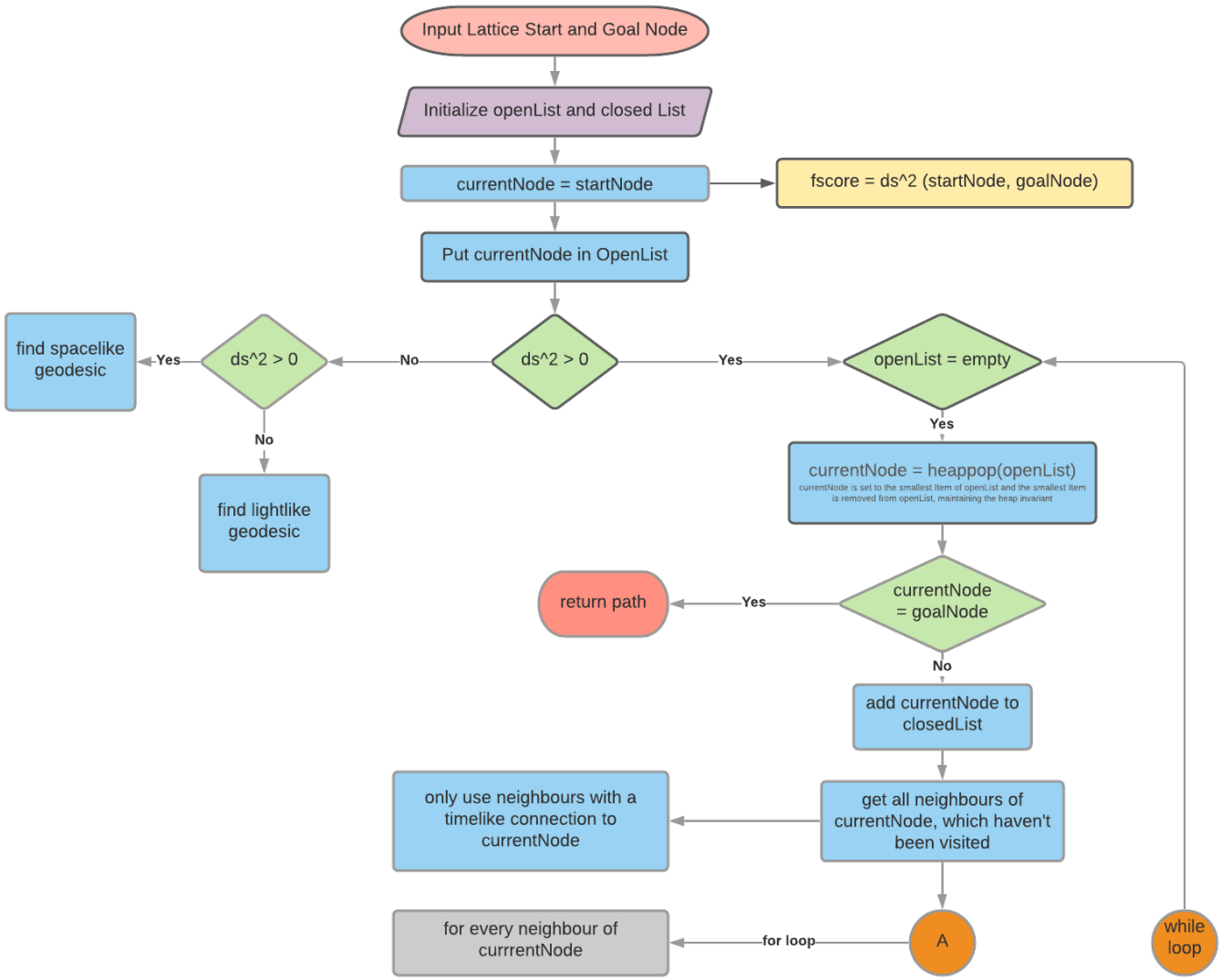


Figure 7: Flowchart 1

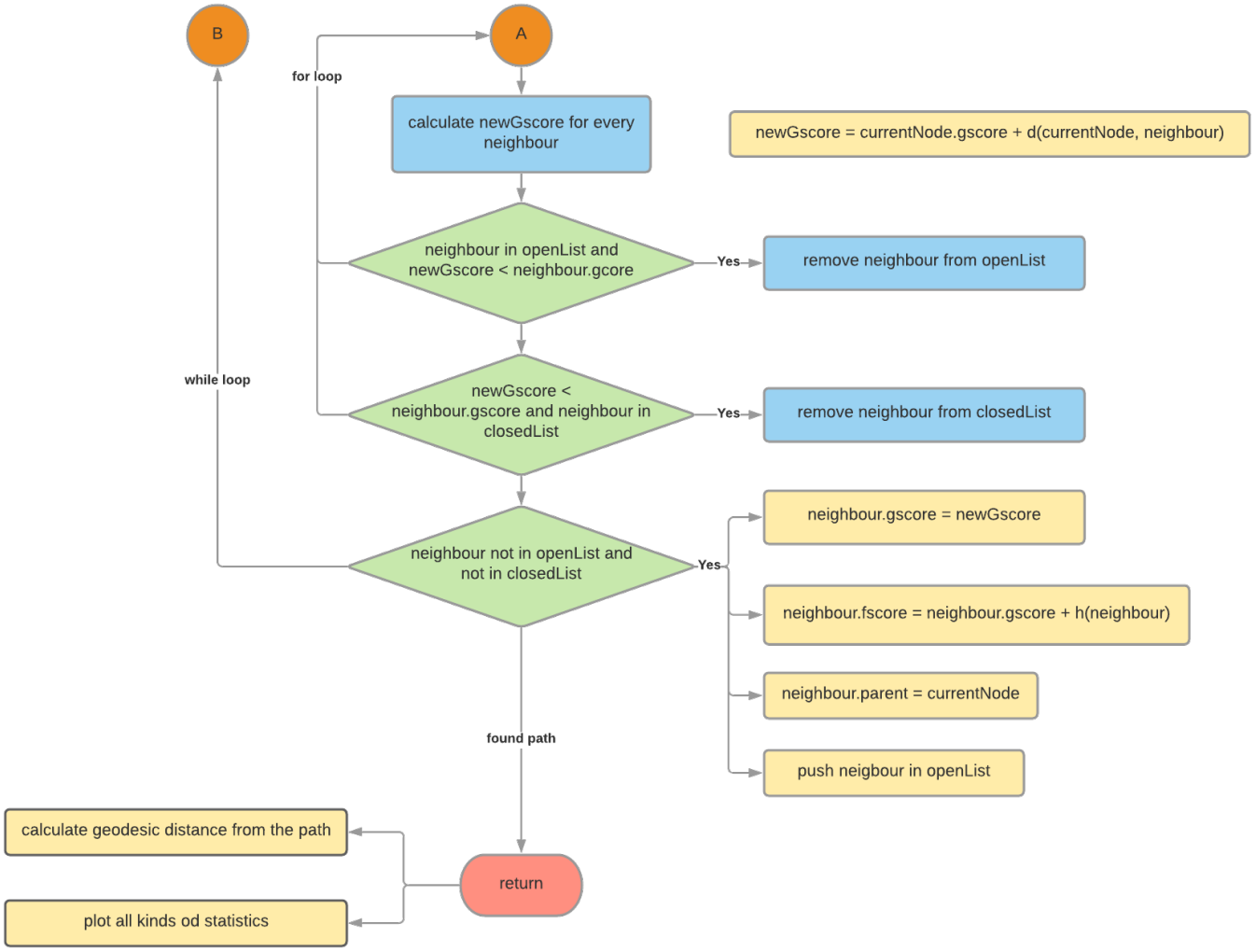


Figure 8: Flowchart 2

8.2 Python Code

Since the main part of the program is implemented in an object-oriented fashion and most of the objects and variables are desribed inside the code via comments, the readability should be given. As such, no further explanation of the variables is made in this context. The program used for the Euclidean case will not be shown here, as it's almost identical to the flat-Minkowski program.

