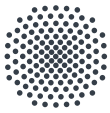

Universität Stuttgart

Elementare Wahrscheinlichkeitstheorie mit SymPy

Bachelorarbeit
im Studiengang Mathematik

```
>>> x = sym.Symbol('x', real=True)
>>> mu = sym.Symbol('mu', real=True)
>>> sigma = sym.Symbol('sigma', real=True, positive=True)
>>> density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
>>> normal_distribution = RandomVariableContinuous(density, x)
>>> normal_distribution.characteristic_function()
exp(t*(I*mu - sigma**2*t/2))
```

Durchgeführt am
Institut für Stochastik und Anwendungen
Fachbereich Mathematik
Universität Stuttgart

Abgabedatum: 06.08.2024

mathematik

Inhaltsverzeichnis

Abstract

Die folgende Zusammenfassung wurde von Bing Copilot erstellt und geringfügig überarbeitet.

Diese Bachelorarbeit beschäftigt sich mit der Implementierung einer Python-Bibliothek zur Berechnung von Momenten, Verteilungsfunktionen und Simulationen von reellen Zufallsvariablen. Die Bibliothek basiert auf dem symbolischen Rechenpaket SymPy und nutzt dessen Fähigkeiten zur Integration, Ableitung und Vereinfachung von Ausdrücken. Die Bibliothek unterstützt verschiedene Typen von Zufallsvariablen und erlaubt die Definition von eigenen Verteilungen basierend auf Dichtefunktionen. Die Bachelorarbeit stellt die wichtigsten Konzepte und Funktionen der Bibliothek vor und illustriert deren Anwendung anhand von Beispielen aus der Wahrscheinlichkeitstheorie. Die Bachelorarbeit zeigt auch die Leistungsfähigkeit und Grenzen der Bibliothek auf und diskutiert mögliche Erweiterungen für die Zukunft.

Vorwort

Zu dieser Bachelorarbeit gehören einige Python Skripts. Diese finden Sie unter

https://github.com/Blondai/Elementare-Wahrscheinlichkeitstheorie-mit-SymPy

In diesem Ordner befindet sich eine `.whl`-Datei, welche sich mit

```
pip install ProbabilityTheoryWithSymPy-1.0.0-py3-none-any.whl
```

wie ein normales Python-Paket installieren lässt. Mittels

```
from ProbabilityTheoryWithSymPy import *
```

lassen sich alle Funktionalitäten importieren. Um nicht alles auf einmal zu importieren, lässt sich der Stern durch die entsprechenden Unterklassen ersetzen, welche im Verlauf dieser Bachelorarbeit vorgestellt werden. Des Weiteren lassen sich dort die vier Python-Dateien finden, auf denen diese Arbeit beruht.

Hinter den meisten Definitionen stehen die zugehörigen englischen Begriffe in Klammern. Dies dient vor allem dazu, die entsprechenden Methoden im Code zu finden. Der gesamte Code inklusive Kommentare wurde, wie in der Informatik üblich, auf Englisch verfasst.

Als nächstes möchte ich darauf hinweisen, dass es zu den meisten Definitionen und Sätzen passende Beispiele gibt. Zum einen sind die Rechnungen mit Zwischenschritten von Hand ausgeführt und zum anderen gibt es passende Codeschnipsel, die die Funktionalität des Programms zeigen sollen.

An dieser Stelle möchte ich etwas näher auf den dargestellten Code eingehen. Allgemein verwenden `Teile des Codes` diese monospaced Schriftart. Teile, die direkt aus dem Programm entnommen sind, werden folgendermaßen gezeigt

```
def foo(text):
    print(f"Hallo {text}!")
```

Dies sind meist Funktionen beziehungsweise Methoden, die anhand des gezeigten Codes etwas näher erläutert werden sollen. Beispiele für die Verwendung des Codes haben zusätzlich Zeilennummern

```
1 text = "Welt"
2 foo(text)
```

Der so gezeigte Code verwendet die standardmäßigen Python-Farben: Grün für Strings und Orange für Keywords, was leider bei der Ausgabe in LaTeX manchmal etwas verwirrend aussieht, wenn man `lambda` als Symbol λ verwendet und nicht die entsprechende Python-Funktionalität

meint. Es sei erwähnt, dass die sich in der Bachelorarbeit befindenden Codeschnipsel keine Kommentare und Docstrings enthalten. Diese sind im Programmcode zu finden und hätten entsprechende Teile nur unnötig verlängert.

Da die gesamte Arbeit in LaTeX verfasst ist, sind häufig klickbare Hyperlinks eingearbeitet, die zu den erwähnten Stellen führen. Diese sind blau gekennzeichnet. Ebenso sind eingebaute Links zu Webseiten blau. Die meisten Sätze, Definitionen, Beispiel sind prägnant benannt, um auf diese später zu verweisen.

