Julius-Maximilians-Universität-Würzburg

Bachelor Thesis

A Study of the Statistic Nature of γ-ray Variability of Blazars

Author:
Luca Kohlhepp

Supervisor
Prof. Dr. Karl Mannheim

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Science

in the group of

Prof. Dr. Karl Mannheim
Chair for Astronomy

Contents

Zusammenfassung

Als Aktive Galaxienkerne (Active Galactic Nuclei - AGN) werden astronomische Objekte im Zentrum von Galaxien bezeichnet, deren Leuchtkraft durch die Akkretion von Materie in ein massereiches Schwarzes Loch hervorgerufen wird. Die Leuchtkraft der AGN kann die der umgebenden Galaxie um bis zu fünf Größenordnungen überschreiten. Sie verändert sich auf Zeitskalen von Minuten bis Jahrzehnten. Die stärkste Variabilität wird bei einer Untergruppe der AGN, den sogenannten Blazaren, beobachtet. Den Ursprung dieser Variabilität zu verstehen verspricht neue Erkenntnisse über die Energieumwandlungsprozesse in der Nähe Schwarzer Löcher.

Dafür ist es zunächst notwendig, einen Weg zu finden, die Variabilität in AGN mathematisch zu beschreiben. Physikalische Modelle der Strahlungsprozesse können dadurch eingeschränkt werden. Einfache Beschreibungen wie die periodische Veränderlichkeit können bereits ausgeschlossen werden. Darum konzentriert sich diese Arbeit auf statistische Prozesse und insbesondere auf den einfachsten Markow-Prozess mit Gedächtnis, der sogenannte Ornstein-Uhlenbeck (OU) Prozess. Der OU-Prozess stellt die Lösung einer stochastischen Differentialgleichung (OUDE) dar. Mit dem OU-Prozess können die unterschiedlichsten zeitlichen Variationen von Observablen von Modellsystemen beschrieben werden. Beispiele dafür sind die Brownsche Bewegung, der Ohmsche Widerstand eines elektrischen Leiters oder Börsenkurse.

Im Rahmen der Astrophysik wurde der OU-Prozess bereits erfolgreich eingesetzt, um die Variabilität der Strahlung von AGN Akkretionsscheiben im optischen und Röntgenbereich des elektromagnetischen Spektrums zu modellieren Takata et al. (2018, 2019); Kelly et al. (2011); Kelly et al. (2014). Da die Flußvariabilität im Bereich der Gammastrahlung am stärksten ausgeprägt ist, werden in dieser Arbeit AGN mit starker Gammastrahlung untersucht. Dies sind die sogenannten Blazare, in denen vom Schwarzen Loch ein Plasmajet ausgeht, der innerhalb weniger Grad auf den Beobachter ausgerichtet ist. Mit dem Fermi-LAT Observatorium wurden Zeitreihen der Flußdichte von Blazaren im Gammastrahlenbereich über mehrere Jahre aufgezeichnet, die als Datenbasis dienen.

Zunächst wurde die OUDE als Generator für Zahlenreihen implementiert. Es wurde eine Methode entwickelt, um die optimalen Parameter der OUDE bestimmen, die zur Erzeugung von Zahlenreihen mit den statistischen Eigenschaften der beobachteten Lichtkuren von Blazaren geeignet sind. Als zusätzlicher Test der Güte der Beschreibung wird die spektralen Leistungsdichte verglichen und gezeigt, dass im Rahmen der Fehler eine sehr gute Übereinstimmung erzielt werden kann.

Abstract

Active galactic nuclei (AGN) are astronomical objects in the center of galaxies. They are emitting light by the accretion of mass to a central supermassive black hole. The luminosity of AGN can exceed that of the host galaxy up to five magnitudes and varies on the scale of minuets to decades. The greatest variability can be observed from blazars, sub type of AGN. Understanding the origin of this variability will help to improve the knowledge of black hole physics, especially the energy transfers close it.

Therefore it is necessary to describe the variability in AGN mathematically. Trough this it is possible to restrict the physical models of the radiation processes. Simple descriptions like periodicity are already tested and excluded from this purpose. Therefore this work will focus upon statistical processes, especially the simplest Markov-Processes with memory, the so called Ornstein-Uhlenbeck (OU) process. The OU process is a solution of a stochastic differential equation (OUDE). The OU process can model a wide variety of time depended phenomena, such as Brownian motion, resistance in a wire and some financial models.

In Astrophysics the OU process is already successfully used to describe the variability of accretion discs in the optical and x-ray spectrum by Takata et al. (2018, 2019); Kelly et al. (2011); Kelly et al. (2014). In this thesis AGN with strong γ-ray emissions are explored, because the variability in the γ-ray spectrum is most pronounced. This are the so called blazars, in which a plasma jet extrudes from the black hole within a few degrees of the viewing angle. The Fermi/LAT satellite is recording time series of fluxes for more then 10 years, this is used as the data base.

For this the OUDE is implemented as a generator to produce time series. Then a method is developed, to find the optimal parameter of the OUDE, which can be used to reproduce time series that imitate the light curves (LCs) of the observed blazars. As an additional quality check the power spectrum density (PSD) is used to compare the artificial and original LCs. With that it is shown that there is a good alignment within error.

Chapter 1

Introduction

1.1 Active Galactic Nuclei

Active Galactic Nuclei (AGN) describe a galaxies with a supermassive black hole (SMBH) in its center, which is accreting mass. The estimated mass of the SMBH ranges from $6 \lesssim \log_{10}\left(\frac{M}{\mathrm{M}_\odot}\right) \lesssim 10$ (Vestergaard & Peterson 2006; Peterson et al. 2004). The accretion process gives rise to a powerful non-thermal radiation, originating from a disc formed by the accreting material as described by Shakura & Sunyaev (1973). Jets of plasma, which are found in some galaxies, can be another source of radiation, as they are leaving the galaxy perpendicular to the accretion disc and the galactic plane at highly relativistic velocities. These jets emit radio radiation which overpowers the galaxies in the optical spectrum (by a factor of 2-10). Thus, they are are called radio loud. The radiation, that is emitted from the AGN, has a high luminosity ranging form $42 \lesssim \log_{10}\left(\frac{L}{\mathrm{erg\,s^{-1}}}\right) \lesssim 47$ (Vestergaard & Peterson 2006; Peterson et al. 2004) and ranges over a wide band of frequencies. The spectral energy distribution features two humps, shown in fig. 1.2, dominated by different ways of emission. The low energy hump is likely caused by synchrotron radiation. The the high energy regime can be explained by either leptonic, hadronic or lepto-hadronic processes (Mannheim 1993).

1.1.1 Unification of AGN

There exists a multitude of different types of AGNs, that can be classified based on their morphology and their spectrum of the source. All of these manifestations can be summarized by one unified model, which has been introduced by Urry & Padovani (1995); Antonucci & Barvainis (1990); Antonucci (1993). According to the unification paradigm of AGNs, which today is considered the standard model of AGN, the appearance of the

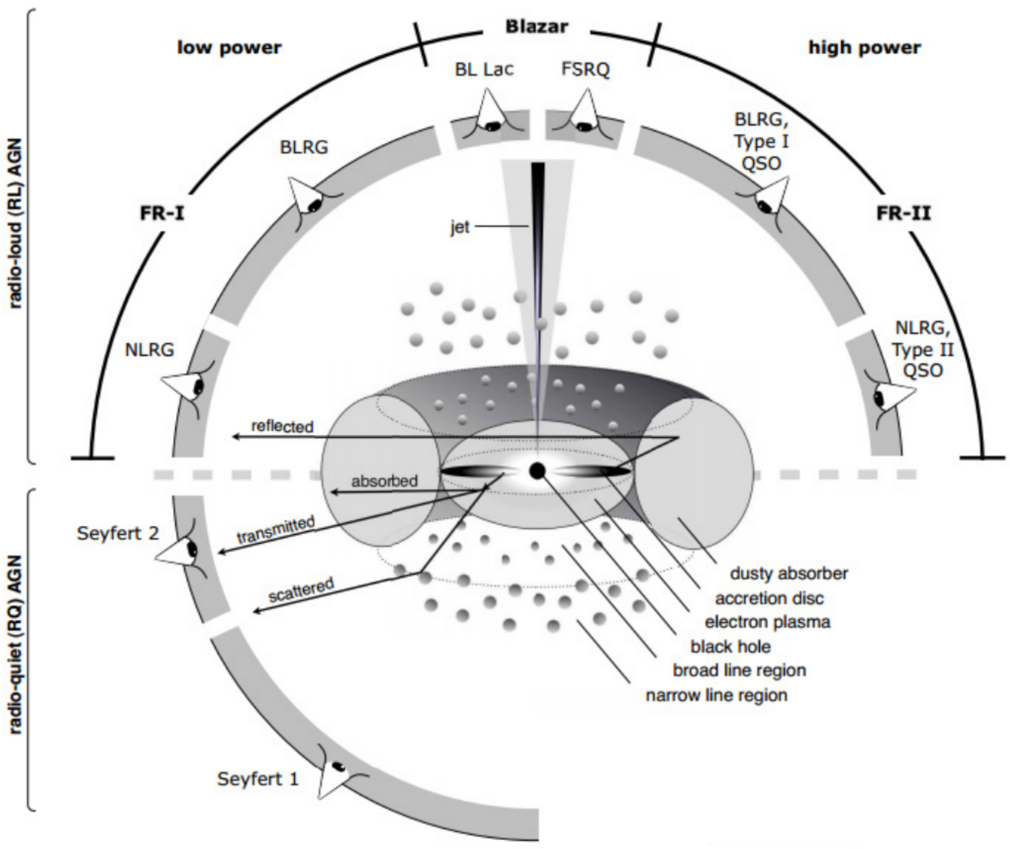

FIGURE 1.1: Visualisation of the unified model of AGN by Urry & Padovani (1995). Graphic by Beckmann & Shrader (2012)

different manifestations are depended on the viewing angle, the luminosity and the radio loudness, which itself is determined by the existence and extension of a jet. At the center of an active galaxy a SMBH is located, onto which mass is accreting, forming a radiating disc. The disc is surrounded by a torus of dust clouds blocking line of sight inside the galactic plane. Therefore obstructing the broad line region (disc) for high inclinations.

1.1.2 Blazars

Because of their large amplitude variability, blazars are a prime candidate to examine in this thesis. Blazars are a type of AGN that are described as a radio-loud AGN in the standard model, with a small angle between the jet and the line of sight. In other words the jet and its containing are moving towards the observer at relativistic speeds. Blazars count to the most luminous types of AGNs, this is explained by Doppler boosting and beaming of the relativistic jet. Blazars are commonly divided into two subgroups. The BL Lac objects, that are characterised by a optical featureless continuum emission and a variability on the time scale of minutes (Albert et al. 2007). The other class, has

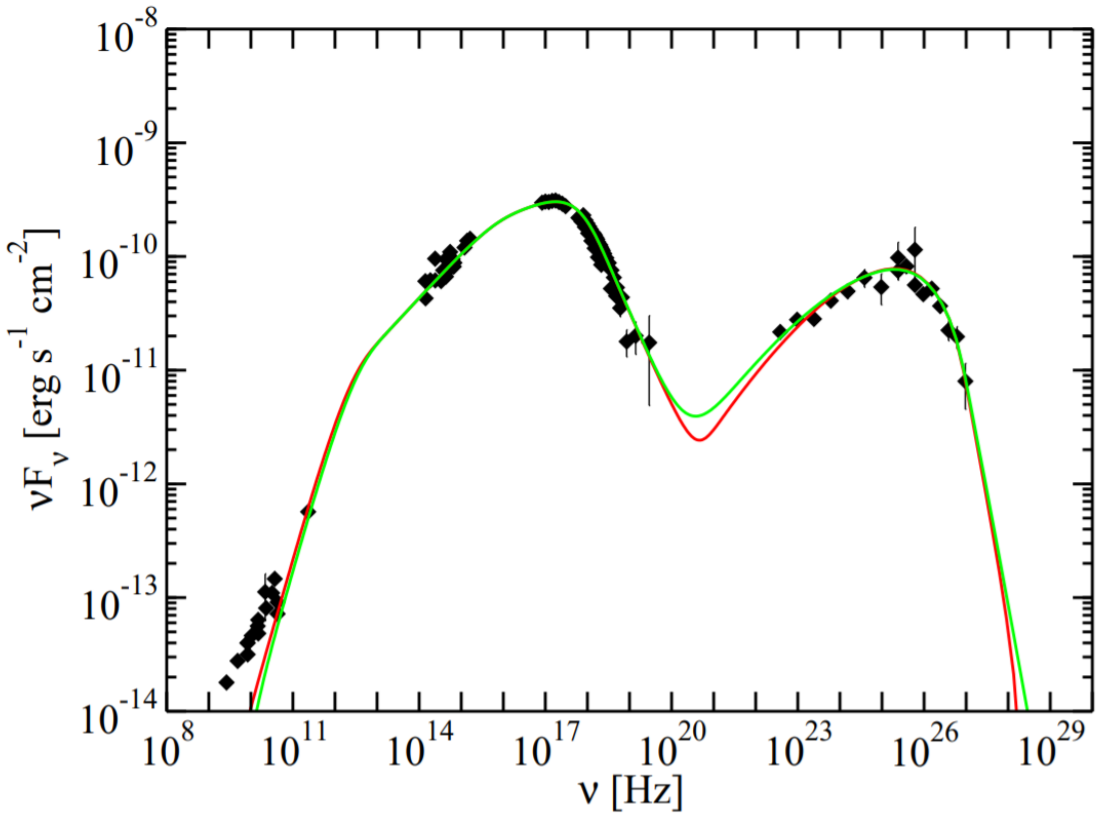

FIGURE 1.2: Characteristic spectrum of AGN shown by the example of Mrk 421 (Abdo et al. 2011). The green and red line show two fits of 1-zone synchrotron self-Compton models to Mrk 421.

a higher luminosity and an optical spectrum, which features emission lines as well as thermal emission. This class is named Flat-Spectrum Radio Quasar (FSRQ). FSRQs typically show a variability on longer time scales such as on weekly or monthly (Ulrich et al. 1997), but the are exceptions known such as CTA 102 (Shukla et al. 2018).

The word Blazar is a composite of the words 'blazing' and 'quasar'. The first emphasizes the luminous and beamed nature of the radiation, whereas the word "quasar", which short for quasi stellar, describes the point source, so star-like, nature of the source (Kreter 2018).

1.1.3 Spectra and Light Curves

The spectral energy distribution (SED) of blazars is characterised by two humps, as shown in fig. 1.2. These two humps can be explained by different radiation processes.

1.1.3.1 Synchrotron

The low-frequency hump is very well described by synchrotron radiation. According to Rybicki & Lightman (1986), synchrotron radiation is emitted, when relativistic charged particles are accelerated in a magnetic field $\vec{B}$ by Lorentz force. The Lorentz force in the relativistic case is given by:

$$\frac{d}{dt}(m\gamma\vec{v}) = \frac{q}{c}\left(\vec{v}\times\vec{B}\right). \tag{1.1}$$

m is the proper mass of the particle, γ is the Lorentz factor, $\vec{v}$ the velocity of the particle, $\vec{B}$ the magnetic field and q the charge of the particle. While the Lorentz factor is given by:

$$\gamma = \frac{1}{\sqrt{1-\beta^2}} \tag{1.2}$$

where $\beta = \frac{v}{c}$. Assuming a helical path of motion for the charged particle, an energy loss of that particle can be calculated by:

$$P = -\frac{2e^4B^2}{3m^2c^3}\beta^2\gamma^2\sin^2\alpha \tag{1.3}$$

with $B = |\,\vec{B}\,|$ and α being the pitch angle between $\vec{v}$ and $\vec{B}$. The charge of protons and electrons can be set as $q = \pm e$ respectively. For highly relativistic electrons that are present in the jet $\beta \approx 1$ is a good approximation. The radiated energy of a particle with elementary charge is proportional to m^{-2} for the same γ. Thus, synchrotron radiation emitted by protons is negligible in comparison to that emitted by electrons and positrons.