8.2.1 Main Program- A*- Algorithm

The following Code contains the main part- The Algorithm.

```
import heapq
import matplotlib.pyplot as plt
import numpy as np

class Node(object):
    def __init__(self, time, position_x):
        self.time = time
        self.position_x = position_x
```

```
    def distance(self, otherNode): #cost of the edge between two points ->
    ↪  metric
        return (otherNode.time - self.time)**2 - (otherNode.position_x -
        ↪  self.position_x)**2


    @property
    def position(self): #Node.position: Tupel of 2 numbers (x, y)
        return self.time, self.position_x

    def __hash__(self):
        return hash((self.time, self.position_x))

    def __eq__(self, other_node): #recognizes that A -> B is the same edge as
    ↪  B -> A ist
        return self.time == other_node.time and self.position_x ==
        ↪  other_node.position_x

    def __str__(self): #position only with 2 decimals
        return f"({self.time:.2f}, {self.position_x:.2f})"

    __repr__ = __str__

class Lattice(object):
    def __init__(self, spacing_time, spacing_x):
        self.nodes = []
        x_max = len(spacing_x[0])
        t_max = len(spacing_time)
        time_range = np.arange(0, t_max)
        x_range = 0.5 * np.arange(0, x_max)


        if t_max > 1 and x_max > 1:
            for i in range(t_max):
                column = []
                for k in range(x_max):
                    t = time_range[i] + spacing_time[i][k]
                    x = x_range[k] + spacing_x[i][k]
                    column.append(Node(t, x))

                self.nodes.append(column)
                #fixes the position of the start- and the endpoint
            self.nodes[0][0] = Node(time_range[0], x_range[0])
            self.nodes[t_max - 1][x_max - 1] = Node(time_range[t_max - 1],
            ↪  x_range[x_max - 1])


        elif len(spacing_time) == 1:
```

```
        column = []
        column.append((Node(time_range[0], x_range[0])))

        for k in range(x_max):
            t = spacing_time[0][k]
            x = x_range[k] + spacing_x[0][k]
            column.append(Node(t, x))
        column.append(Node(time_range[t_max - 1], x_range[x_max - 1]))

        self.nodes.append(column)

        #fixes the position of the start- and the endpoint
        self.nodes[0][0] = Node(time_range[0], x_range[0])
        self.nodes[t_max - 1][x_max - 1] = Node(time_range[t_max - 1],
        ↪  x_range[x_max - 1])

    elif len(spacing_x[0]) == 1:

        for k in range(t_max):
            column = []
            t = time_range[k] + spacing_time[k][0]
            x = spacing_x[k][0]
            column.append(Node(t, x))

            self.nodes.append(column)

            #fixes the position of the start- and the endpoint
            self.nodes[0][0] = Node(time_range[0], x_range[0])
            self.nodes[t_max - 1][x_max - 1] = Node(time_range[t_max -
            ↪  1], x_range[x_max - 1])


@staticmethod
def gaussDistributedSpacingLattice(loc, spacing, n_t, n_x):
    return Lattice(np.sort(np.random.normal(loc, spacing, size=(n_t,
    ↪  n_x)), axis=0),
                   np.sort(np.random.normal(loc, spacing, size=(n_t,
                   ↪  n_x)), axis=0))



def plot(self, axes):

    for i in range(len(self.nodes)):
        for j in range(len(self.nodes)):
            node = self.getNodeAt((i, j))
            neighbours = self.nodeNeighbours((i, j))
            for otherNode in neighbours:
                axes.plot([node.position_x,
                ↪  self.getNodeAt(otherNode).position_x],
```

```
                        [node.time, self.getNodeAt(otherNode).time],
                        ↪  'b-', linewidth=0.1)
                axes.plot(node.position_x, node.time, 'o',
                ↪  markersize=len(self.nodes)/10)