Definitionen, Sätze und Ähnliches, die nicht aus Maß- und Wahrscheinlichkeitstheorie bekannt sind und mehr oder weniger wortwörtlich aus Büchern oder anderen Quellen entnommen wurden, sind entsprechend durch [Name der Quelle] gekennzeichnet. Diese Quellen befinden sich am Ende dieser Arbeit.

1 Einführung in SymPy

Das gesamte Projekt ist in Python (3.10) und SymPy (1.12) geschrieben. Außerdem ist für manche Funktionalitäten matplotlib (3.8.0) und NumPy (1.26.3) sowie eine LaTeX-Installation nötig. In den Klammern befindet sich jeweils die von mir verwendete Version. Die Programme sollten auch ohne Probleme mit neueren oder älteren Versionen funktionieren.

Wir wollen uns in diesem Kapitel mit einigen Grundfunktion von SymPy beschäftigen. Sollten schon Vorkenntnisse zu SymPy vorhanden sein, so kann dieser Teil übersprungen werden. Bei offenen Fragen ist die Dokumentation [Tea23] sehr hilfreich.

Wir werden häufig die folgenden englischen Abkürzungen verwenden.

`expr`	Symbolischer Ausdruck
`var`	Symbolische Variable
`lower`	Untere Grenze
`upper`	Obere Grenze
`int`	Ganze oder natürliche Zahl

Um im Code nicht jedes mal `sympy` ausschreiben zu müssen, wurde zu Beginn `import sympy as sym` verwendet.

Wir wollen nun mit SymPy Symbole definieren. Wollen wir beispielsweise der Python-Variablen `x` die mathematische Variable x zuweisen, so verwenden wir

```
x = sym.Symbol('x')
```

Es ist wichtig SymPy möglichst viele Informationen (assumptions) über diese Variable zu geben. Gibt man nichts weiter an, so wird die Variable als komplexe Zahl interpretiert. Die folgende Tabelle zeigt einige Einstellungsmöglichkeiten [Tea23].

Eigenschaft	SymPy Befehl
$x \in \mathbb{R}$	`real=True`
$x \in \mathbb{Q}$	`rational=True`
$x \in \mathbb{Z}$	`integer=True`
$x > 0$	`positive=True`
$x \geq 0$	`nonnegative=True`
$x < 0$	`negative=True`
$x \leq 0$	`nonpositive=True`
$x \neq 0$	`nonzero=True`

Diese Befehle lassen sich auch, solange kein Widerspruch entsteht, kombinieren. Möchte man beispielsweise `n` als eine positive natürliche Zahl definieren, so verwendet man

```
n = sym.Symbol('n', integer=True, positive=True)
```

Des Weiteren ist es möglich mehrere Symbole gleichzeitig zu definieren. Wollen wir zum Beispiel `x`, `y` und `z` als reelle Zahl definieren, so schreiben wir

```
x, y, z = sym.symbols('x, y, z', real=True)
```

Möchte man nun noch mehr Symbole auf einmal definieren, wie zum Beispiel $x_1, \ldots, x_5$, so geht das mit

```
x_1, x_2, x_3, x_4, x_5 = sym.symbols('x_1:6')
```

Es ist zu beachten, dass die letzte Zahl nicht eingeschlossen ist.

Wollen wir nun die Funktion $x^2 + 3x/y - \sqrt{z}$ definieren, so verwenden wir

```
expr = x**2 + 3 * x / y - sym.sqrt(z)
```

Wichtig ist, dass man die Malpunkte, wie in Python üblich, nicht weglassen darf. Außerdem wird nicht mit `^`, sondern mit `**` potenziert. Zu den meisten Funktionen, wie arcsin, exp, $\sqrt{\ }$ oder Γ, gibt es entsprechende SymPy Gegenstücke `sym.asin`, `sym.exp`, `sym.sqrt` oder `sym.gamma`.

Weiterhin ist es nötig, SymPy zu erklären, wenn man einen Bruch, wie beispielsweise $1/2$ als symbolischen Bruch definieren möchte. Verwenden wir, wie in Python üblich, `1 / 2`, so wird dies automatisch als Gleitkommazahl (float) interpretiert. Diese kann SymPy später nicht mehr richtig vereinfachen. Besondere Probleme machen periodische Zahlen oder beispielsweise die Summe `1 / 10 + 2 / 10`, was fälschlicherweise zu `0.30000000000000004` summiert wird. Möchte man also den Bruch $1/2$ definieren, so verwendet man

```
sym.Rational(1, 2)
```

Möchten wir hingegen das Symbol `x` halbieren, so ist `sym.Rational(x, 2)` nicht die richtige Herangehensweise und liefert Fehler, denn SymPy interpretiert schon `x / 2` als den symbolische Bruch $x/2$. Falls man in SymPy Unendlichkeiten verwenden möchte, so funktioniert dies mithilfe von `sym.oo`.