1.1.3.2 High Energy Emission

In comparison to the low-frequency hump it is not entirely clear which process causes the emission in the x-ray and γ-ray. Both leptonic and hadronic models are able to describe the data. Even though they are competing models, there are not mutually exclusive, which is shown by the existence of lepto-hadronic models (Mannheim 1993).

Leptonic Model - Inverse Compton Scattering

Compton Scattering is a well known effect of photons interacting with free electrons, in which the photons transfer energy to the electron. At highly relativistic electron speeds

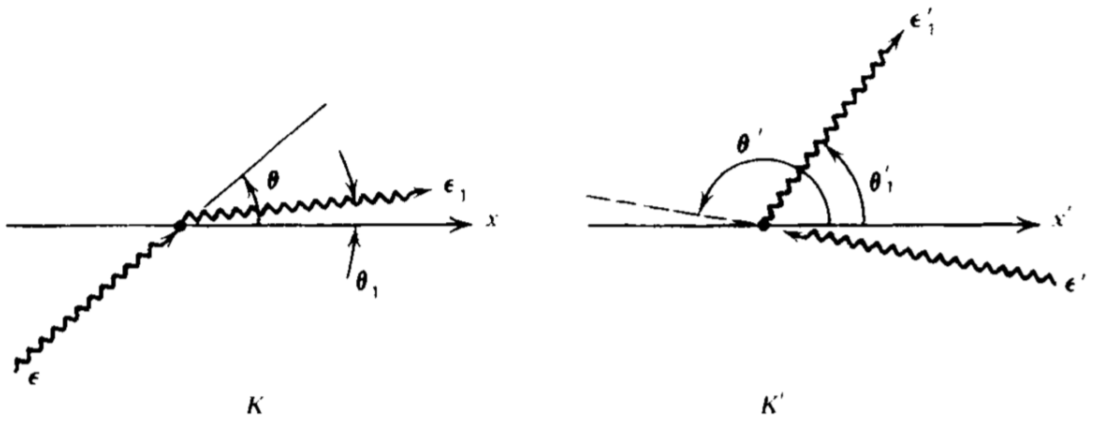

FIGURE 1.3: Scattering geometries in the observer's frame K and in the electron rest frame K' (Rybicki & Lightman 1986).

the opposite effect happens. In the inverse Compton scattering the highly relativistic electrons transfer energy to the photons they are scattering with.

By following the summary from Rybicki & Lightman (1986) and using the relativistic principle the inverse Compton scattering observed in the laboratory system (K) can be described as Thompson scattering from the rest frame of the electron (K'). Thompson scattering describes low energy photons ($h\nu \ll mc^2$) scattering elastically on electrons.

Therefore the photon energy in the rest frame of the electron needs to be calculated. For that Doppler shift equations are being used

$$\epsilon' = \epsilon\gamma(1 - \beta\cos\theta) \tag{1.4a}$$

$$\epsilon_1 = \epsilon'_1\gamma(1 + \beta\cos\theta'_1). \tag{1.4b}$$

Using the equation for energy transfer by Compton scattering in the rest frame of the electron, the posterior energy of the electron in the rest frame can be calculated by:

$$\epsilon'_1 = \frac{\epsilon'}{1 + \frac{\epsilon}{mc^2}(1 - \cos\Theta)} \tag{1.5a}$$

with Θ being

$$\cos\Theta = \cos\theta'_1\cos\theta' + \sin\theta'\sin\theta'_1\cos\phi' - \phi'_1 \tag{1.5b}$$

with ϕ' and ϕ'_1 being the azimuth angles of the incident photon angle and the scattered photon angle in the rest frame of the electron.

For relativistic electrons that satisfy the condition $\gamma^2 - 1 \gg \frac{h\nu}{mc^2}$, the photon energy before the scattering, in the rest frame of the electron after the scattering and after the scattering in the frame of the observer, are approximately in the ratios.

$$1 : \gamma : \gamma^2 \tag{1.6}$$

Thus, inverse Compton scattering can convert low energy photons into photons with γ^2-times the energy and can therefore explain high energy radiation.

Hadronic Model - Shock acceleration

The Hadronic Model requires protons in the jet that get accelerated at the shock front by the mechanism described by Fermi (1949). These highly relativistic protons can then interact with either the ambient matter or photons originating from the accretion disk or the dust torus (Sikora et al. 1987). This process is called *pion photoproduction* and can be described by the following reaction diagram (Kreter 2018)

$$p + \gamma \rightarrow \Delta^+ \rightarrow \begin{cases} \pi^0 + p \\ \pi^+ + n \end{cases} . \tag{1.7}$$

The next reaction diagrams show the further decay of the pions:

$$\begin{aligned} \pi^0 &\rightarrow \gamma + \gamma \\ \pi^+ &\rightarrow \mu^+ + \nu_\mu \\ \mu^+ &\rightarrow e^+ + \bar{\nu}_\mu + \nu_e . \end{aligned} \tag{1.8}$$

Assuming only decays of the leading order, the π^0-process is two times more likely than the π^+-decay.

The interaction with ambient matter can be described as a proton-proton interaction

$$p + p \rightarrow \begin{cases} p + p + \pi^0 \\ p + n + \pi^+ \end{cases} , \tag{1.9}$$

with a similar proportion of π^0 and π^+ generated the same way as with the pion photo production.

1.2 Nature of the Used Data

The data that used for this thesis is taken from the Fermi Large Area Telescope (LAT). The Fermi/LAT is a γ-ray space telescope launched in June 2008. It is a pair conversion telescope sensible in an energy range from 20 MeV to over 300 GeV. The telescope has a large field of view of 2.4 sr at 1 GeV and performs a scanning motion in its normal mode of operation to cover most of the sky with an uniform exposure in 2 orbits. This takes roughly 3 hours (Atwood et al. 2009).

For this thesis the flux values, for photons over 1 GeV of over 2000 extragalactic sources, binned at 28-day were used. The data was originally binned and employed for a coincidence analysis of a neutrino by IceCube Collaboration et al. (2018).

The data was prepocessed using the recommended process by the Fermi/LAT collaboration for point-sources and handed to the author in a binned format giving fluxes as well as additional data. No single photon events were provided.

The data used in this thesis covers the time from August 2008 to October 2017 and contains 460 FSRQs and 836 BL Lacs. The remaining sources also display blazar-like behaviour, but remain mostly unclassified.

1.3 Problem Addressed in this Work

The successful implementation of one, multiple or infinite OU processes to describe AGNs in the optical and x-ray spectrum by Kelly et al. (2011), Kelly et al. (2014) and Takata et al. (2018, 2019), has lead to the question if such an implementation of the OU process is also suitable to describe LCs in the γ-ray spectrum, which is known for its high variability. Also the γ-ray radiation does not originate in the disk, as the thermal dominated optical and x-ray emission does, but instead in the non-thermal jet and thus can be used to explore the mechanisms at work inside of the jet. The used data set from the Fermi/LAT presents itself as well suited for this, with a high number of sources spanning over a long observation time, without any major gaps in the acquisition process. This allows a analysis of the distribution of the whole population instead of examining single sources. To be able to deal with a data set of this size efficiently, a new method to extract the parameters of the OU process is developed.

Chapter 2

Methods

2.1 Mathematical Basis of the Ornstein-Uhlenbeck Process

The mathematical basis on which the light curve generator of this thesis is build, is the OUDE. It is a Stochastic-Differential-Equation (SDE) of first order, that was first proposed by Uhlenbeck & Ornstein (1930). Its basis is the Brownian motion, as described by Wiener. The difference is that the particle is not visualized in a vacuum anymore, but in a medium, where friction is introduced. To mimic the behavior of Fermi LCs at energies $> 1\,\mathrm{GeV}$ and monthly binning, the characteristics, describing the shape and signatures for characteristic time scales, regarding possible periodicity, have to be understood. Abdo et al. (2010) found indication for a correlated noise behavior in Fermi LCs. Timmer & Koenig (1995) showed that time sequences showing long term variation with respect to the sampling time scale (time binning) feature slopes in the power spectral density (PSD $\propto f^{\beta}$) of $\beta \sim -2$ (red noise). Time series showing short time variability with respect to the sampling time show pink noise behavior ($\beta \sim -1$) instead. A process, capable of realizing this is the OU-process, see Uhlenbeck & Ornstein (1930), described by the OUDE, a stochastic differential equation (SDE). The basis for the OU process is the Brownian motion of particles exposed to friction by an ambient medium. In the following a motivation for the OUDE will be discussed based on the Wiener process. This requires a time dependent, random noise function $\Gamma(t)$ for which the necessary properties will be described. As the OUDE stands, solutions can be given in a discrete form and the stationarity of the OU process around a given mean value can be proven.

2.1.1 Motivation of the SDE

The basic SDE for the Wiener process describing particles in vacuum undergoing Brownian motion reads:

$$\frac{du_W(t)}{dt} = \Gamma(t). \tag{2.1}$$

Where $u(t)$ denotes the velocity of the particle and $\Gamma(t)$ a white noise. The properties of the white noise $\Gamma(t)$ are detailed in Sec. 2.1.2., Uhlenbeck & Ornstein (1930) then introduced a friction, which is directly proportional to the velocity:

$$\frac{du_{OU}(t)}{dt} = -\theta u(t) + \Gamma(t). \tag{2.2}$$

Following Uhlenbeck & Ornstein (1930), the strength of the friction is described by a friction coefficient θ. A friction term, being directly proportional to the velocity covers a wide range of specific type of frictions, such as Stokes and Doppler friction (Uhlenbeck & Ornstein 1930). While in this case θ is a scalar coefficient, this can in principle be extended to more dimensions by introducing θ as a friction tensor instead.

Accounting for a velocity of the surrounding medium, μ, Eq.2.2 can be re-written as

$$\frac{du(t)}{dt} = \theta(\mu - u(t)) + \Gamma(t). \tag{2.3}$$

2.1.2 Properties of $\Gamma(t)$

Before solving the SDE, the properties of the in-homogeneity in Eq.2.3, $\Gamma(t)$, will be discussed. $\Gamma(t)$ is defined to be white noise. Samples obtained from white noise are completely uncorrelated and distributed in a way that their mean is 0:

$$\langle \Gamma(t) \rangle = 0 \tag{2.4a}$$

$$\langle \Gamma(t)\Gamma(t+\tau) \rangle = \delta(\tau) \tag{2.4b}$$

(Gillespie 1996b).

These requirements are satisfied by the normal distribution with a mean $\mu = 0$ and a variance σ, also written as

$$N(0, \sigma^2). \tag{2.5}$$

Gillespie (1996b) summarizes further properties of the normal distribution, which are listed in the following and used in the subsequent sections:

$$\alpha + \beta N(m, \sigma^2) = N(\alpha + \beta m, \beta^2\sigma^2) \tag{2.6a}$$

$$\alpha + \beta N(0, 1) = N(\alpha, \beta^2) \tag{2.6b}$$

$$N(m_1, \sigma_1^2) + N(m_2, \sigma_2^2) = N(m_1 + m_2, \sigma_1^2 + \sigma_2^2). \tag{2.6c}$$

To solve the SDE the time infinitesimal of the normal distribution should be studied first.

$$N(0, \sigma^2)dt \tag{2.7}$$

Let $dt' = \frac{dt}{n}$, where $n > 0 \in \mathbb{N}$ and thus $dt' < dt$, see Gillespie (1996b), then Eq.2.7 can be re-written as the sum over n steps. Gillespie (1996b) gives following proof with an arbitrary random distribution satisfying Eqs.2.6a-2.6c whereas here the focus is laid on a Gaussian distribution for $\Gamma(t)$:[1]

$$N(0, \sigma_1^2)\ dt = \sum_{i=0}^{n} N(0, \sigma_2^2)\ dt'. \tag{2.8a}$$

Using 2.6c, 2.8a reads

$$N(0, \sigma_1^2)\ dt = dt'\ N\left(0, \sum_{i=0}^{n} \sigma_2^2\right) \tag{2.8b}$$

$$N(0, \sigma_1^2)\ dt = \frac{dt}{n} N(0, n\sigma_2^2) \tag{2.8c}$$

and using 2.6a, 2.8c becomes

$$N(0, \sigma_1^2)\ dt = dt\ N\left(0, \frac{\sigma_2^2}{n}\right) \tag{2.8d}$$

$$\implies \sigma_1^2 = \frac{\sigma_2^2}{n}\ \forall n. \tag{2.8e}$$

For Eq.2.8e to be true, σ_1 and σ_2 need to be dependent on dt and dt' respectively.

Ansatz:

$$\sigma_1^2 = \frac{\sigma'^2}{dt}; \sigma_2^2 = \frac{\sigma'^2}{dt'} \tag{2.9a}$$

$$\frac{\sigma'^2}{dt} = \frac{\sigma'^2}{n\ dt'} \tag{2.9b}$$

$$\frac{\sigma'^2}{dt} = \frac{n\sigma'^2}{n\ dt} \tag{2.9c}$$

[1] Also a uniform distribution would suffice Eqs.2.6a-2.6c

Hence, the distribution of $\Gamma(t)$ is defined in such a way that its infinitesimal does not change when the size of dt is varied:

$$\Gamma(t) = N\left(0, \frac{\sigma^2}{dt}, t\right) \tag{2.10a}$$

with $N(t) = N(0,1,t)$, follows:

$$\Gamma(t) = \frac{\sigma}{\sqrt{dt}} N(t) \tag{2.10b}$$

$$\Gamma(t)dt = \sigma N(t)\sqrt{dt}. \tag{2.10c}$$

2.1.3 Solutions of the SDE

The time derivative of the velocity $\frac{du}{dt}$ can be expressed as

$$\frac{du}{dt} = \frac{1}{dt}\left(u(t+dt) - u(t)\right). \tag{2.11}$$

Applying this to Eq.2.3 and solving for $u(t+dt)$, yields

$$u(t+dt) = u(t) + \theta(\mu - u(t))dt + \Gamma(t)dt. \tag{2.12a}$$

Utilizing a normal distribution in the manner of Eq.2.10c for the noise term $\Gamma(t)$ results in following SDE:

$$u(t+dt) = u(t) + \theta(\mu - u(t))dt + \sigma N(t)\sqrt{dt}. \tag{2.12b}$$

To be calculated by a computer, a discrete formulation of the process is needed, that equals or approximates a sampled continuous process, so that

$$u_T = u(t = \Delta t\, T)\ \forall T \in \mathbb{N}. \tag{2.13}$$

If the sampling of time steps in the computation is smaller than the relevant time scales of the considered process, the continuous SDE can be approximated by a discrete function ($dt \to \Delta t \ll 1$), which is proven in appendix C:

$$u_{T+1} = u_T + \theta\,\Delta t(\mu - u_T) + \sigma\sqrt{\Delta t}N_T. \tag{2.14}$$

An exact updating formula, where $\Delta t \ll 1$ is not necessary to hold, can be found in the paper of Gillespie (1996a) and in the appendix C. Gillespie (1996a) introduces exponential decay terms, ensuring u_T to return back to μ in any given step.

2.1.4 Stationarity of the OU-Process

For a process to be stationary in a wide sense, it has to fulfill the following criteria:

$$\langle u(t)\rangle = \langle u(t+\tau)\rangle \tag{2.15a}$$

$$\langle u^2(t)\rangle < \infty \; \forall t. \tag{2.15b}$$

$$\langle u(t)u(t+\tau)\rangle - \langle u(t)\rangle\langle u(t+\tau)\rangle = \langle u(\tau)u(0)\rangle - \langle u(\tau)\rangle\langle u(0)\rangle \;\; \forall t \; \forall \tau > 0 \tag{2.15c}$$

Equation 2.15a states that the mean value is a constant function and thus independent on the time step. Equation 2.15b gives a finite variance for all time steps and Eq.2.15c states that the co-variance is only dependent on the difference between two time steps.