    axes.set_xlabel("x")
    axes.set_ylabel("t")
    axes.set_title("Lattice")

def plotHasse(self, axes):

    for i in range(len(self.nodes)):
        for j in range(len(self.nodes)):
            node = self.getNodeAt((i, j))
            neighbours = self.nodeNeighbours((i, j))
            for otherNode in neighbours:
                ds_2 = (node.time - self.getNodeAt(otherNode).time) ** 2
                ↪  \
                       - (node.position_x -
                       ↪  self.getNodeAt(otherNode).position_x) ** 2

                if ds_2 > 0:
                    axes.plot([node.position_x,
                    ↪  self.getNodeAt(otherNode).position_x],
                              [node.time,
                              ↪  self.getNodeAt(otherNode).time], 'b-',
                              ↪  linewidth=0.1)
                    axes.plot(node.position_x, node.time, 'o',
                    ↪  markersize=len(self.nodes) / 10)

    axes.set_xlabel("x")
    axes.set_ylabel("t")
    title = "Hasse-Diagram of a " + str(len(self.nodes)) + "x" +
    ↪  str(len(self.nodes)) \
            + " lattice with $\\sigma$ = 0.4"
    axes.set_title(title)

def plotPath(self, axes, path):
    if david_sein_path is not None:
        for step in range(len(path) - 1):
            axes.plot([self.getNodeAt(path[step]).position_x,
            ↪  self.getNodeAt(path[step + 1]).position_x],
                      [self.getNodeAt(path[step]).time,
                      ↪  self.getNodeAt(path[step + 1]).time],
                      'r-')


@property
def n_t(self):
    return len(self.nodes)
```

```
    @property
    def n_x(self):
        return len(self.nodes[0])

    #neighbouring relations

    def nodeNeighbours(self, pos):
        (t, x) = pos


        return [(dt, dx) for (dt, dx) in [(t + 1, x + 1), (t - 1, x + 1), (t
        ↪  + 1, x - 1), (t - 1, x - 1),
                                          (t + 1, x), (t - 1, x), (t, x + 1),
                                          ↪  (t, x - 1)] if
                0 <= dt < self.n_t and 0 <= dx < self.n_x]


    def getNodeAt(self, pos):  # Tupel of (t,x)
        return self.nodes[int(pos[0])][int(pos[1])]


infinity = float("inf")


class AStar(object):
    def __init__(self, start, lattice):
        self.start = start
        self.lattice = lattice

    class Node(object):
        def __init__(self, position, fscore=infinity, gscore=infinity,
        ↪  parent=None):
            self.fscore, self.gscore, self.position, self.parent = fscore,
            ↪  gscore, position, parent

        def __lt__(self, comparator):
            return self.fscore < comparator.fscore

    def getPath(self, endPoint):
        current, path = endPoint, []
        while current.position != self.start:
            path.append(current.position)
            current = current.parent
        path.append(self.start)
        return list(reversed(path))

    def computePath(self, end):
```

```
#A*- algorithm uses a metric called fscore, which is a score to gauge
↪  how fruitful moving to the neighbour
#will be in finding the optimal path.
#fscore = gscore + h, where gscore is the path cost of a node and h
↪  is a heuristic used to estimate distance
#between a node and the goal. -> in this case the negativ proper time
↪  intervall

openList, closedList, nodeDict = [], [], {}
#openList...enthält
#closedList...
start_lattice_node = self.lattice.getNodeAt(self.start) #gets the
↪  startNode
currentNode = AStar.Node(self.start,
↪  fscore=start_lattice_node.distance(self.lattice.getNodeAt(end)),
↪  gscore=0)
#gscore: distance of currentNode to the node (x,y) (neighbour)
#fscore = gscore + h
#h: heuristic function
heapq.heappush(openList, currentNode) #heapq algorithm -> priority
↪  queue algorithm.