Um zu überprüfen, ob SymPy das Eingegebene auch richtig versteht, können wir unseren Ausdruck in die Funktion `sym.srepr` geben. Diese gibt dann genau den internen Aufbau unseres Ausdrucks wieder. Verwenden wir den Ausdruck von oben, so gibt SymPy folgendes aus:

```
Add(Pow(Symbol('x', real=True), Integer(2)), Mul(Integer(3), Symbol('x',
 real=True), Pow(Symbol('y', real=True), Integer(-1))), Mul(Integer(-1),
 Pow(Symbol('z', real=True), Rational(1, 2))))
```

Es fällt auf, dass SymPy Subtraktion durch Multiplikation mit -1 und Addition verarbeitet. Ähnlich wird Division durch Potenzierung mit -1 und Multiplikation dargestellt.

Nun werden wir einige Funktionen und Methoden betrachten, die im Programmcode häufig verwendet werden.

(i) `sym.Sum`
Die Funktion `sym.Sum(expr, (var, lower, upper))` berechnet die Summe eines Ausdrucks über eine Variable von der unteren bis zur oberen Grenze.

(ii) `sym.integrate`
Die Funktion `sym.integrate(expr, (var, lower, upper))` berechnet zum einen das Integral eines Ausdrucks über eine Variable von der unteren bis zur oberen Grenze. Zum anderen kann mit `sym.integrate(expr, var)` auch eine Stammfunktion des Ausdrucks bezüglich der Variablen bestimmen werden. Verwendet man `sym.Integral`, so erhält man das unevaluierte Integral.

(iii) `sym.diff`
Die Funktion `sym.diff(expr, (var, int))` berechnet die `int`-fache Ableitung eines Ausdrucks nach einer Variablen. Lässt man `int` weg, so wird einmal abgeleitet. Analog zur Integration kann `sym.Derivative` verwendet werden, um ein unevaluiertes Ableitungsobjekt zu erhalten.

(iv) `.doit`
Die Methode `expr.doit()` zwingt SymPy einen Ausdruck zu evaluieren.

(v) `sym.simplify`
Die Funktion `sym.simplify(expr)` erlaubt es SymPy einen Ausdruck zu vereinfachen. Diese Funktion kann unter Umständen viel Rechenzeit benötigen.

(vi) `.evalf`
Die Methode `expr.evalf(int)` zwingt SymPy einen Ausdruck mit einer bestimmten Anzahl an signifikanten Stellen zu berechnen. Dies sind standardmäßig zehn.

(vii) `.subs`
Die Methode `expr.subs(var, number)` kann dazu verwendet werden, in einem Ausdruck ein bestimmtes Symbol durch eine Zahl, ein anders Symbol oder einen ganzen Ausdruck zu ersetzen.

Um Plots zu bearbeiten kann es sinnvoll sein, sich mit matplotlib zu beschäftigen. Da dies nur ein recht kleiner Teil dieser Bachelorarbeit ist, wird auf nähere Erläuterung verzichtet. Ebenso wird NumPy eine untergeordnete Rolle spielen.

2 Grundlegende Begriffe

Zur Erinnerung werden wir in diesem Abschnitt einige grundlegende Begriffe aus der Maßtheorie und der Stochastik wiederholen. Die meisten Definitionen und Sätze lassen sich in ähnlicher Form auch in [Kle20] und [MS05] finden. Da dies Grundlagen aus Maß- und Wahrscheinlichkeitstheorie sind, wurde auf entsprechende Zitate verzichtet.

2.1 Zufallsvariablen

Die Objekte, mit denen wir uns im Folgenden beschäftigen wollen, sind reelle Zufallsvariablen. Dazu benötigen wir zunächst folgende

Definition 2.1 (σ-Algebra)**:** *Sei Ω eine beliebige Menge und $\mathscr{A} \subseteq \mathcal{P}(\Omega)$ ein Mengensystem. Gilt*

(S1) $\Omega \in \mathscr{A}$

(S2) $\forall A \in \mathscr{A} : A^{\complement} \in \mathscr{A}$

(S3) $\forall (A_n)_{n\in\mathbb{N}_0} \subseteq \mathscr{A} : \bigcup_{n\in\mathbb{N}_0} A_n \in \mathscr{A}$,

so ist $\mathscr{A}$ eine σ-Algebra und $(\Omega, \mathscr{A})$ ein Messraum.