2.1.4.1 Rearrangement (to simplify the proofs)

u_T is written as $u_{s\cdot\Delta t} = x(s), s \in \mathbb{N}$, so that the following calculations can be written on the dependence on steps. At first the solution of the OU-Process is simplified for statistical analysis:

$$x(s) = x(s-1) + \theta\Delta t(\mu - x(s-1)) + \sigma\sqrt{\Delta t}N(0, 1, (s\cdot\Delta t) - 1). \tag{2.16a}$$

Then a closed expression of $x(s)$ is needed that only depends on $x(0)$ and the samples of the normal distribution:

$$\Rightarrow x(s) = (1-\theta\Delta t)^s x(0) + \sum_{k=1}^{s} [(1-\theta\Delta t)^{s-k} \cdot (\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0, 1, (k-1)\cdot\Delta t))]. \tag{2.16b}$$

That this equation satisfies the OUDE can be shown by induction. By definition the initial case is fulfilled with

$$x(0) = x(0). \tag{2.16c}$$

Thus the induction can be started:

$$\begin{aligned} x(s+1) &= (1-\theta\Delta t)x(s) + \theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,\Delta t s) \\ &= (1-\theta\Delta t)\left[(1-\theta\Delta t)^s x(0) + \sum_{k=1}^{s}\Big((1-\theta\Delta t)^{s-k}(\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big)\right] \\ &\quad + \theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,\Delta t s) \end{aligned}$$

$$= (1-\theta\Delta t)^{s+1}x(0) + (1-\theta\Delta t)\sum_{k=1}^{s}\Big((1-\theta\Delta t)^{s-k}(\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big)$$
$$+\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,\Delta t s)$$
$$= (1-\theta\Delta t)^{s+1}x(0) + \sum_{k=1}^{s}\Big((1-\theta\Delta t)^{s+1-k}(\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big)$$
$$+\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,\Delta t s)$$
$$= (1-\theta\Delta t)^{s+1}x(0) + \sum_{k=1}^{s+1}\Big((1-\theta\Delta t)^{s+1-k}(\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big)\,\square \tag{2.16d}$$

At this point n is introduced as $n = s - k$. Therefore $k = s - n$. The start index of the sum reads

$$k = 1 = s - n \Rightarrow n = s - 1. \tag{2.16e}$$

The end index becomes

$$k = s = s - n \Rightarrow n = 0. \tag{2.16f}$$

Because of the sum is associative, the start and end points of the sum can be interchanged:

$$\Rightarrow x(s) = (1-\theta\Delta t)^{s}x(0) + \sum_{n=0}^{s-1}[(1-\theta\Delta t)^{n}\cdot(\theta\Delta t\mu + \sigma\sqrt{\Delta t}N(0,1,((s-n)\cdot\Delta t)-1))]. \tag{2.16g}$$

In the next step equation 2.6c is applied. All normal distributed independent variables from 0 to $((s-n)\cdot\Delta t)-1$ are summed. For each new time step a new uncorrelated variable is added. Thus, the sum equals a normal distribution at a specific time step, e.g. the time the last step was added. Here for convenience of short equations the time step that will be added next is chosen instead:

$$\Rightarrow x(s) = (1-\theta\Delta t)^{s}x(0) + N\left(\theta\Delta t\mu\sum_{n=0}^{s-1}(1-\theta\Delta t)^{n}, \sigma^2\Delta t\sum_{n=0}^{s-1}(1-\theta\Delta t)^{2n}, s\cdot\Delta t\right). \tag{2.16h}$$

If one writes this as a single normal distributed variable using 2.6a, it becomes

$$\Rightarrow x(s) = N\left((1-\theta\Delta t)^{s}x(0) + \theta\Delta t\mu\sum_{n=0}^{s-1}(1-\theta\Delta t)^{n}, \sigma^2\Delta t\sum_{n=0}^{s-1}(1-\theta\Delta t)^{2n}, s\cdot\Delta t\right)$$
$$\hat{=} N(\text{Mean}(x), \text{Var}(x), t). \tag{2.16i}$$

2.1.4.2 Proof of the 1st Criterion

To proof the first criterion 2.15a, equation 2.17a is shown to be true for all s:

$$\langle x(s)\rangle \overset{!}{=} \langle x(s+1)\rangle \ \forall s \in \mathbb{N}. \tag{2.17a}$$

Hence $x(s)$ can be written as normal distributions 2.16i. The mean is the first argument of N. Therefore 2.17a simplifies to:

$$(1-\theta\Delta t)^s x(0) + \theta\Delta t\mu \sum_{n=0}^{s-1}(1-\theta\Delta t)^n = (1-\theta\Delta t)^{s+1}x(0) + \theta\Delta t\mu\sum_{n=0}^{s}(1-\theta\Delta t)^n \tag{2.17b}$$

$$0 = [(1-\theta\Delta t)^{s+1} - (1-\theta\Delta t)^s]x(0) + \theta\Delta t\mu(1-\theta\Delta t)^s \tag{2.17c}$$

$$\underbrace{(1-\theta\Delta t)^{s+1}x(0)}_{\text{LHS}} = \underbrace{(x(0)-\theta\Delta t\mu)(1-\theta\Delta t)^s}_{\text{RHS}}. \tag{2.17d}$$

From there can see, that this equation is satisfied for all s if

$$x(0) = \mu \Rightarrow \text{RHS} = (\mu - \theta\Delta t\mu)(1-\theta\Delta t)^s = \mu(1-\theta\Delta t)^{s+1} = \text{LHS}|_{x(0)=\mu}\square. \tag{2.17e}$$

If a system runs sufficiently long, the criterion of 2.17e loses its importance, because after a sufficient amount of time a steady-state is reached, making the initial conditions unimportant.

This implies that

$$\mu = x(0) = \langle x(0)\rangle = \langle x(s)\rangle. \tag{2.17f}$$

2.1.4.3 Proof of the 2nd Criterion

For the proof of 2.15b it is again used that $x(s)$ is normal distributed and that $\langle d^2\rangle = Var(d)$ for some distribution d:

$$Var(x(s)) = \sigma^2\Delta t\sum_{n=0}^{s-1}(1-\theta\Delta t)^{2n}. \tag{2.18a}$$

This, expression is required to be finite for all times. Hence a lower limit of $s = 0$ is given and only the limit of $s \to \infty$ is analyzed:

$$Var(x(s)) = \lim_{s\to\infty}\sigma^2\Delta t\sum_{n=0}^{s-1}(1-\theta\Delta t)^{2n} = \lim_{s\to\infty}\sigma^2\Delta t\sum_{n=0}^{s-1}[(1-\theta\Delta t)^2]^n \tag{2.18b}$$

$$Var(x(s)) \overset{\text{geo. series}}{=} \begin{cases} \frac{\sigma^2 \Delta t}{1-(1-\theta\Delta t)^2}, & (1-\theta\Delta t)^2 < 1 \\ \infty, & (1-\theta\Delta t)^2 \geq 1 \end{cases} \tag{2.18c}$$

$$Var(x(s)) = \begin{cases} \frac{\sigma^2 \Delta t}{2\theta\Delta t-\theta^2\Delta t^2}, & |1-\theta\Delta t| < 1 \\ \infty, & |1-\theta\Delta t| \geq 1 \end{cases}. \tag{2.18d}$$

Thus this requirement for a stationary process is fulfilled for $|1-\theta\Delta t| < 1$.

2.1.4.4 Proof of the 3rd Criterion

For the following proof it is necessary to express $x(t+\tau)$, dependent on $x(t)$ and independent of $x(0)$:

$$x(s+\tau) = (1-\theta\Delta t)^{s+\tau}x(0) + \sum_{k=1}^{s+\tau}\Big[(1-\theta\Delta t)^{s+\tau-k}\cdot(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t))\Big]. \tag{2.19a}$$

Separating the one sum into two, so the first goes from index 1 to s and the second from $s+1$ to $s+\tau$. It becomes

$$\begin{aligned} x(s+\tau) &= (1-\theta\Delta t)^s(1-\theta\Delta t)^\tau x(0)+ \\ &\sum_{k=1}^{s}\Big[(1-\theta\Delta t)^{s-k}(1-\theta\Delta t)^\tau\cdot(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big]+ \\ &\sum_{k=s+1}^{s+\tau}\Big[(1-\theta\Delta t)^{s+\tau-k}\cdot(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t))\Big] \\ &= (1-\theta\Delta t)^\tau \underbrace{\left((1-\theta\Delta t)^s x(0) + \sum_{k=1}^{s}\Big[(1-\theta\Delta t)^{s-k}(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t)\Big]\right)}_{x(s)} \\ &+ \sum_{k=s+1}^{s+\tau}\Big[(1-\theta\Delta t)^{s+\tau-k}\cdot(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k-1)\Delta t))\Big] \\ &= (1-\theta\Delta t)^\tau\cdot x(s) + \sum_{k=1}^{\tau}\Big[(1-\theta\Delta t)^{\tau-k}\cdot(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t))\Big]. \end{aligned} \tag{2.19b}$$

The result is an explicit equation for $x(s+\tau)$ based on $x(s)$. Thus, it can be begun with proofing the criterion of equation 2.15c:

$$\langle x(s)x(s+\tau)\rangle =$$

$$\left\langle x(s)\left((1-\theta\Delta t)^{\tau}x(s)+\sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t))\right]\right)\right\rangle. \tag{2.20a}$$

Multiplying $x(s)$ in the sum and using the linearity of the expectation value, one yields:

$$\begin{aligned}\langle x(s)x(s+\tau)\rangle &= \left\langle (1-\theta\Delta t)^{\tau}x^2(s)\right\rangle+\left\langle\sum_{k=1}^{\tau}\left[x(s)(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu x(s)\right]\right\rangle+\\ &\left\langle\sum_{k=1}^{\tau}(1-\theta\Delta t)^{\tau-k}\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t)\right\rangle.\end{aligned} \tag{2.20b}$$

The linearity of the expectation value is used again to pull it into the sum and all scalars out of it.

$$\begin{aligned}\langle x(s)x(s+\tau)\rangle &= (1-\theta\Delta t)^{\tau}\left\langle x^2(s)\right\rangle+\sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu\left\langle x(s)\right\rangle\right]+\\ &\left\langle\sum_{k=1}^{\tau}x(s)(1-\theta\Delta t)^{\tau-k}\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t)\right\rangle\end{aligned} \tag{2.20c}$$

At this point it is necessary to simplify the last summand.

AUX:

Replacing $x(s)$ with its explicit form yields:

$$\begin{aligned}&\left\langle\sum_{k=1}^{\tau}x(s)(1-\theta\Delta t)^{\tau-k}\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t)\right\rangle\\ &=\left\langle\sum_{k=1}^{\tau}\left[\sum_{i=1}^{s}\left[(1-\theta\Delta t)^{s}(\theta\Delta t\mu+\sigma\sqrt{\Delta t}N(0,1,(i-1)\Delta t))\right]\cdot\right.\right.\\ &\left.\left.(1-\theta\Delta t)^{\tau-k}\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t)\right]\right\rangle=S.\end{aligned} \tag{2.20da}$$

The product inside the sum is expended inside the inner sum:

$$\begin{aligned}S=&\left\langle\sum_{k=1}^{\tau}\sum_{i=1}^{s}\left[(1-\theta\Delta t)^{s+\tau-k}\theta\Delta t\sigma\sqrt{\Delta t}N(0,1,(k+s-1)+\right.\right.\\ &\left.\left.(1-\theta\Delta t)^{s+\tau-k}\sigma^2\Delta tN(0,1,(i-1)\Delta t)N(0,1,(k+s-1)\Delta t\right]\right\rangle.\end{aligned} \tag{2.20db}$$

Using linearity of the expectation values, the expectation value of each of the inner summands is calculated separately:

$$S = \left\langle \sum_{k=1}^{\tau}\sum_{i=1}^{s}\left[(1-\theta\Delta t)^{s+\tau-k}\theta\Delta t\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\right]\right\rangle + \left\langle \sum_{k=1}^{\tau}\sum_{i=1}^{s}[(1-\theta\Delta t)^{s+\tau-k}\sigma^2\Delta t N(0,1,(i-1)\Delta t)N(0,1,(k+s-1)\Delta t]\right\rangle. \tag{2.20dc}$$

Linearity of the expectation value is applied again to pull it inside the sum

$$S = \sum_{k=1}^{\tau}\sum_{i=1}^{s}[(1-\theta\Delta t)^{s+\tau-k}\theta\Delta t\sigma\sqrt{\Delta t}\langle N(0,1,(k+s-1)\rangle]+ \sum_{k=1}^{\tau}\sum_{i=1}^{s}[(1-\theta\Delta t)^{s+\tau-k}\sigma^2\Delta t\langle N(0,1,(i-1)\Delta t)N(0,1,(k+s-1)\Delta t\rangle]. \tag{2.20dd}$$

By definition the expectation value of the normal distribution is defined as $\langle N(0,1,t)\rangle = \langle\Gamma(t)\rangle = 0$ (Eq. 2.4a) and the auto-correlation as $\langle N(0,1,t)N(0,1,t+\tau) = \langle\Gamma(t)\Gamma(t+\tau)\rangle = \delta(\tau)$ (Eq. 2.4b). Thus, the first summand vanishes and the equation reads:

$$S = \sum_{k=1}^{\tau}\sum_{i=1}^{s}[(1-\theta\Delta t)^{s+\tau-k}\sigma^2\Delta t\delta((k+s-1)\Delta t-(i-1)\Delta t)]. \tag{2.20de}$$

$(k+s-1)-(i-1) = k+s-i$ with $k>1$ and $i<s$. Thus $k+s-i>0\ \forall k,i$. $\Rightarrow \delta((k+s-i)\Delta t) = 0\ \forall k,i$.

$$\Rightarrow \left\langle \sum_{k=1}^{\tau} x(s)(1-\theta\Delta t)^{\tau-k}\sigma\sqrt{\Delta t}N(0,1,(k+s-1)\Delta t)\right\rangle = 0 = S. \tag{2.20df}$$

With that knowledge the main calculation can be continued:

$$\langle x(s)x(s+\tau)\rangle = (1-\theta\Delta t)^{\tau}\left\langle x^2(s)\right\rangle + \sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu\left\langle x(s)\right\rangle\right]. \tag{2.20e}$$

Using equation 2.17f it follows that

$$\langle x(s)x(s+\tau)\rangle = (1-\theta\Delta t)^{\tau}\left\langle x^2(s)\right\rangle + \sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu^2\right]. \tag{2.20f}$$

Considering the relative autocovariance between two time steps, the following statement needs to hold:

$$\langle x(t_1)x(t_1+\tau\rangle \overset{!}{=} \langle x(t_2)x(t_2+\tau\rangle. \tag{2.20g}$$

$$\begin{aligned}\Leftrightarrow (1-\theta\Delta t)^{\tau}\langle x^2(t_1)+\rangle \sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu^2\right] &= \\ (1-\theta\Delta t)^{\tau}\langle x^2(t_2)+\rangle \sum_{k=1}^{\tau}\left[(1-\theta\Delta t)^{\tau-k}\theta\Delta t\mu^2\right]\end{aligned} \tag{2.20h}$$

$$\Rightarrow \langle x^2(t_1)\rangle = \langle x^2(t_2)\rangle \tag{2.20i}$$

If t_1 and t_2 are sufficiently large and the difference $t_1 - t_2$ is constant, Eq.2.20i reads

$$\Rightarrow \lim_{t_1\to\infty}\langle x^2(t_1)\rangle = \lim_{t_2\to\infty}\langle x^2(t_2)\rangle \tag{2.20j}$$

Using equation 2.18a and 2.18c it becomes.

$$\Rightarrow \frac{\sigma^2\Delta t}{1-(1-\theta\Delta t)^2} = \frac{\sigma^2\Delta t}{1-(1-\theta\Delta t)^2}\ \square \tag{2.20k}$$

2.1.5 Extraction of the Parameters

To describe light curves properly with the OU-Process, it is necessary to be able to determine the parameters of the OUDE from a given time series. This means a unique projection form the physical parameters to the OU parameters, (physical parameters) $\rightarrow$ (μ, θ, σ), needs to be found. This allows to choose OU parameters based on physical measurements to create artificial light curves that mimic the natural ones as close as possible. For that reason, methods to determine μ, σ and θ from the time series of an OU process are developed. In the following, a mathematical description of the methods is given as well as its derivation. The method for determining μ and σ is described first. From those the θ parameter can be derived. The parameter μ is extracted straight forwardly from the time series by calculating the expectation value (mean).