#puts currentNode in openList


↪  #-------------------------------------------------------------------------
#calculate the actual shortest distance by Definition (straight line
↪  between start- and endpoint:

actualshortestDistance_2 =
↪  self.lattice.getNodeAt(start_lattice_node.position).distance(
    self.lattice.getNodeAt(end))

if actualshortestDistance_2 > 0:
    actualshortestDistance = np.sqrt(actualshortestDistance_2)
else:
    actualshortestDistance = - np.sqrt(-actualshortestDistance_2)


while openList:

    currentNode = heapq.heappop(openList)
    #removes the bottom element of openList and saves it in
    ↪  currentNode
    if currentNode.position == end:
        return self.getPath(currentNode)
    else:
        closedList.append(currentNode)
        neighbours = []
        for toCheck in
        ↪  self.lattice.nodeNeighbours(currentNode.position):
```

```
            if toCheck not in nodeDict.keys():
                nodeDict[toCheck] = AStar.Node(toCheck)
                neighbours.append(nodeDict[toCheck])
        #neighbours contains all neighbours of currentNode which
        ↪  haven't been visited yet

        if actualshortestDistance_2 > 0:
            # neighbours should only contain Nodes to which the
            ↪  connection is time-like

            neighbours = [x for x in neighbours if

                          ↪  self.lattice.getNodeAt(currentNode.position)
                              .distance(
                              self.lattice.getNodeAt(x.position)) >
                              ↪  0]


    #let time only progress in the forward direction,
            # necessaty for the reverserd triangular equation to hold

            neighbours = [x for x in neighbours if

                          ↪  self.lattice.getNodeAt(currentNode.position)
                              .time <
                          self.lattice.getNodeAt(x.position).time]


            #fscore = gscore - h (sign is flipped in order to find
            ↪  the path with the maximum proper time
            for neighbour in neighbours:
                #newGscore is current Gscore + distance from
                ↪  currentNode to neighbour
                newGscore = currentNode.gscore +
                ↪  np.sqrt(start_lattice_node.distance(
                    self.lattice.getNodeAt(neighbour.position)))

                if neighbour in openList and newGscore <
                ↪  neighbour.gscore:
                    openList.remove(neighbour)
                if newGscore < neighbour.gscore and neighbour in
                ↪  closedList:
                    closedList.remove(neighbour)
                if neighbour not in openList and neighbour not in
                ↪  closedList:
                    neighbour.gscore = newGscore
                    #fscore = gscore + h -> gscore - distance from
                    ↪  neighbour to the goal node
                    neighbour.fscore = neighbour.gscore -
                    ↪  np.sqrt(start_lattice_node.distance(
```

```
                                self.lattice.getNodeAt(end)))
                            neighbour.parent = currentNode
                            heapq.heappush(openList, neighbour)
                        heapq.heapify(openList)
        return None


if __name__ == "__main__":

    r = [] #calculated distance r = r_c + rho
    r_c = [] #classic geodesic distance
    figure = plt.figure()

    sigma = []
    sig_asd = []
    sig_sd = []
    for sig in [0.3]: #value for sigma
        sigma.append(sig)
        for i in range(100):
            axes_ = figure.add_subplot(1, 1, 1)
            david_sein_lattice = Lattice.gaussDistributedSpacingLattice(0,
            ↪  sig, 10, 10) #lattice 10x10
            david_sein_lattice.plotHasse(axes_)
            david_sein_star = AStar((0, 0), david_sein_lattice)  # (t_0, x_0)
            ↪  startNode
            david_sein_path = david_sein_star.computePath((9, 9))
            #(t_g, x_g) goalNode #should not be changed-> scale x- axis
            ↪  accordingly
            david_sein_lattice.plotPath(axes_, david_sein_path)
            figure.show()
            print('shortest path: ', david_sein_path)

            distance = 0
            if david_sein_path is not None:
                for step in range(len(david_sein_path) - 1):
                    x_value = \
                    david_sein_lattice.getNodeAt(david_sein_path[step +
                    ↪  1]).position[1] - \

                    ↪  david_sein_lattice.getNodeAt(david_sein_path[step]).position[1]

                    t_value = \
                    david_sein_lattice.getNodeAt(david_sein_path[step +
                    ↪  1]).position[0] - \

                    ↪  david_sein_lattice.getNodeAt(david_sein_path[step]).position[0]

                    if step == 0:
                        x_start = \
```

```
                    ↪ david_sein_lattice.getNodeAt(david_sein_path[step]).position[1]
                        t_start = \

                    ↪ david_sein_lattice.getNodeAt(david_sein_path[step]).position[0]
                    elif step == len(david_sein_path) - 2:
                        x_goal = \
                    david_sein_lattice.getNodeAt(david_sein_path[step +
                    ↪ 1]).position[1]
                        t_goal = \
                    david_sein_lattice.getNodeAt(david_sein_path[step +
                    ↪ 1]).position[0]

                    ds_2 = t_value ** 2 - x_value ** 2
                    distance = distance + np.sqrt(ds_2)

                r_c.append(np.sqrt((t_start - t_goal) ** 2 - (x_start -
                ↪ x_goal) ** 2))
                r.append(distance)

        sig_asd.append(np.mean(r_c))
        sig_sd.append(np.mean(r))


    from Statistics import bar_plot, sigma_distribution,
    ↪ best_fitting_distribution


    bar_plot(r_c, r)
    best_fitting_distribution(r, r_c)
    sigma_distribution(sig_asd, sig_sd, sigma)
```

8.2.2 Statistics

The following code contains the subprograms to compute the plots and statistics:

```
def bar_plot(r_c, r):
    import numpy as np
    import matplotlib.pyplot as plt

    y_value = []
    for i in range(len(r)):
        y_value.append(r_c[i] - r[i])

    y_min = min(y_value)
    y_max = max(y_value)
    x = np.linspace(y_min, y_max, 100)

    freq = []
    p = []
    for j in range(len(x) - 1):
```

```
            count = 0
            for k in range(len(r)):
                if y_value[k] >= x[j] and y_value[k] <= x[j + 1]:
                    count += 1
                else:
                    continue
            freq.append(count)
            p.append(count/len(r))
        freq.append(0)
        p.append(0)

        plt.figure()
        plt.bar(x, freq, width=0.005)
        plt.xlim(y_min, y_max)
        plt.ylabel('$f$')
        plt.xlabel('$|\\rho|$')
        plt.show()


        plt.figure()
        plt.bar(x, p, width=0.005)
        #plt.xlim(y_min, y_max)
        plt.xlim([0, 0.8])
        plt.ylabel('$p(\\tilde{\\rho})$')
        plt.xlabel('$\\tilde{\\rho}$')
        plt.show()


def sigma_distribution(sig_asd, sig_sd, sigma):
    import numpy as np
    import matplotlib.pyplot as plt
    r = []
    for i in range(len(sig_asd)):
        r.append(-sig_asd[i] + sig_sd[i])
    plt.figure()
    plt.plot(sigma, r)
    plt.ylabel('$\\tilde{\\rho}$')
    plt.xlabel('$\\sigma$')
    plt.show()


def best_fitting_distribution(r, r_c):
    # import matplotlib and set inline
    import matplotlib

    #matplotlib.use("Agg")
    import matplotlib.pyplot as plt

    # import other libraries
    import scipy.stats as st
```

```
    import seaborn as sns