2.1.5.1 Extraction of σ

To extract σ, data points where u_T is close to the mean (within an ϵ environment) are considered. Following Ansatz is used:

$$u_T = \mu + \epsilon. \tag{2.5}$$

Then the SDE 2.14 reads:

$$u_{T+1} = u_T + \theta\Delta t(\mu - (\mu + \epsilon)) + \sigma\sqrt{\Delta t}N(t) \tag{2.6}$$

$$\Rightarrow u_{T+1} - u_T = -\theta\Delta t\epsilon + \sigma\sqrt{\Delta t}N(t). \tag{2.7}$$

For an $\epsilon \ll \frac{\sigma\sqrt{\Delta t}}{\theta\Delta t}$ the term with ϵ is negligible. It is therefore reduced to

$$u_{T+1} - u_T = \sigma\sqrt{\Delta t}N(t) = N(0, \sigma^2\Delta t, t). \tag{2.8}$$

If a given set of u_T is within the ϵ environment encompassing μ, the variance can be calculated on both sides. For normal distributions its variance is its second argument, like it is shown in equation 2.7, thus

$$Var(u_{T+1} - u_T) = Var(N(0, \sigma^2\Delta t, t)) = \sigma^2\Delta t. \tag{2.9}$$

Therefore $\sigma\sqrt{\Delta t}$ can be written in a closed expression as:

$$\sigma\sqrt{\Delta t} = \sqrt{Var\left(\left\{u_{T+1} - u_T \middle| u_T - \mu < \epsilon << \frac{\sigma\sqrt{\Delta t}}{\theta\Delta t}\right\}\right)} \tag{2.10}$$

2.1.5.2 Extraction of θ

With the values extracted for $\sigma\sqrt{\Delta t}$, $\theta\Delta t$ can be determined by utilizing the variance of the complete time series. From the equations 2.18a and 2.18c follows that

$$Var(u(t)) = \begin{cases} \frac{\sigma^2\Delta t}{1-(1-\theta\Delta t)^2}, & (1-\theta\Delta t)^2 < 1 \\ \infty, & (1-\theta\Delta t)^2 >= 1 \end{cases}. \tag{2.11}$$

Hereafter stationary processes will be focused upon, so that

$$Var(u(t)) = \frac{\sigma^2\Delta t}{1-(1-\theta\Delta t)^2} \tag{2.12}$$

$$1 - \theta\Delta t = \pm\sqrt{1 - \frac{\sigma^2\Delta t}{Var(u_T)}} = \alpha. \tag{2.13}$$

Because the variance is insensitive for the sign of the deviation from the mean, the sign of α can not be determined by this method. Only the absolute $|\alpha|$ is known as

$$|\alpha| = \sqrt{1 - \frac{\sigma^2\Delta t}{Var(u_T)}}. \tag{2.14}$$

Hence, another method is used to evaluate the sign. This other method is less precise than the first method since it depends on data points sufficiently far from μ which in most cases of time series are more scarce than the bulk of data points scattering close to μ. However, the following method however is sensitive to the sign of α and can be used in combination with the method shown above to pinpoint the value and sign of

α. Therefore equation 2.14 is considered for large deviations of u_T from μ and a small σ. Such that the term with the normal distribution is small against the first two terms with the reversion rate θ and vanishes:

$$u_{T+1} = u_T + \theta\Delta t(\mu - u_T). \tag{2.15a}$$

This is now rewritten in terms of α:

$$\Rightarrow u_{T+1} = (1 - \theta\Delta t)u_T + \theta\Delta t\mu = \alpha u_T + (1 - \alpha)\mu \tag{2.15b}$$

$$\Rightarrow u_{T+1} = \alpha u_T + \mu - \alpha\mu \tag{2.15c}$$

$$\Rightarrow \frac{u_{T+1} - \mu}{u_T - \mu} = \alpha. \tag{2.15d}$$

The α is averaged for all sufficiently large u_T. The sign sensitive α can be calculated by

$$\alpha_\pm = \left\langle \left\{ \frac{u_{T+1} - \mu}{u_T - \mu} \middle| N(t) \ll \frac{\alpha u_T + \theta\Delta t\mu}{\sigma\sqrt{\Delta t}}. \right\} \right\rangle \tag{2.15e}$$

From these two methods the absolute and sign of α can be calculated separately. The absolute value of the more precise method (Eq. 2.14) is used and the sign of the sign sensitve method (Eq. 2.15e):

$$\alpha = sign(\alpha_\pm) \cdot |\alpha|. \tag{2.16}$$

$\theta\Delta t$ can now be calculated from this α, using the definition of α in equation 2.13:

$$\theta\Delta t = 1 - \alpha. \tag{2.17}$$

2.2 Implementation of the Algorithms

In the following section the implementation of the Ornstein-Uhlenbeck-Simulator as well as the parameter extractor are explained. Both are designed as python 3 (version 3.7.3) modules to be imported into other python programs or juypter notebooks. In both modules numpy (version 1.17.2) was used, because its arrays handle large amount of data more efficiently than default python lists.

2.2.1 Ornstein-Uhlenbeck-Simulator

The generation of random numbers, like the white noise needed for the UO process, requires significant computing power. Generating multiple random numbers at once can reduce the computing power needed for each random number. To make use of this feature and to reduce computational overhead, this implementation was designed to calculate large numbers of Monte-Carlo-Simulations (MCS) at once. To have this functionality and still be highly customizable by the user some of the parameters of the main function can accept a variety of types as arguments.

For the implementation in python the numpy module is utilized. Numpy is used for two reasons. On the one hand, python lists get highly inefficient when multiple (n) lists are stacked inside of each other, the access time complexity is $\mathcal{O}(n)$, compared to a multidimensional numpy array where it is $\mathcal{O}(1)$. Additionally, for all cases where the type of the numpy array is not `object`, the memory used by the pyhton list is bigger. While having the same space complexity ($\mathcal{O}(n)$), each element requires an additional pointer. Also, python lists stacked into each other cannot easily be generalized for an arbitrary number of lists stacked into each other. While the depths of stacking represents the number of axes[2]. On the other hand, numpy has high quality random number generators that can generate numbers in high quantities directly into numpy arrays.

The main function of the module is named `ou_generate` and it is the only function of the module that needs to be called by the user for basic functionality and most advanced functionality. The mandatory parameters of that function are `iterations`, `theta`, `sigma`, `mu` and `x0`.

- `iterations` is the number of time steps calculated per MCS. Its type is `int`.
- `theta`, `sigma` and `mu` are the parameters of the equation 2.14. Their type can differ and either be an `object` of the type `ndarray` or with the attribute `__iter__`, so it can be converted into a `ndarray`. If it is either, after conversion, its `shape` attribute needs to be equal to the optional parameter `size`. They can also be a callable that can generate a `ndarray` with the shape `size` if given a size argument. Or they can be a `float`.
- `x0` are the start values. Its type can be `None` in addition to all the types `theta`, `sigma` and `mu` can be. If it is `None`, values from a white noise distribution are used to set the start values.

In addition to these mandatory parameters there are optional parameters, commonly called ko-arguments, which always have a default value. The ko-arguments of the main function are `dt`, `noise_generator`, `noise_parameters`, `size` and `unpack`.

[2] the length of one axis is its dimension

- `dt` sets the Δt value of the equation 2.14. It does not change the amount of time steps, but the length of those steps. That means the total duration of the MCS is given by `iterations` $\cdot$ `dt`. Default value is `1`.
- `noise_generator` is the function used for generation of the white noise. It needs to take an argument named `size` and return a `ndarray`, with its `shape` equal `size`. It can also take more arguments, then these must be specified in `noise_parameters`. Default value is `numpy.random.standard_normal`.
- `noise_parameters` will pass through arguments to the noise generator. Its type needs to be `dict`. The keys need to be the name of the argument and the values the corresponding arguments. The default value is {}.
- `size` is the value that determines how many MCS are generated at once. It also changes dimensions of the output `ndarray` accordingly, so that its `shape` is `size + (iterations, )` (`(iterations, )` is a `tuple` with `iteration` in its single entry). The default value is `(1, )`.
- `unpack` specifies if the general n-dimensional structure of the output is discarded for a one-dimensional `ndarray` of the size `iterations`. This parameter only has an effect if `size` does equal `(1, )` (`tuple` with only a single `1` in it). The default value is `True`.

If called, the main function first checks if `size` is of the type `tuple`. Then it converts `theta`, `sigma` and `mu` into numpy arrays, if they are not all given as floats. After that conversion the array with the random numbers is generated and multiplied with $\sqrt{\texttt{dt}}$. Therefore the function given in the argument `noise_generator` is called. All arguments passed in `noise_parameter` are passed to this function as well as a `size` argument, which is set to `size + (iterations, )`. This returns an n-dimensional numpy array called `noise`, where the index of the last dimension denotes time and the other indices denote a particular MCS. When this array is generated, the start values given in `x0` are copied in `noise`, for that the last index of `noise`, the one responsible for time, is set to `0`. The next step is the main loop. Therefore Numpy slicing is used to efficiently calculate all MCS for an individual time step. To save memory, u_T is written in the `noise` array. To prevent collisions, u_{T+1} is stored at the same address as N_T. After u_{T+1} is calculated using N_T, N_T is overwritten with u_{T+1}. This is possible because N_T is not needed any more.

Additionally to the main function `ou_generate`, the module contains an `Iterator` class, some wrapper functions and a function that is used to convert the parameters `mu`, `theta` and `sigma` from different input types to the corresponding numpy arrays.

The `Iterator` can be used to iterate over each index-tuple of a multidimensional array. Therefore the initializer requires a `size`-argument. There the size of the array must be given as a tuple. For numpy arrays this is stored in the `shape` attribute. Then, the initialized object can be used as a generator in a for-loop. Multiple wrapper functions are provided, which can be used to make functions fulfill the requirements for in `ou_generate`, that do not fulfill them per se. There is `wrap_rename_size` to rename the size argument, if it has another name in a function. `wrap_additional_arguments` makes it possible to use more than only the size argument by saving values for the other parameters in the wrapper. If a function is needed as parameter, that has no size argument, it can also be wrapped by `wrap_no_size`, so that the given function is called once for each cell of the returning array. There is also the wrapper `wrap_no_size_and_additional_arguments` that combines the last two capabilities. It is written separately, because it is not trivial to chain the wrappers mentioned above together.

If seeds are required to make the generated time series reproducible, they can be set depending on the argument given in `noise_generator`. In the case that `noise_generator` is a number generator from the module `numpy.random`, as given in the default argument, the seed can be set by calling `numpy.random.seed` before calling `ou_generate`. This function takes zero or one arguments. If no argument is given, the seed is set to default[3]. If an argument (`int`) is given, the seed is set to the number given. If the used random generator supports getting a seed by an argument, this seed argument can be passed through to the noise generator by `noise_parameters`.

2.2.2 Parameter Analyser

2.2.2.1 Main Functions

The core of this module are the three functions that can calculate $|\alpha|$, $\alpha_\pm$ and $\sigma\sqrt{\Delta t}$. These are called `get_alpha_abs`, `get_alpha_pm` and `get_sigma` respectively. `get_sigma` and `get_alpha_pm` take a lower and upper limit instead of an ϵ. Although there are functions provided to calculate these limits from a fixed ϵ or an ϵ, that is set by properties of the time series.

The argument `data` that is used in all three main functions and some (all) of the others, has always the same meaning and requirements. `data` contains the OU time series that shall be analyzed. It needs to be an `ndarray` from the numpy module, with only one dimension. `data` is always interpreted as a time series with equal stime steps. Missing values can be given as `numpy.nan` to archieve constant time steps even with data that is not spaced equally by default.

[3]usually dependent on the system time

`get_sigma` implements the equation 2.10 as executable code. Its parameters are `data`, `lower` and `upper`.

- `data` is, as already described, an array of the OU time series.
- `lower`. If a value is above this value and below `upper` it is used to calculate σ.
- `upper`. See `lower`.

`get_alpha_abs` will be able to calculate equation 2.14 for the user when it is provided with a data set and $\sigma\sqrt{\Delta t}$, provided by the parameters `data` and `sigma`.

- `data` is, as already described, an array of the OU time series.
- `sigma` is a parameter of equation 2.14 and will be used in the equation as $\sigma\sqrt{\Delta t}$.

`get_alpha_pm` is the implementation of the calculation of $\alpha_\pm$ by the equation 2.15e. `get_alpha_pm` requires the parameters `data`, `mean`, `lower` and `upper`.

- `data` is, as already described, an array of the OU time series.
- `mean` will be used as μ from equation 2.15e
- `lower`. If a value is below this value or above `upper` it will be used to calculate $\alpha_\pm$.
- `upper`. See `lower`.

2.2.2.2 Limit Calculators

The following functions are used to calculate a mean, lower and upper limit that can be used as parameters for the main functions. It was chosen to calculate these limits in methods outside of the main methods, so that the user can change and try out different methods of setting an ϵ. They all return a tuple of three elements that can be used as `mean`, `lower` and `upper` (in this order) in the main functions. There are three predefined methods of setting the limit.

`set_limit_by_std` allows to set an ϵ symmetrical around the mean of the time series. The ϵ is determined as a multiple of the standard deviation of the time series. This method requires the two parameters `data` and `sigma`.

- `data` is, as already described, an array of the OU time series.

- `sigma` describes to how many standard deviations around the mean ϵ is set to. Its type is either `int` or `float`.

`set_limit_by_epsilon` calculates the mean, lower and upper limit for an absolute epsilon, given as a parameter. Its parameters are `data` and `epsilon`.

- `data` is, as already described, an array of the OU time series.
- `epsilon` is an absolute value. In its distance to the mean, the lower and upper limits are set.

`set_limit_by_percentage_of_mean` sets the lower and upper limit to be at a certain percentage of the mean. This method is only recommended when the deviations from the mean are much smaller than the mean. It needs the parameters `data` and `percent`.

- `data` is, as already described, an array of the OU time series.
- `percent` determines how far the limits are away from the mean,as a fraction of it.

2.2.2.3 Generic Application Functions

The following functions are programmed in a way that they can easily be used without a deeper understanding of the details of the program. The goal of all of these functions is to provide a good calculation of α and σ with only a few but significant parameters that work over wide ranges of different time series.

`all_by_std` calculates a σ and a α from a given time series by only using two float as arguments in addition to the time series itself. The parameters are `data`, `sigma_sigma` and `sigma_alpha`.

- `data` is, as already described, an array of the OU time series.
- `sigma_sigma` is equivalent to m_σ from section 2.3.
- `sigma_alpha` is equivalent to m_α from section 2.3.

There are other generic application functions in this module, but they were mainly used to explore different methods of how to set the ϵ-environment and are not recommended for use.

2.3 Calibration of the ϵ-Environment

Before the extraction program can be used to determine the parameters that are common for the Fermi LCs, it is necessary to first find a suitable algorithm to set the ϵ parameters of the equations 2.10 and 2.15e, introduced in section 2.1.5. In this context the quality of the parameter extraction is tested as well (sanity check).

In addition to `numpy`, which is used for the modules listed above, further packages are used for data reprocessing and visualisation of the results. These modules are:

- scipy 1.4.1
- matplotlib 3.1.3
- tqdm 4.42.1
- time (native module)

To avoid confusion, the ϵ of equation 2.10 that is used to calculate σ will be called ϵ_σ and the ϵ of equation 2.15e that is used to calculate $\alpha_\pm$ will be called ϵ_α.

2.3.1 Choosing the Method

The parameter space of μ and $\sigma\sqrt{\Delta t}$ is not tightly restricted ($\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$), thus it is not recommend to just set ϵ_σ and ϵ_α to a fixed value for all time series. If one did so, it would be very likely that either all, too few or none, of the points are within the ϵ boundary. Therefore a more sophisticated method needs to be found to set the ϵ by properties of the time series that are easy to obtain.

One method that satisfies these requirements is to set ϵ as a multiple (factor) of the standard deviation of the time series. This method is completely independent of μ and thus works equally well over the complete parameter space of μ, while also scaling with $\sigma\sqrt{\Delta t}$ and α to ensure there are always data points inside or respectively outside the ϵ-environment. These multiple of the standard deviation are be called m_σ and m_α respectively.

As part of this work other methods are explored, but then discarded in favor of the method using the standard deviation. These methods are suffering from the absence of decoupling from μ or being biased by the time series (sample) size. Discarding these methods is not only done on how they would behave on different parameters of μ and σ, also on the basis of tests done similar to 2.3.2, but on a smaller scale, that shows the presumed issues.