    import numpy as np
    import warnings

    # set seaborn style
    sns.set_style("whitegrid")

    # my distribution
    mydistr = []
    for i in range(len(r)):
        mydistr.append(-r[i] + r_c[i])


    min_sD = min(mydistr)
    max_sD = max(mydistr)
    mydistr_x = np.linspace(min_sD, max_sD, 100)

    # define a set of distributions to check

    '''
    distribution_list = [
        st.alpha, st.anglit, st.cauchy, st.chi2,
        st.dgamma, st.dweibull, st.erlang, st.expon, st.exponnorm,
    ↪ st.exponweib, st.exponpow, st.f, st.fatiguelife, st.fisk,
        st.foldcauchy, st.foldnorm, st.frechet_r, st.frechet_l,
    ↪ st.genlogistic, st.genpareto, st.gennorm, st.genexpon,
        st.genextreme, st.gausshyper, st.gamma, st.gengamma,
    ↪ st.genhalflogistic, st.gilbrat, st.gompertz, st.gumbel_r,
        st.gumbel_l, st.halfcauchy, st.halflogistic, st.halfnorm,
    ↪ st.halfgennorm, st.hypsecant, st.invgamma, st.invgauss,
        st.invweibull, st.johnsonsb, st.johnsonsu, st.ksone, st.kstwobign,
    ↪ st.laplace,
        st.logistic, st.loggamma, st.loglaplace, st.lognorm, st.lomax,
    ↪ st.maxwell, st.mielke, st.nakagami, st.ncx2, st.norm, st.pareto,
    ↪ st.pearson3, st.powerlaw, st.powerlognorm, st.powernorm, st.rdist,
    ↪ st.reciprocal,
        st.rayleigh, st.rice, st.recipinvgauss, st.semicircular, st.t,
    ↪ st.triang, st.truncexpon, st.truncnorm,
        st.tukeylambda,
        st.uniform, st.vonmises, st.vonmises_line, st.wald, st.weibull_min,
    ↪ st.weibull_max, st.wrapcauchy
    ]
    '''
    distribution_list = [
        st.cauchy, st.hypsecant, st.laplace,
        st.logistic, st.norm, st.gumbel_r, st.gumbel_l
    ]
```

```
# this list will contain the result of the fittings
distribution_fitting = []


# fit every distribution in the list above
for ref_distr in distribution_list:

    # pick distribution name
    distr_name = type(ref_distr).__name__.split("_")[0]

    try:
        # ignore warnings
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")

            # get parameters from the best fit
            params = ref_distr.fit(mydistr)
            mu, sigma = ref_distr.fit(mydistr)
            arg = params[:-2]
            loc = params[-2]
            scale = params[-1]

            # build the PDF from previous parameters
            pdf_fitted = ref_distr.pdf(mydistr_x, loc=loc, scale=scale,
            ↪  *arg)

            # calculate maximum likelihood estimator
            mle = ref_distr.nnlf(params, mydistr)

            # as an alternative: sum of square error
            sse = np.sum(np.power(mydistr_x - pdf_fitted, 2.0))

            # add results to list
            distribution_fitting.append({
                "distr_name": distr_name, "mle": mle, "sse": sse,
                "pdf_fitted": pdf_fitted, "params": params})

        # ignore distributions that could not be fitted
    except Exception:
        print("Discarded distribution: {}".format(distr_name))

# plot data histograms
fig, ax = plt.subplots(1, 1, dpi=120)
_ = ax.hist(mydistr, mydistr_x, density=1, color="gray",  alpha=0.4)


# set stuff for this axis
top = 1
ax.set_prop_cycle('color', plt.cm.rainbow(np.linspace(0, 1, top)))
ax.set_title("Best fitting PDFs")
```

```
# just print TOP distributions according to maxim. likelihood estimator

sort_by = "mle"  # or "sse" for least square or "mle" for maximum
↪  liklihood estimator
for d in sorted(distribution_fitting, key=lambda k: k[sort_by])[:top]:
    distr_name = d['distr_name']
    mle = d[sort_by]
    pdf_fitted = d['pdf_fitted']
    params = d['params']
    print(params)

    label = "{} [{:.2f}]".format(distr_name, mle)
    _ = ax.plot(mydistr_x, pdf_fitted, label=label)


_ = ax.legend()
plt.xlabel('$|\\tilde{\\rho}|$')
plt.ylabel('$f(|\\tilde{\\rho}|)$')
fig.show()
```

List of Figures