	Spearman R	Person R	Kendal-τ	AVG
m_σ	0.312	0.322	0.393	0.343±0.036
m_α	1.00	1.60	1.84	1.48±0.36

TABLE 2.1: m_σ and m_α with the best result in respect to each statistical test and the average

2.3.2 Evaluating the Best Parameters

For getting the best possible values for m_σ and m_α, a multitude of time series are generated with the OU-generator, which is described in sec 2.2.1 . Then, an ϵ-environment for each time series is calculated for a multitude of m_σ and m_α. For each of these ϵ-environments σ and θ are extracted using the program from sec. 2.2.2 and the algorithm developed in sec. 2.1.5. For each (unique) combination of m_σ and m_α the extracted values are then compared to those which are used to generate the time series using three statistical tests. The results of these tests can be seen in Fig. 2.1.

The tests used to determine correlation between the extracted $\sigma\sqrt{\Delta t}$ and $\theta\Delta t$ and the given $\sigma\sqrt{\Delta t}$ and $\theta\Delta t$, those that are used to generate the time series, are correlation coefficients by Spearman (Spearman R), Pearson (Pearson R) and Kendall (Kendall-τ).

For the generation of the OU time series, θ is drawn from $N(5,5)$, σ from $N(0,1)$ and μ from $N(0,1)$. Δt is set to 0.1. 100,000 time series are generated, but those where $2 > \theta\Delta t > 0$, are discarded as not stationary for all extractions and evaluations, so that more then 80,000 time series remain. A seed was used, to assure that the results are reproducible. The python code for the generation can be found in Appendix B.

2.3.3 Determination of the Maximum

To determine the best values for m_σ and m_α, the gradient in m_σ direction is calculated first. This gives a value for each unique combination of m_σ and m_α. For each m_σ the absolutes of all gradients (over all m_α) are summed. The m_σ for that this sum is the lowest, is the best. Then, the gradients in m_α direction are calculated for m_σ that is determined to be the best. The m_α where the absolute of the gradient is the lowest, is determined as the best m_α.

This is done for all three statistical tests. The results for the best m_σ and m_α are slightly different. Thus, the mean and standard error of m_σ and m_α are calculated from the results of all tests.

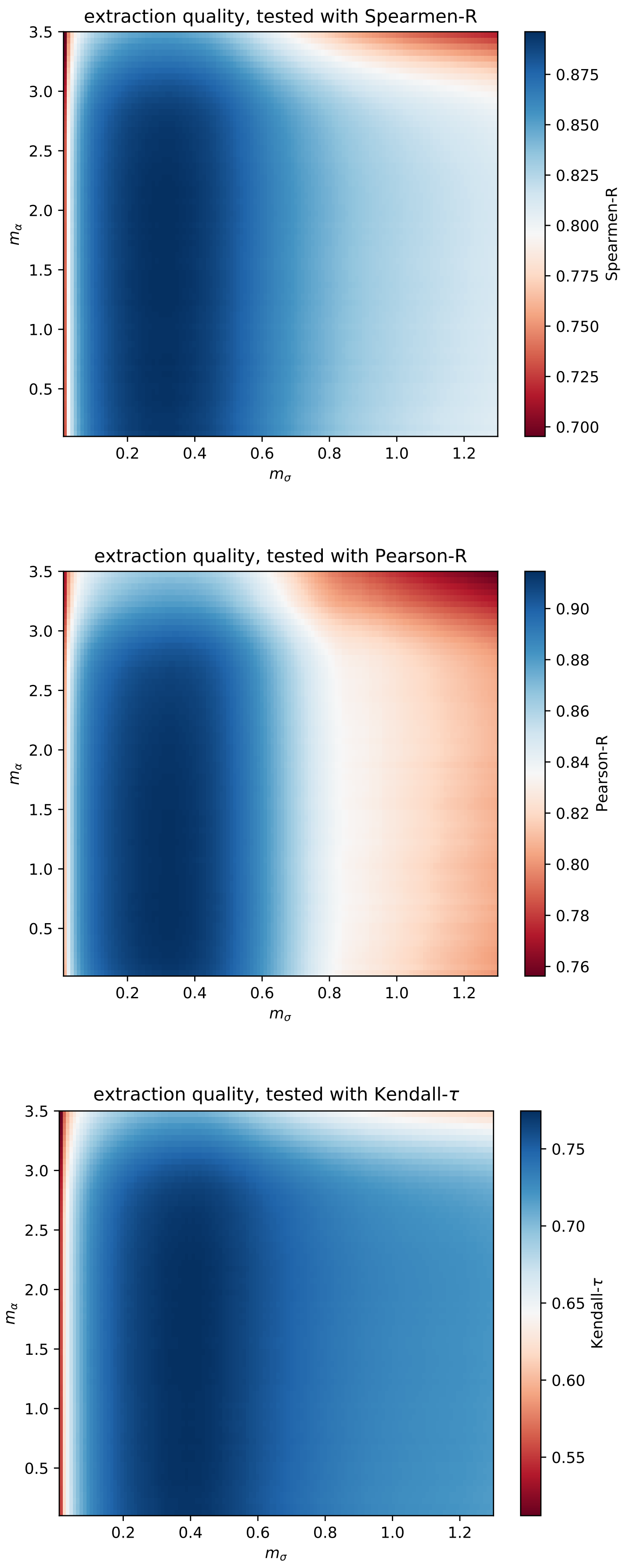

FIGURE 2.1: Quality of the extraction measured with different correlation coefficients, between the extracted θ and the given θ, while m_α and m_σ are varied

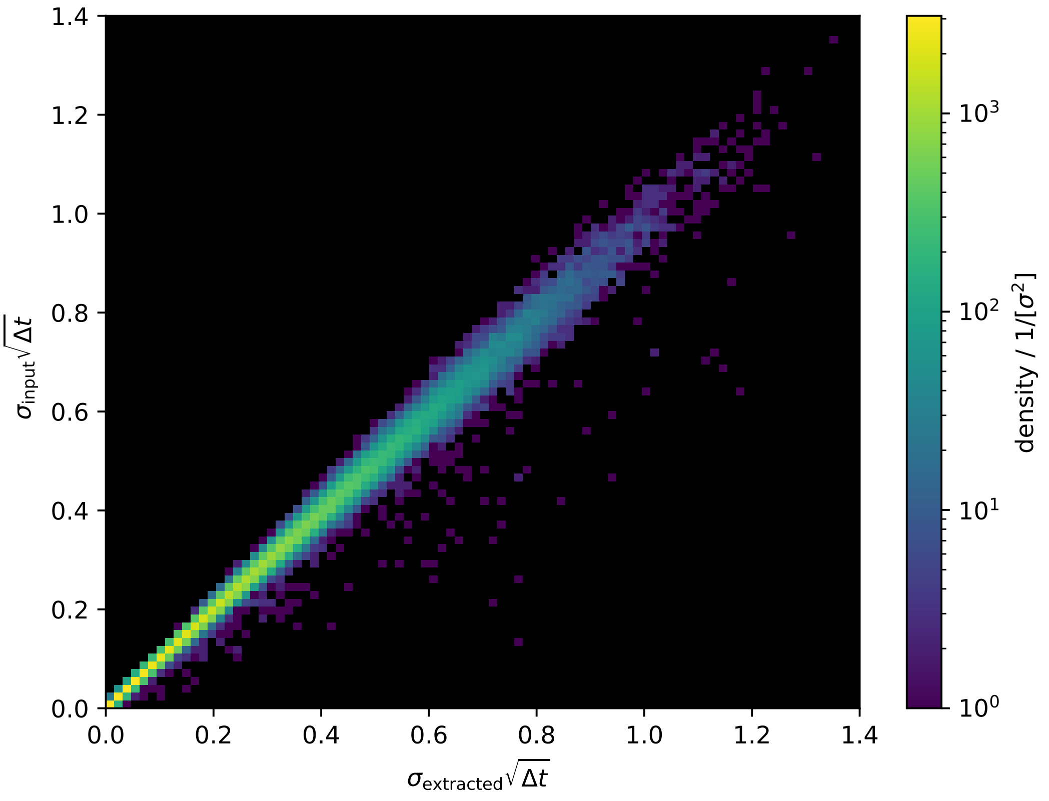

FIGURE 2.2: Density plot of the extracted $\sigma\sqrt{\Delta t}$, against the actual $\sigma\sqrt{\Delta t}$ parameter used to generate the time series.

2.3.4 Verification of the Extraction Method

To extract σ and α from the OU time series used for calibration, the optimized $m_\sigma = 0.343$ and $m_\alpha = 1.48$ are applied. This sanity check ensures the validity of the extracted parameters and functionality of the algorithm. Therefore the extracted $\sigma\sqrt{\Delta t}$ are plotted against the given ones used to generate the time series in fig. 2.2. The same was done for $\theta\Delta t$ in fig. 2.3.

As one can see, the algorithm accomplishes its task. It is also apparent that its quality degrades for $\theta\Delta t \to 1$. This was expected, because the approximation of small ϵ breaks at this point.

2.3.5 Error Estimation of the Method

To estimate the accuracy of the parameter extraction, a cone that encompasses 68% of all data points is fitted to the data from section 2.3.4. For the estimation of the $\theta\Delta t$-error only $\theta\Delta t < 1$ are used in the calculation. From the opening angle of the cone a relative error of $\sigma\sqrt{\Delta t}$ and $\theta\Delta t$ can be found. Due to the nature of the spread of the

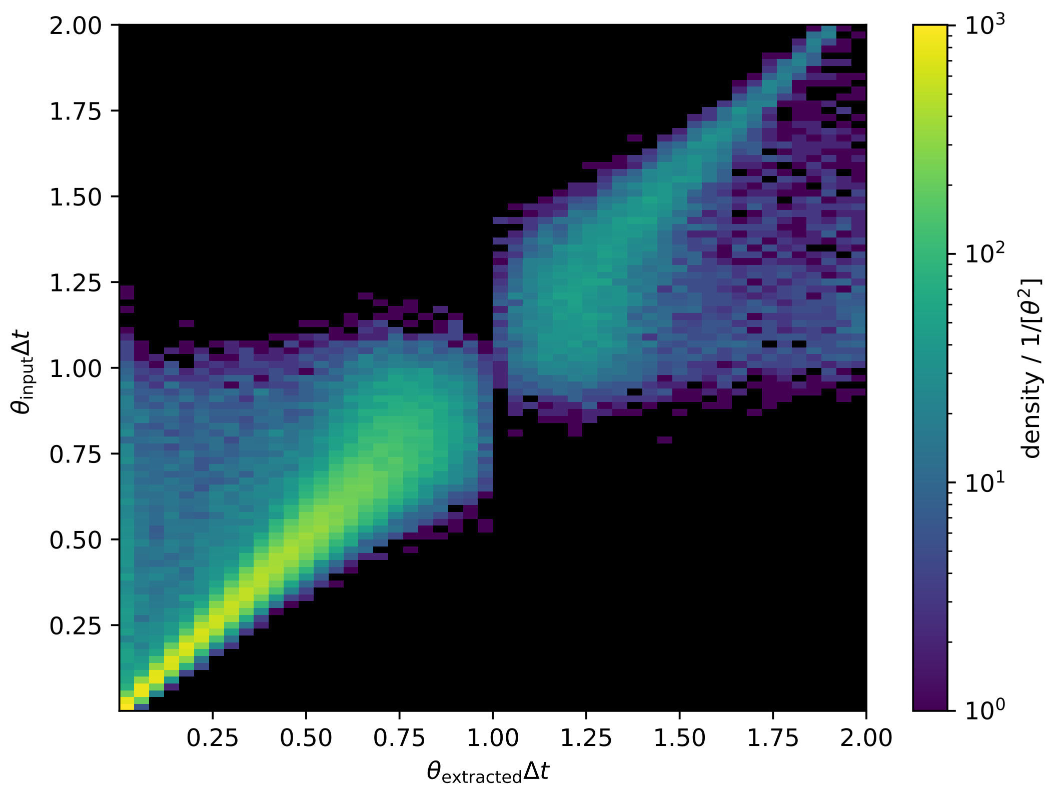

FIGURE 2.3: Density plot of the extracted $\theta\Delta t$ against the actual $\theta\Delta t$ parameter used to generate the time series. The gap in the middle equals a α close to 0. In this case assumption for the extraction breaks and the estimation gets less precise.

extraction it seems more useful to give a relative error than a absolute one. The relative errors $\sigma_{\theta\Delta t}$ and $\sigma_{\sigma\sqrt{\Delta t}}$ are calculated to the values in tab 2.2:

$\sigma_{\theta\Delta t}$	$\sigma_{\sigma\sqrt{\Delta t}}$
17.55 %	4.45%

TABLE 2.2: Relative Errors for $\sigma\sqrt{\Delta t}$ and for $\theta\Delta t < 1$

The constant relative error for $\theta\Delta t$ only makes sense for $\theta\Delta t < 1$. For $\theta\Delta t > 1$ the absolute error can be calculated with

$$\sigma_{abs,\theta\Delta t} = \sigma_{\theta\Delta t} \cdot (2 - \theta\Delta t). \tag{2.18}$$

Using this equation is equivalent of fitting a cone with the same opening angle from the other side.

Chapter 3

Results

For the extraction the OU-parameters and the subsequent simulations, not the fluxes measured by the Fermi/LAT, will be used but instead their logarithm (base 10). Therefore it is necessary to have a clear nomenclature that can distinguish between source and logarithmic or expontionated data. In the nomenclature chosen, "OU" and "Fermi" identify the source, while "time series" is the data where OU-parameters are extracted from or data can be generated as. "LC" is 10 to the power of the time series. So in case of Fermi they are the actual flux values measured by the Fermi/LAT. These relations are shown clearly in in table 3.1.

3.1 Extraction of the Parameters

The extraction of the OU-parameters is done to all points of the Fermi time-series, where the test statistic for the detected flux is greater or equal to 9 ($TS \geq 9$). Furthermore, if a time series has less than 38% significant data points left, it is discarded completely. After this procedure 253 of the 2278 LCs remain.

	OU	Fermi
time series	gen	$log_{10}(flux)$
light curve (LC)	10^{gen}	$flux$

Table 3.1: Clarification of nomenclature. While $flux$ is the flux (in $\frac{\text{erg}}{s \cdot cm^2}$) measured by the Fermi/LAT and gen are the numbers given by the OU-Generator described in section 2.2.1

3.2 Building a Random Number Generator

To generate OU-LCs that resemble the Fermi data, it is necessary to define random number generators (RNG) that draw their numbers from the distributions that the parameters extracted from the Fermi light curve have. To understand the distribution of the Fermi-LCs, all extracted parameters $\mu, \theta\Delta t, \sigma\sqrt{\Delta t}$ of all LCs fulfilling the test statistic are plotted in a histogram which is shown in fig. 3.1. A normal distribution is fitted over the histogram. The normal distributions are then numerically integrated, to obtain the cumulative distribution function (CDF) for each parameter.

Base of the RNG is the basic `numpy.random.random` number generator that has a uniform distribution between 0 and 1. To generate a random number with the wanted distribution, a number is drawn from `numpy.random.random`. Then, it is calculated at which value the CDF will reach that drawn number. That value is then returned as the random number that has the desired probability density function (PDF).

3.3 Comparison of the Statistical Properties of Real and Generated LCs

To check if the extraction of the parameters is successful, LCs generated with the distributions derived from the Fermi data are compared with those used to calibrate the parameter extraction. This is done to make sure that the extraction does not simply rebuild the distribution that was used for calibration, in the following called prior distribution. Therefore 100,000 time series were simulated for the distributions of μ, $\theta\Delta t$, $\sigma\sqrt{\Delta t}$ extracted in figure 3.1.

To compare the two sets of LCs, the power spectrum density (PSD) is calculated and compared for the LC originating from the prior distribution and the distribution extracted from the Fermi data, in the following called posterior distribution.

PSDs are commonly used in AGN research to quantitatively describe the variability of a source as it is done by Abdo et al. (2010). Going further, Timmer & Koenig (1995) even build a random number generator to generate artificial X-ray data by a PSD power-law slope.

The PSD can be approximated by the periodogram. The periodogram as it is defined by Timmer & Koenig (1995) reads:

$$P(\omega) = \frac{1}{n}\sum_{t=0}^{n} |x_t e^{-i\omega t}|^2. \tag{3.1}$$

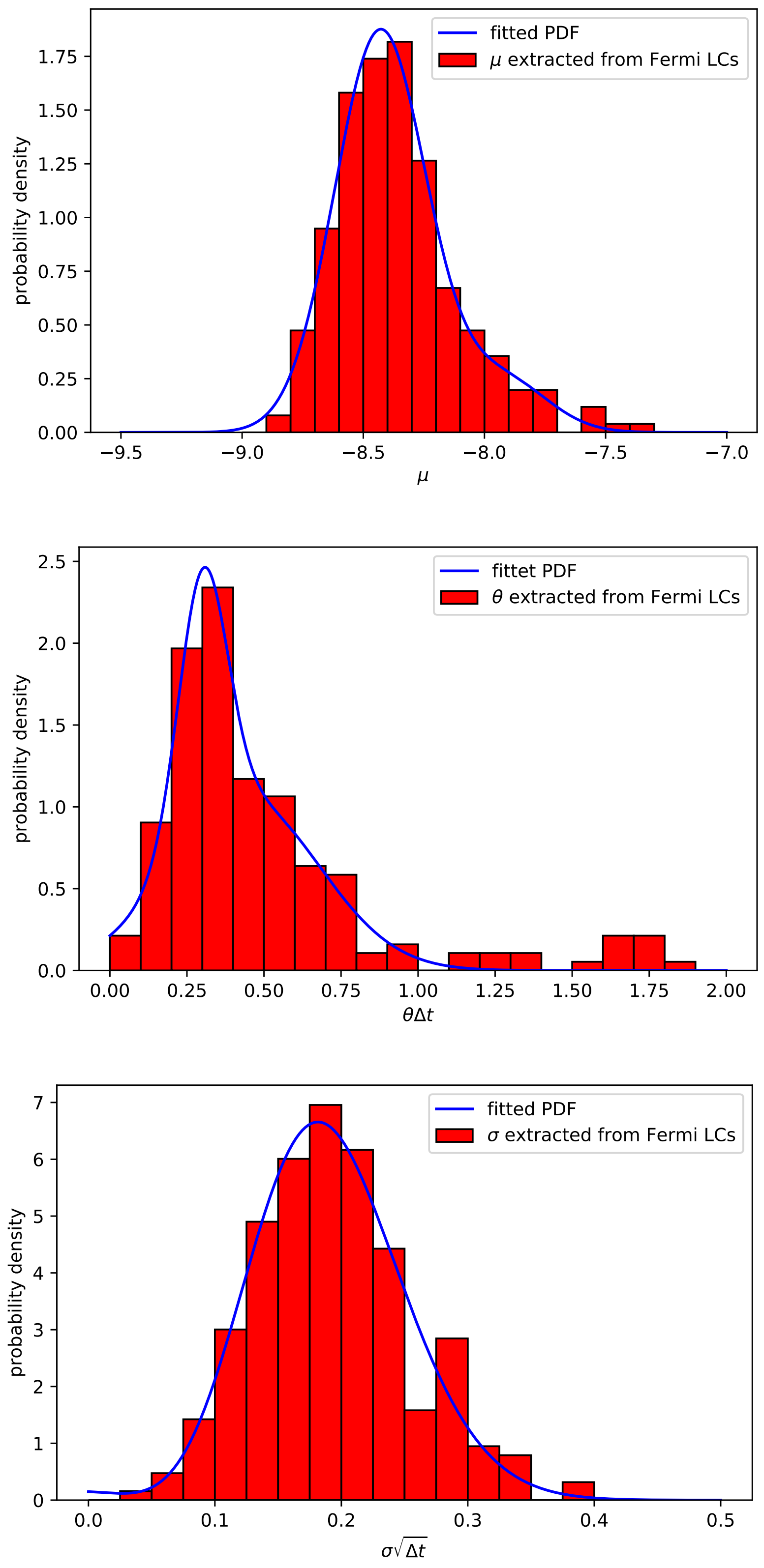

FIGURE 3.1: PDF of μ, $\theta\Delta t$, $\sigma\sqrt{\Delta t}$ extracted from the Fermi LCs

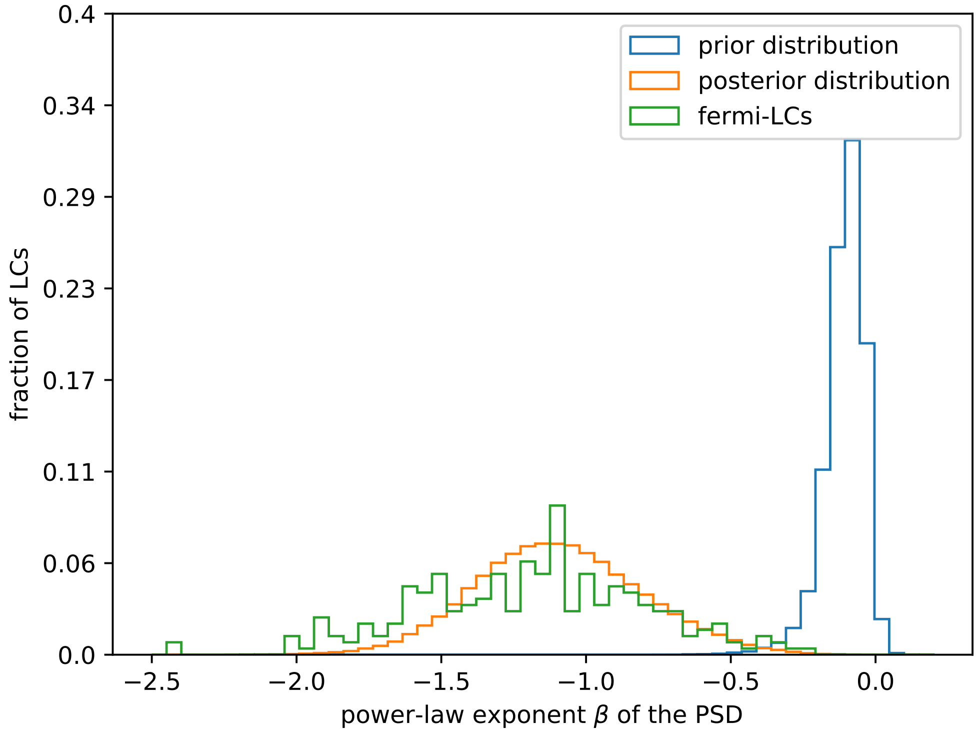

FIGURE 3.2: Distribution of the power-law exponent β of the PSD for the OU-LCs used in the calibration of m_α and m_σ, see section 2.3 (blue), the Fermi-LCs (green) and the OU-LCs generated to replicate the Fermi-LCs with the OU-parameter distribution fitted in section 3.2 (orange)

In this thesis the periodiogram was calculated by the algorithm of Lomb (1976) and Scargle (1982). An implementation of this can be found in the `scipy` module for python. To compare the PSDs a power law of the form ω^β is fitted to $P(\omega)$. The exponent β that is obtained for each LC can now be compared.

The orange histogram of fig. 3.2 shows how well the OU process could replicate the PSD of Fermi-LCs (green) once the distributions for the OU-parameters are calibrated. To show that the adjustment of the OU-parameter distributions is successful and the distribution of β is not an inherent property of the OU process, β of the distribution used to calibrate m_α and m_σ, see section 2.3, is also calculated and plotted in comparison.

To have a more quantitative measure of how well the distributions are equal to each other, the fist 4 central moments are compared to each other. For mean and variance the implementation of the `numpy` module is used, for the skewness and kurtosis the `scipy` module is used. The results for the central moments can be found in tab. 3.2. The means of the two distributions can not be distinguished. The skewness of both distributions is small. While the OUs are basically indistinguishable from 0, the Fermi-LCs are slightly left leaning but with such a low value that it is still compatible with 0 and thus with

central moment	Fermi-LCs	OU-LCs
mean	-1.2	-1.1
variance	0.16	0.083
skewness	-0.17	0.095
kurtosis	-0.038	0.087

TABLE 3.2: Statistical properties of the distribution of β for the Fermi-LCs and the OU-LCs build to imitate the Fermi-LCs. The distributions are shown in fig. 3.2 as histograms.

each other. For the kurtosis both values are so low that they are compatible with 0 and with each other. The only parameter where there is a relevant difference in the two distribution is the variance.

The slightly left leaning of the Fermi-LCs as well as the higher variance can both be explained by the so called red noise leakage. In contrast to the computer generated OU-LCs, the Fermi-LCs are not sampled equidistant. While there is a flux value given for each month per LC, filtering the test-statistic yields LCs that are not sampled equidistant anymore. Calculating the periodiogram for non equidistant sampled LCs comes with a bias towards lower frequencies, this is called red noise leakage. If one calculates a power law slope with these, this yields a lower β. While the Lomb-Scargle Algorithm used to calculate the periodiogram is designed to reduce the red noise leakage, it can not be eradicated completely.

Chapter 4

Summary and Outlook

In this thesis, a method to extract OU-parameters from a time series is described. After showing that this method as well as its implementation work properly on mock data, the method is employed to extract OU-parameters from gamma-ray data obtained with the Fermi telescope. On the basis of 253 lightcurves satisfying the test-statistics requirements, distributions for each of the three parameters of the OU process are obtained. Drawing random values from these distributions, a set of artificial Fermi-lightcurves is generated. Using power spectral densities for the amplitude variations as a reference for metric purposes, it is shown that within errors, the artificial lightcurves show the same statistics as the original data.

Even though the results are promising in showing that a simple (3-parameter) description for the complex variability patterns found in Fermi-LAT lightcurves is possible, further research needs to be done: This includes especially a closer look on data from other energy ranges of the electromagnetic spectrum. The emission at different wavelengths may originate from different regions of the AGN governed by different stochastic or correlated processes. Further studying the wavelength-dependent lightcurves may thus help to resolve the physical processes at work in these powerful gamma-ray sources.

In particular, a comparison between the X-ray/optical lightcurves of accretion-disc dominated AGN and the gamma-ray lightcurves of jet-dominated AGN may provide important clues about the connection between disk and jet. Furthermore, other time scales of variability, especially those that are more rapid than monthly, need to be investigated. Rather compact regions within the AGN seem to be involved in the gamma-ray emission corresponding to length scales much shorter than a light-travel distance of one month, given the reported shortes time scales of minutes for blazars. Such investigations will help to discern the radiation processes giving rise to the gamma-ray variability of blazars.

List of Figures

List of Tables

Bibliography

Abdo A.A., Ackermann M., Ajello M., et al., 2010, ApJ 722, 520

Abdo A.A., Ackermann M., Ajello M., et al., 2011, The Astrophysical Journal 736, 131

Albert J., Aliu E., Anderhub H., et al., 2007, ApJ 669, 862

Antonucci R., 1993, ARA&A 31, 473

Antonucci R., Barvainis R., 1990, ApJ 363, L17

Atwood W.B., Abdo A.A., Ackermann M., et al., 2009, ApJ 697, 1071

Beckmann V., Shrader C.R., 2012, Active Galactic Nuclei

Fermi E., 1949, Phys. Rev. 75, 1169

Gillespie D.T., 1996a, Phys. Rev. E 54, 2084

Gillespie D.T., 1996b, American Journal of Physics 64, 225

IceCube CollaborationAartsen M.G., Ackermann M., et al., 2018, Science 361, eaat1378

Kelly B.C., Becker A.C., Sobolewska M., et al., 2014, The Astrophysical Journal 788, 33

Kelly B.C., Sobolewska M., Siemiginowska A., 2011, ApJ 730, 52

Kreter M., 2018, Ph.D. thesis, JMU Würzburg

Lomb N.R., 1976, Astrophys. Space. Sci. 39, 447

Mannheim K., 1993, A&A 269, 67

Peterson B.M., Ferrarese L., Gilbert K.M., et al., 2004, ApJ 613, 682

Rybicki G.B., Lightman A.P., 1986, Radiative Processes in Astrophysics

Scargle J.D., 1982, ApJ 263, 835

Shakura N.I., Sunyaev R.A., 1973, A&A 24, 337

Shukla A., Mannheim K., Patel S.R., et al., 2018, 854, L26

Sikora M., Kirk J.G., Begelman M.C., Schneider P., 1987, ApJ 320, L81

Takata T., Mukuta Y., Mizumoto Y., 2018, The Astrophysical Journal 869, 178

Takata T., Mukuta Y., Mizumoto Y., 2019, The Astrophysical Journal 879, 132

Timmer J., Koenig M., 1995, Astronomy and Astrophysics 300, 707

Uhlenbeck G.E., Ornstein L.S., 1930, Phys. Rev. 36, 823

Ulrich M.H., Maraschi L., Urry C.M., 1997, ARA&A 35, 445

Urry C.M., Padovani P., 1995, PASP 107, 803

Vestergaard M., Peterson B.M., 2006, ApJ 641, 689

Acknowledgements

I thank Prof. Dr. Karl Mannheim for giving me the challenge and opportunity to work in research, particularly in such an interesting field as astronomy. In particular I'm grateful for his insightful inputs and his trained eye, which objections to my work need to be considered.

I thank Paul R. Burd for the long discussions and helpful tips and explanations. Also for giving me the great opportunity to work with you on a publication during this time, as well as sharing an office with me, so I wouldn't go mad. He also helped to debug and sanity check my programming code, doing so he was a lot more useful than a rubber duck.

I thank Prof. Dr. Sara Buson for providing the already binned data form the Fermi-telescope, as well as Jeffrey Scargle and Greg Madjeski, for the discussions during our meetings, especially in the light, that he is experienced on the topic of statistical progresses.

I thank Henri Scheppach, Theodor Kaiser, Leonard Müller for discussions and finding errors in grammar and spelling as well as for the moral support.

I thank my parents for supporting me in my decisions, so that I was able to reach my goal of studying physics.

Declaration of authorship

I, Luca Kohlhepp, declare that this thesis titled, 'A Study of the Statistic Nature of γ-ray Variability of Blazars' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Luca Kohlhepp

Date: 21.04.2020

Appendix A

Python Code of the Main Modules

Formating of the python code can seem off, because it was formated to be easily read in the source code. The formating was not changed here, so that the lines shown here coincide with the source code lines.

In this Appendix, the main modules are shown. Their documentation can be found in section 2.2.

If those modules contain a "main" method (`if __name__ == '__main__':`), they are either left over for texting the validity of the module or to show case it. These modules are not to be run as independent programs, but as modules to be included into other programs.

```python
import numpy as np


class DimensionError(Exception):
    pass


# iterates over all indices of an n-dimensional array, which shape is given in
    the 'size' argument
class Iterator:

    def __init__(self, size):
        self.size = size
        self.itlist = [0]*len(size)

    def __iter__(self):
        while self.itlist[-1] < self.size[-1]:
            for i in range(len(self.size)):
                if self.itlist[i] >= self.size[i]:
                    self.itlist[i] = 0
                    self.itlist[i+1] += 1
            yield tuple(self.itlist)
            self.itlist[0] += 1


# converts multiple types into the type np.ndarray
def to_nparray(para, size, all_para_floats):
    if callable(para):
        para = para(size)
    elif type(para) == np.ndarray:
        if not para.shape == size:
            raise DimensionError("shape of parameter is not equal to size")
    elif hasattr(para, '__iter__'):
        para = np.array(para)
        if not para.shape == size:
            raise DimensionError("shape of parameter is not equal to size")
    elif not all_para_floats:
        para = np.full(size, para)
    return para


# Main-function generates a MC simulated noise by the Ornstein-Uhlenbeck-SDE
def ou_generate(iterations, theta, sigma, mu, x0, dt=1, noise_generator=np.random.standard_normal, noise_parameters={}, size=(1, ), unpack=True):
    # validation and preparation of the parameters

    # size is forced into a tuple
    if type(size) == int:
        size = (size, )
    size = tuple(size)

    # check if all run parameters (theta, sigma, mu) are float if not make them all to arrays
    all_para_floats = isinstance(theta, (int, float)) and isinstance(sigma, (int, float)) and isinstance(mu, (int, float))
    theta = to_nparray(theta, size, all_para_floats)
    sigma = to_nparray(sigma, size, all_para_floats)
    mu = to_nparray(mu, size, all_para_floats)

    # theta is only used with dt, so it will be once resized here
    theta = theta * dt
    # mu is only used with theta, so this value is calculated once here
    mu = theta * mu

    # generate the noise and iter_size
    noise_parameters['size'] = size+(iterations, )
    noise = noise_generator(**noise_parameters)
    # iter_size is an iterator, that iterates over a slice of the size 'size'. So it can be used to iterate over the
    # complete theta, sigma, mu arrays or over a slice, that represents one time step, of the noise array
    iter_size = tuple([slice(0, length, 1) for length in size])

    # resize noise with time step length
    noise = noise*dt**0.5

    # initialize start values in the noise array, where t=0 (last index = 0)
    if x0 is None:
        pass
    elif callable(x0):
        x0 = x0(size)
        noise[iter_size+(0, )] = x0[iter_size]
    elif type(x0) == np.ndarray:
        if x0.shape == size:
            noise[iter_size+(0, )] = x0[iter_size]
        else:
            raise DimensionError("parameter shape is not equal to size")
    elif hasattr(x0, '__iter__'):
        x0 = np.array(x0)
        if x0.shape == size:
            noise[iter_size+(0, )] = x0[iter_size]
        else:
            raise DimensionError("parameter shape is not equal to size")
    else:
        noise[iter_size+(0, )] = x0

    # main iteration process
    if all_para_floats:
        for i in range(iterations-1):
            noise[iter_size+(i+1, )] = noise[iter_size+(i, )] * (1-theta) + mu + sigma*noise[iter_size+(i+1, )]
    else:
        for i in range(iterations-1):
            noise[iter_size+(i+1, )] = noise[iter_size+(i, )] * (1-theta[iter_size]) + mu[iter_size] + sigma[iter_size]*noise[iter_size+(i+1, )]

    # discards the wrapping array in case size = (1, ) and feature is enables
    if size == (1, ) and unpack:
        return noise[0, 0:iterations]
    else:
        return noise


# These are wrappers, to make functions usable as arguments in ou_generate, that do not full fill it prerequisites

# Use this if your function has no size argument, but also doesn't need more arguments.
# It works by calculating the function once per cell in an size shaped array,
# saving the result in an array and returning that array.
def wrap_no_size(func):
    def wrapper(size, *args, **kwargs):
        result = np.zeros(size, dtype=float)
        for i in Iterator(size):
            result[i] = func(*args, **kwargs)
        return result
    return wrapper


```

```
# This wrapper is useful, when the method has size, but also needs additional
    arguments. This is not necessary,
# when the noise function needs additional arguments, these can be given as
    kwargs in noise_parameter

# sizepos is the position of size in args (if in args). Dummy value must be given
    .
# If size not in args, sizepos needs to be set to -1
def wrap_additional_arguments(func, args, kwargs, sizepos=-1):
    def wrapper(size):
        if sizepos != -1:
            args[sizepos] = size
        else:
            kwargs['size'] = size
        return func(*args, **kwargs)
    return wrapper


# This wrapper is useful when the function already has a size argument, but it is
     named differently.
def wrap_rename_size(func, name):
    def wrapper(*args, **kwargs):
        kwargs[name] = kwargs['size']
        del kwargs['size']
        return func(*args, **kwargs)
    return wrapper


# This combines the need of additional arguments and the lack of a size argument.
     This was implemented,
# because it is not trivial to wrap no_size and additonal_arguments into each
    other.
def wrap_no_size_and_additional_arguments(func, args, kwargs):
    def wrapper(size):
        result = np.zeros(size, dtype=float)
        for i in Iterator(size):
            result[i] = func(*args, **kwargs)
        return result
    return wrapper
```

Listing A.1: OU-Generator, file: ou_generator.py

```
import numpy as np


# General:
# feed with np.array with flux only, non significant values, need to be set to np
    .nan


# Some basic exceptions used (subject to change)

# This exception is thrown, if some values that shouldn't are NaN
class IsNANException(Exception):
    pass
```

```python


# Implementation of the given formulas, from paper/thesis. Instead of a epsilon
    an upper and lower limit are given


# returns a sigma*sqrt(dt) and the number of points used to calculate it
def get_sigma(data, lower, upper):
    # positions of u_T, that are NOT NaN AND over lower AND under upper
    pos = np.array((~np.isnan(data))*(data > lower)*(data < upper), dtype=bool)
    # discards last element (no u_T+1 would exist)
    pos[-1] = False
    # positions of u_T+1 (shift positions by +1)
    pos1 = np.zeros(len(pos), dtype=bool)
    pos1[1:] = pos[:-1]
    # calculates u_t+1 - u_t for all u_t
    distance = data[pos1] - data[pos]
    # discards all that are NAN
    distance = distance[~np.isnan(distance)]
    # if no points are left, no sigma can be calculated
    if len(distance) == 0:
        return np.nan, 0
    # standard deviation is calculated and returned
    return np.std(distance), len(distance)


def get_alpha_abs(data, sigma):
    return np.sqrt(1-(sigma**2/np.var(data[~np.isnan(data)])))


def get_alpha_pm(data, mean, lower, upper):
    # positions of u_T, that are NOT NaN AND (over lower OR under upper)
    pos = np.array((~np.isnan(data))*((data < lower)+(data > upper)), dtype=bool)
    # discards last element (no u_T+1 would exist)
    pos[-1] = False
    # positions of u_T+1 (shift positions by +1)
    pos1 = np.zeros(len(pos), dtype=bool)
    pos1[1:] = pos[:-1]
    # calculates all the alphas at once
    alphas = (data[pos1] - mean)/(data[pos] - mean)
    if len(alphas) == 0:
        return np.nan
    # calculates the mean of all alphas that are not NaN
    return np.mean(alphas[~np.isnan(alphas)])


# auxiliary functions to easily calculate limits


# gets limits in terms of standard deviations form mean
# this is the method chosen to be used in the thesis/paper
def set_limit_by_std(data, sigma):
    mean = np.mean(data[~np.isnan(data)])
    std = np.std(data[~np.isnan(data)])
    if mean == np.nan or sigma == np.nan or std == np.nan:
        raise IsNANException()
    return mean, mean - sigma * std, mean + sigma * std


# get limits form epsilon distance
def set_limit_by_epsilon(data, epsilon):
    mean = np.mean(data[~np.isnan(data)])
    if mean == np.nan:
        raise IsNANException()
    return mean, mean - epsilon, mean + epsilon


def set_limit_by_percentage_of_mean(data, percent):
    mean = np.mean(data[~np.isnan(data)])
    if mean == np.nan:
        raise IsNANException()
    return mean, mean * (1 - percent), mean * (1 + percent)


# calculates alpha and/or sigma directly, with use of the functions above


def sigma_by_percentage_of_mean(data, percent):
    limits = set_limit_by_percentage_of_mean(data, percent)
    return get_sigma(data, *limits[1:])


def alpha_by_std(data, sigma_para, sigma_std):
    limits = set_limit_by_std(data, sigma_std)
    return np.sign(get_alpha_pm(data, *limits)) * get_alpha_abs(data, sigma_para)


# calculates sigma and alpha by the method described in the thesis/paper (
    sigma_sigma <=> m_\sigma, sigma_alpha <=> m_\alpha
def all_by_std(data, sigma_sigma, sigma_alpha):
    limits_sigma = set_limit_by_std(data, sigma_sigma)
    limits_alpha = set_limit_by_std(data, sigma_alpha)
    sigma, n = get_sigma(data, *limits_sigma[1:])
    return sigma, np.sign(get_alpha_pm(data, *limits_alpha))*get_alpha_abs(data,
    sigma), n
```

Listing A.2: Parameter extractor for σ and α, file: get_para.py

```python
import numpy as np


# PDF refers to probability density function
# CDF refers to cumulative distribution function


# Iterator class, used in random_array to iterate over a n-dimesional np array -
    copied from ou_generator
# iterates over all indices of an array, which shape is given in the 'size'
    argument
class Iterator:

    def __init__(self, size):
        self.size = size
        self.itlist = [0]*len(size)

    def __iter__(self):
        while self.itlist[-1] < self.size[-1]:
            for i in range(len(self.size)):
                if self.itlist[i] >= self.size[i]:
                    self.itlist[i] = 0
                    self.itlist[i+1] += 1
            yield tuple(self.itlist)
            self.itlist[0] += 1


# this is the main random generator
# cdf refers to an np.ndarray, that contains the numeric CDF (numeric integration of the PDF)
# cdf[0] needs to be 0 and cdf[-1] needs to be 1
def random(cdf, min_limit, max_limit):
    # draws random number from uniform distribution
    ran = np.random.random()
    # gets the position where the cdf is the first time bigger then random value
    over_ran = np.where(cdf > ran)[0][0]
    # the value before that is the last where random is bigger
    under_value = cdf[over_ran-1]
    # calculates the "float position" ran would have, if cdf would be continuous sampled
    #       last pos bigger then ran :: linear interpolation between the points bigger and smaller then ran
    float_pos = (over_ran-1) + (ran - under_value) / (cdf[over_ran] - under_value)
    # calculate a value from the "float position" (from the uniform sampling of the CDF)
    return min_limit + (max_limit - min_limit) * (float_pos / (len(cdf)-1))


# this is version uses the random function, but has a additional size argument. A random number is generated for each
# cell of a size shaped array
def random_array(cdf, min_limit, max_limit, size):
    result = np.zeros(size, dtype=float)
    # calculates a random number for each cell in result
    for i in Iterator(size):
        result[i] = random(cdf, min_limit, max_limit)
    return result


# if you only have a PDF, calculate a fitting CDF here
def cdf_by_pdf(pdf):
    cdf = np.zeros(len(pdf)+1, dtype=float)
    for i in range(len(pdf)):
        cdf[i+1] = cdf[i] + pdf[i]
    cdf[0] = 0
    cdf[-1] = 1
    return cdf


if __name__ == "__main__":
    # tests only
    cdf = np.array([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1])
    print(random(cdf, 0, 1))
    pdf = np.array([0.1]*10)
    print(cdf_by_pdf(pdf))
```

Listing A.3: Random number generator, file: random_by_cdf.py

Appendix B

Python Code of the Scripts

As well as for Appendix A, the code shown is directly from the programs and thus not specially formated.

This code here doesn't have the module like nature of the code of Appendix A, even if parts of it may be used as such. It is special tailored to the exact use case of the situation and data. For that reason things like paths or the use of very specific data structure, are coded directly into the program code. In most cases paths and other variables are initialised in the "main" (`if __name__ == '__main__':`) and not set in the functions themselves, so that it should be easy to change code, or use the program as a module instead, even though it is not designed as such.

Might not be as well documented and commented as the main modules

```python
import numpy as np
import ou_generator as ou
import time


"""
The format used to save the np.array with the OU-time series is a 2d ndarray with dtype=object.
Its shape is (amount, 4), where amount is the amount of time series saved in the array.
In the second dimension the [time series, theta, mu, sigma] for each time series is stored in this order.
"""


"""
While this is code is not written to be imported as a module and calling this as one will generate the exact
time series used in the thesis/paper, one can shift all the non essential code into a if __name__ == '__main__' clause.
```

```
It might be easier to just copy the repack2 method into the program where it is
    needed.
"""


# set the amount, length and dt of the time series to generate
amount = 100000
length = 1190
dt = 0.1
seed = 982947937

# just renaming the fuction, so that one does not need to write np.random.normal
    all the time
normal = np.random.normal


# not tested if order stays the same
# doesn't work, don't use, use repack2 instead
def repack(theta, sigma, mu, noise):
    dummyList = []
    one_chain = slice(0, noise.shape[-1], 1)
    for i in ou.Iterator(theta.shape):
        #dummyList.append([noise[i+(one_chain, )], theta[i], mu[i], sigma[i])
        pass
    return np.reshape(np.array(dummyList), theta.shape+(4, ))


# repacks the generated time series in the chosen format
def repack2(theta, sigma, mu, noise):
    result = np.empty(theta.shape+(4, ), dtype=object)
    for i in ou.Iterator(theta.shape):
        one_chain = slice(0, noise.shape[-1], 1)
        result[i+(0,)] = noise[i+(one_chain, )]
        result[i+(1,)] = theta[i]
        result[i+(3,)] = sigma[i]
        result[i+(2,)] = mu[i]
    return result


# generate parameters
# the parameters are generated out of function, so that they can be saved with
    the generated numbers
np.random.seed(seed)
theta = normal(5, 5, amount)
sigma = normal(0, 1, amount)
mu = normal(0, 1, amount)

# for some manual checking (not required)
print(type(mu) == np.ndarray)

# generating ou_noise
print('generating ou processes')
start = time.time()
# generates the ou-time series with the ou_generator module
```

```python
generated = ou.ou_generate(length, theta, sigma, mu, None, dt=dt, size=(amount, )
    )
print('finished in {}s'.format(time.time()-start))
# some prints mean to mu parameter; sanity check if generator works and doesn't
    mess up order.
for i in range(len(mu)):
    print('mu: {}, <x>: {}'.format(mu[i], np.mean(generated[i])))
# repack to array form needed for next step
start = time.time()
print('repackaging')
print(generated.shape)
# repacks it in the format used during the project
array = repack2(theta, sigma, mu, generated)
print(array.shape)
print('finished in {}s'.format(time.time()-start))
print('saving')
start = time.time()
# saves the array using the np module
np.save('ouV4.npy', array)
print('finished in {}s'.format(time.time()-start))
print('exit')
```

Listing B.1: Generates the spezific time series used in 2.3.4, file: newBulk.py

```python
import numpy as np
import get_para as gp
import os
from scipy.stats import ks_2samp as ks, spearmanr as sr, pearsonr as pr,
    kendalltau as kl
import tqdm


"""

"""


# data: [time series, theta, mu, sigma]

class QueueElement:

    def __init__(self, name, alpha_func, sigma_func, alpha_paras, sigma_paras):
        self.name = name
        self.alpha_func = alpha_func
        self.sigma_func = sigma_func
        self.alpha_paras = alpha_paras
        self.sigma_paras = sigma_paras


# this calculates sigma and alpha (2x) with given functions and given parameters
    for one time series
def single_estimator(single_data, alpha_func, sigma_func, alpha_para, sigma_para,
     sigma_given):
    # If the sigma or alpha parameters are iterable they will be unpack when
    thrown into
    # the limit generating funtion. This allows for limit functions that use more
     then one argument
    if hasattr(sigma_para, '__iter__'):
        sig_limit = sigma_func(single_data, *sigma_para)
    else:
        sig_limit = sigma_func(single_data, sigma_para)
    if hasattr(alpha_func, '__iter__'):
        alpha_limit = alpha_func(single_data, *alpha_para)
    else:
        alpha_limit = alpha_func(single_data, alpha_para)

    # estimation of sigma
    sigma_est = gp.get_sigma(single_data, *sig_limit[1:])[0]
    # estimation of alpha with estimated sigma
    alpha_est = gp.get_alpha_abs(single_data, sigma_est) * np.sign(gp.
    get_alpha_pm(single_data, *alpha_limit))
    # estimation of alpha with given sigma
    alpha_giv = gp.get_alpha_abs(single_data, sigma_given) * np.sign(gp.
    get_alpha_pm(single_data, *alpha_limit))
    # return values as tuple
    return sigma_est, 1 - alpha_est, 1 - alpha_giv


# this calculates sigma and alpha (2x) with given functions and given parameters
    for all time series
def estimator(data, alpha_func, sigma_func, alpha_para, sigma_para, dt):
    # !!! the array cells at (0, x) are reserved for the parameters of the
    functions !!!
    estimation = np.zeros((data.shape[0] + 1, 3), dtype=float)
    # sets the input values in the first cell
    estimation[0, 0], estimation[0, 1], estimation[0, 2] = sigma_para, alpha_para
    , alpha_para
    # iterates over all times series
    for i in tqdm.tqdm(range(data.shape[0]), desc='time series', leave=False):
        estimation[i + 1, 0], estimation[i + 1, 1], estimation[i + 1, 2] =
    single_estimator(data[i, 0], alpha_func,

                   sigma_func, alpha_para,

                   sigma_para,

                   data[i, 3] * np.sqrt(dt))
    return estimation


# this calculates sigma and alpha (2x) for one function each over a given
    parameter space for each.
# the results per parameter are saved and the match between the calculated and
    given sigma and alphas are calculated
# with the functions given in test.
def one_function(data, path, name, alpha_func, sigma_func, alpha_paras,
    sigma_paras, dt, tests=[ks, sr, pr, kl],
                 test_names=None):
    # sets standard names for standard tests
    if test_names is None and tests == [ks, sr, pr, kl]:
```

```python
        test_names = ['ks', 'sr', 'pr', 'kl']
    statistics = np.empty((len(tests) + 1, len(alpha_paras) + 1, len(sigma_paras) + 1, 3), dtype=object)
    # writes used tests and parameters in the 0th line of the array
    for i in range(len(tests)):
        statistics[i + 1, 0, 0, 0] = test_names[i]

    for i in range(len(alpha_paras)):
        statistics[0, i + 1, 0, 0] = alpha_paras[i]

    for i in range(len(sigma_paras)):
        statistics[0, 0, i + 1, 0] = sigma_paras[i]
    # writes it's own name in [0, 0, 0, 0]
    statistics[0, 0, 0, 0] = name
    # creates output dir if necessary
    if not os.path.exists(f'{path}/{name}'):
        os.mkdir(f'{path}/{name}')
    # main iteration over both parameter spaces
    # for i in range(len(alpha_paras)):
    for i in tqdm.tqdm(range(len(alpha_paras)), desc='alpha'):
        for k in tqdm.tqdm(range(len(sigma_paras)), desc='sigma', leave=False):
            estimation = estimator(data, alpha_func, sigma_func, alpha_paras[i], sigma_paras[k], dt)
            np.save(f'{path}/{name}/est-{name}#{i}-{k}.npy', estimation)
            for m in range(len(tests)):
                # test quality of sigma
                statistics[m + 1, i + 1, k + 1, 0] = tests[m](
                    np.abs(data[:, 3][~np.isnan(estimation[1:, 0])] * np.sqrt(dt)),
                    estimation[1:, 0][~np.isnan(estimation[1:, 0])])
                # test quality of theta
                statistics[m + 1, i + 1, k + 1, 1] = tests[m](data[:, 1][~np.isnan(estimation[1:, 1])] * dt,
                                                              estimation[1:, 1][~np.isnan(estimation[1:, 1])])
                statistics[m + 1, i + 1, k + 1, 2] = tests[m](data[:, 1][~np.isnan(estimation[1:, 2])] * dt,
                                                              estimation[1:, 2][~np.isnan(estimation[1:, 2])])
    np.save(f'{path}/stat-{name}.npy', statistics)


# this can execute one_function multiple times, so it is easier to queue multiple functions with the same tests
def multiple_functions(data, path, dt, queue, tests=[ks, sr, pr, kl], test_names=None):
    # sets standard names for standard tests
    if test_names is None and tests == [ks, sr, pr, kl]:
        test_names = ['ks', 'sr', 'pr', 'kl']

    for element in tqdm.tqdm(queue, desc='functions'):
        one_function(data, path, element.name, element.alpha_func, element.sigma_func, element.alpha_paras,
                     element.sigma_paras, dt, tests=tests, test_names=test_names)

```

```python

if __name__ == '__main__':
    testqueue = [QueueElement('test', gp.set_limit_by_std, gp.set_limit_by_std, [1, 2, 3, 4], [0.05, 0.1, 0.5, 1])]
    data = np.load("ouV4.npy", allow_pickle=True)
    stableindices = np.where(np.array(np.where(data[:, 1] < 20, True, False) * np.where(data[:, 1] > 0, True, False), dtype=bool))[0]
    newdata = np.empty((len(stableindices), 4), dtype=object)
    for i in range(newdata.shape[0]):
        newdata[i, :] = data[stableindices[i], :]
    del data
    print('data loaded and cleand sucessfully, starting the fun part')
    # the next line is for testing purposes only
    # multiple_functions(newdata, './Extract', 0.1, testqueue)
    # generates the queue for testing all methods (also the rejected) for a small sampling in a chosen parameter space
    queue1 = [QueueElement('std-std', gp.set_limit_by_std, gp.set_limit_by_std, [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4],
                           [0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1]),
              QueueElement('per-std', gp.set_limit_by_percentage_of_mean, gp.set_limit_by_std,
                           [0.1, 0.2, 0.5, 1, 1.5, 2], [0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1]),
              QueueElement('std-per', gp.set_limit_by_std, gp.set_limit_by_percentage_of_mean,
                           [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4], [0.02, 0.05, 0.1, 0.2, 0.5, 1]),
              QueueElement('per-per', gp.set_limit_by_percentage_of_mean, gp.set_limit_by_percentage_of_mean,
                           [0.1, 0.2, 0.5, 1, 1.5, 2], [0.02, 0.05, 0.1, 0.2, 0.5, 1]),
              QueueElement('max-max', gp.set_limit_by_min_max, gp.set_limit_by_min_max, [0.2, 0.5, 0.8],
                           [0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3]),
              QueueElement('max-std', gp.set_limit_by_min_max, gp.set_limit_by_std, [0.2, 0.5, 0.8],
                           [0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1]),
              QueueElement('std-max', gp.set_limit_by_std, gp.set_limit_by_min_max, [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4],
                           [0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3])
              ]
    # uncomment the following line to test the rejected methods
    # multiple_functions(newdata, './Extract2', 0.1, queue1, tests=[sr, pr, kl], test_names=['spearmanr', 'personr', 'kendalltau'])
    # generates queue for std-std method only, this was found to be the best method. For that reason a finer sampling
    # was calculated, to find the maximum quality better.
    queue_std = [QueueElement('std-std-more', gp.set_limit_by_std, gp.set_limit_by_std, np.linspace(0.1, 3.5, 70 - 1),
                              np.linspace(0.01, 1.3, 129))]
    # this starts the main calculation
    multiple_functions(newdata, './Extract-std', 0.1, queue_std, tests=[sr, pr, kl],
                        test_names=['spearmanr', 'personr', 'kendalltau'])
    print('finished')
```

Listing B.2: This codes test multiple methods and parameters, to select and ϵ-environment, file: best_epsilon.py

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


"""
This program will get the maximum for each test. It also includes some plotting (
    I know that isn't a nice separation)
"""


# repackaging into [test, alpha, sigma, extraction]
def get_2dmesh(d4array):
    d2mesh = np.zeros((d4array.shape[0] - 1, 3, d4array.shape[1] - 1, d4array.
    shape[2] - 1))
    zeroed = d4array[1:, 1:, 1:, :]
    for i in range(zeroed.shape[0]):
        for k in range(zeroed.shape[1]):
            for m in range(zeroed.shape[2]):
                for n in range(zeroed.shape[3]):
                    d2mesh[i, n, k, m] = zeroed[i, k, m, n][0]
    return d2mesh


# calculates the minimum of an array in sigma direction, by summing over alpha
    space
def get_sigma_minimum(mesh):
    return np.argmin([np.sum(mesh[:, i]) for i in range(mesh.shape[1]-1)])


def plot(raw_data, mesh, path, test_index, para_index, test, para):
    plt.pcolormesh(raw_data[0, 0, 1:, 0], raw_data[0, 1:, 0, 0], mesh[i, k], cmap
    ='RdBu')
    plt.xlabel('$m_{\\sigma}$')
    plt.ylabel('$m_{\\alpha}$')
    plt.title(f'extraction quality, tested with {test}')
    cbar = plt.colorbar()
    cbar.set_label(f'{test}')
    plt.savefig(f'{path}{test_index}-{para_index}.pdf', dpi=300)
    plt.close()


def plot_grad(raw_data, mesh, path, test, test_index, para, para_index):
    print(mesh.shape)
    plt.pcolormesh(raw_data[0, 0, 1:, 0], raw_data[0, 1:, 0, 0], mesh[test_index,
     para_index], cmap='gist_yarg',norm=colors.LogNorm(vmin=1E-5,vmax =1E-1))
    plt.xlabel('$m_{\\sigma}$')
    plt.ylabel('$m_{\\alpha}$')
    plt.title(f'gradient of the extraction quality, tested with {test}')
```

```python
    plt.colorbar()
    plt.savefig(f'{path}{test_index}-{para_index}-grad.pdf', dpi=300)
    plt.close()


if __name__ == '__main__':
    raw_data = np.load('stat-std-std-more.npy', allow_pickle=True)
    mesh = get_2dmesh(raw_data)

    plt.pcolormesh(raw_data[0, 0, 1:, 0], raw_data[0, 1:, 0, 0], mesh[0, 0], cmap='RdBu')
    plt.show()

    # calculate the absolutes gradients of all 3 tests in both (alpha and sigma) direction
    gradients = np.zeros(mesh.shape + (2,), dtype=float)
    for i in range(mesh.shape[0]):
        for k in range(mesh.shape[1]):
            # gradient indizes: [test, parameter, x, y, direction]
            gradients[i, k, :, :, 0], gradients[i, k, :, :, 1] = np.abs(np.gradient(mesh[i, k]))
    #print(gradients[0, 1, :, :, 0])
    #plt.pcolormesh(raw_data[0, 0, 1:, 0], raw_data[0, 1:, 0, 0], gradients[1, 1, :, :, 1], cmap='gist_yarg',norm=colors.LogNorm(vmin=1E-5,vmax =1E-1))
    #plt.show()

    # gets sigma, where the gradient is as small as possible
    # averaged over all tests, this is the best parameter for sigma
    best_sigma = [get_sigma_minimum(gradients[i, 1, :, :, 1]) for i in range(3)]
    best_sigma_values = [raw_data[0, 0, 1:, 0][i] for i in best_sigma]
    print(best_sigma_values)
    print(f'{np.mean(best_sigma_values)}  {np.std(best_sigma_values)}')

    # for the best sigma (not the averaged, but for each test), the minimum of the gradient in alpha direction is computed
    # here also is averaged over all tests
    best_theta = [np.argmin(gradients[i, 1, :, best_sigma[i], 0]) for i in range(3)]
    best_theta_values = [raw_data[0, 1:, 0, 0][i] for i in best_theta]
    print(best_theta_values)
    print(f'{np.mean(best_theta_values)}  {np.std(best_theta_values)}')

    testlist = ['Spearmen-R', 'Pearson-R', 'Kendall-$\\tau$']
    paralist = ['$\\sigma$', '$\\theta$', '$\\theta_{giv}$']

    # visulizes the results from above as plots
    for i in range(len(testlist)):
        for k in range(len(paralist)):
            #plt.pcolormesh(raw_data[0, 0, 1:, 0], raw_data[0, 1:, 0, 0], mesh[i, k], cmap='RdBu')
            #plt.show()
            plot(raw_data, mesh, './', i, k, testlist[i], paralist[k])

            plot_grad(raw_data, gradients[:, :, :, :, 0], './0-', testlist[i], i, paralist[k], k)
        plot_grad(raw_data, gradients[:, :, :, :, 1], './1-', testlist[i], i,
paralist[k], k)
```

Listing B.3: This gets the optimum parameter for each test (only one function) it works with the files generated by `best_epsilon.py`. In line 79, between the mean and the standard deviation is a ± symbol in the python code, that can not be represented correctly in LaTeX, while viewing code, file: get-min.py

Appendix C

Validity of the Updating Formula

For this calculation the exact updating forumula from Gillespie (1996a) is required. First is shown that the updating formula is exact while not depending on the step size. To show that the approximate updating formula (eq. 2.14) corresponds to a 1st order taylor expansions for small Δt it is required to show that the OUDE is fulfilled for infinitesimal time steps.

The exact updating formula as given by Gillespie (1996a) is

$$x(t+\Delta t) = x(t)e^{-\frac{\Delta t}{\tau}} + \left[\frac{c\tau}{2}\left(1-e^{\frac{2\Delta t}{\tau}}\right)\right]^{\frac{1}{2}} N \tag{C.1}$$

Gillespie (1996a) uses a different nomenclature ($\theta = 1/\tau$ and $\sigma = \sqrt{c}$), the correspondence to the parameters used here will be become apparent later. Also x is named X and $N(t)$ is named n.

First it is shown that the Eq.C.1 is exact. For this following expression must hold

$$x(t+\Delta t_1+\Delta t_2) = x(t+\Delta t_3), \tag{C.2}$$

where $\Delta t_3 = \Delta t_1 + \Delta t_2$. Therefore:

$$x(t+\Delta t_1+\Delta t_2) = x(t+\Delta t_1)e^{-\frac{\Delta t_2}{\tau}} + \left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_2}{\tau}}\right)\right]^{\frac{1}{2}} N_2 \overset{!}{=} x(t+\Delta t_3) \tag{C.3}$$

$$\left(x(t)e^{-\frac{\Delta t_1}{\tau}} + \left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_1}{\tau}}\right)\right]^{\frac{1}{2}} N_1\right)e^{-\frac{\Delta t_2}{\tau}} + \left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_2}{\tau}}\right)\right]^{\frac{1}{2}} N_2 \overset{!}{=} x(t+\Delta t_3) \tag{C.4}$$

$$x(t)e^{-\frac{\overbrace{\Delta t_1+\Delta t_2}^{=\Delta t_3}}{\tau}} + e^{-\frac{\Delta t_2}{\tau}}\left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_1}{\tau}}\right)\right]^{\frac{1}{2}} N_1 + \left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_2}{\tau}}\right)\right]^{\frac{1}{2}} N_2 \overset{!}{=} x(t+\Delta t_3) \tag{C.5}$$

$$x(t)e^{-\frac{\Delta t_3}{\tau}} + \left[\frac{c\tau}{2}\left(e^{-\frac{2\Delta t_2}{\tau}} - e^{-\frac{2\overbrace{(\Delta t_1+\Delta t_2)}^{=\Delta t_3}}{\tau}}\right)\right]^{\frac{1}{2}} N_1 + \left[\frac{c\tau}{2}\left(1-e^{-\frac{2\Delta t_2}{\tau}}\right)\right]^{\frac{1}{2}} N_2 \overset{!}{=} x(t+\Delta t_3) \tag{C.6}$$

Using the properties of the normal distributions of equations 2.6b and 2.6c, yields

$$x(t)e^{-\frac{\Delta t_3}{\tau}} + \left[\frac{c\tau}{2}\left(e^{-\frac{2\Delta t_2}{\tau}} - e^{-\frac{2\Delta t_3}{\tau}} + 1 - e^{-\frac{2\Delta t_2}{\tau}}\right)\right]^{\frac{1}{2}} N_3 \overset{!}{=} x(t+\Delta t_3) \tag{C.7}$$

$$x(t)e^{-\frac{\Delta t_3}{\tau}} + \left[\frac{c\tau}{2}\left(-e^{-\frac{2\Delta t_3}{\tau}} + 1\right)\right]^{\frac{1}{2}} N_3 \overset{!}{=} x(t+\Delta t_3)\square \tag{C.8}$$

Because it is proven that eq. C.1 is exact, it is now required to show that it also fulfills the OUDE. Therefore let $\Delta t \to 0$, then

$$e^{-\frac{\Delta t}{\tau}} = 1 - \frac{\Delta t}{\tau} + \mathcal{O}(\Delta t^2). \tag{C.9}$$

Term of $\mathcal{O}(\Delta t^2)$ or higher order are negligible, thus

$$e^{-\frac{\Delta t}{\tau}} = 1 - \frac{\Delta t}{\tau} \tag{C.10}$$

Therefore eq. C.1 becomes

$$x(t+\Delta t) = x(t)\left(1-\frac{\Delta t}{\tau}\right) + \left[\frac{c\tau}{2}\left(1-1+\frac{2\Delta t}{\tau}\right)\right]^{\frac{1}{2}} N \tag{C.11}$$

$$x(t+\Delta t) = x(t)\ \underbrace{\frac{1}{\tau}}_{\theta}\ x(t)\Delta t + \underbrace{\sqrt{c}}_{\sigma}\ \sqrt{\Delta t}N \tag{C.12}$$

For $\Delta t \to 0$, $\Delta t = dt$ and for small Δt, this is the approximate updating formula found in 2.14.