Die Eigenschaft (S2) bezeichnet man auch als *Komplementstabilität* und die Eigenschaft (S3) als *σ-Vereinigungsstabilität.*

Beispiel 2.2 (σ-Algebren)**:** Wir werden nun einige einfache und häufig vorkommende Beispiele für σ-Algebren betrachten. Gegeben sei zunächst eine beliebige Menge Ω.

(i) Die kleinste (*gröbste*) σ-Algebra ist gegeben durch

$$\mathscr{A} = \{\emptyset, \Omega\} \ .$$

(ii) Die größte (*feinste*) σ-Algebra ist gegeben durch

$$\mathscr{A} = \mathcal{P}(\Omega) \ .$$

Ist Ω höchstens abzählbar, so verwenden wir genau diese Potenzmenge als σ-Algebra.

Sei nun $\Omega = \mathbb{R}$ die reellen Zahlen.

(iii) In diesem Fall verwenden wir die *Borelsche-σ-Algebra*

$$\mathscr{A} = \mathscr{B} \ .$$

Diese wird erzeugt von den offenen Mengen.

Sei weiter $\Omega = [a, b]$ ein reelles Intervall mit $a < b$.

(iv) In diesem Fall verwenden wir auch die *Borelsche-σ-Algebra*, müssen sie aber folgendermaßen einschränken

$$\begin{aligned}\mathscr{A} &= \mathscr{B}_{[a,b]} \\ &= \{B \cap [a, b] \mid B \in \mathscr{B}\} \ .\end{aligned}$$

Diese Konstruktion verläuft auch analog für andere Borel-Mengen.

Mit diesen Begriffen können wir nun Messbarkeit definieren.

Definition 2.3 (Messbarkeit)**:** *Gegeben seien zwei Messräume* $(\Omega_1, \mathscr{A}_1)$ *und* $(\Omega_2, \mathscr{A}_2)$*, sowie eine Abbildung* $f : (\Omega_1, \mathscr{A}_1) \to (\Omega_2, \mathscr{A}_2)$*. Gilt für alle* $A_2 \in \mathscr{A}_2$

$$f^{-1}(A_2) \in \mathscr{A}_1 \ ,$$

so nennt man f $\mathscr{A}_1$*-*$\mathscr{A}_2$*-messbar oder kurz messbar.*

Wir wollen nun ein paar Beispiele für messbare Abbildungen betrachten.

Beispiel 2.4 (Messbare Abbildungen)**:**

(i) Gegeben sei $(\mathbb{R}, \mathscr{B})$ und $f \in \mathcal{C}^0(\mathbb{R}, \mathbb{R})$ eine stetige Funktion. Dann ist f messbar. Sei $O_2 \in \mathscr{B}$ eine offene Menge. Nach Definition der Stetigkeit existiert eine weitere offene Menge $O_1 \in \mathscr{B}$, sodass

$$f^{-1}(O_2) = O_1$$

ist. Da die Borelschen-σ-Algebra von den offenen Mengen erzeugt wird, ist f messbar.

(ii) Gegeben sei $(\Omega, \mathscr{A})$ und ein $A \in \mathscr{A}$. Dann ist $\mathbb{1}_A$ messbar. Sei $B \in \mathscr{B}$. Betrachte nun die folgenden Fallunterscheidung

$$\begin{aligned} &0, 1 \in B && \mathbb{1}_B^{-1}(B) = \mathbb{R} \\ &0 \in B, 1 \notin B && \mathbb{1}_B^{-1}(B) = A^{\complement} \\ &0 \notin B, 1 \in B && \mathbb{1}_B^{-1}(B) = A \\ &0, 1 \notin B && \mathbb{1}_B^{-1}(B) = \emptyset \end{aligned}$$

Alle Mengen rechts sind nach Definition in der Borel-σ-Algebra.

(iii) Gegeben sei $\Omega = \{\omega_1, \omega_2, \omega_3\}$ und $\mathscr{A} = \{\emptyset, \{\omega_1, \omega_2\}, \{\omega_3\}, \Omega\}$. Weiter sei

$$\begin{aligned} f : (\Omega, \mathscr{A}) \to (\{0,1\}, \mathcal{P}(\{0,1\})) : \omega_1 &\mapsto 0 \\ \omega_2 &\mapsto 0 \\ \omega_3 &\mapsto 1 \ . \end{aligned}$$

Es gilt

$$\mathcal{P}(\{0,1\}) = \{\emptyset, \{0\}, \{1\}, \{0,1\}\} \ .$$

Betrachte also

$$\begin{aligned} f^{-1}(\emptyset) &= \emptyset \\ f^{-1}(\{0\}) &= \{\omega_1, \omega_2\} \\ f^{-1}(\{1\}) &= \{\omega_3\} \\ f^{-1}(\{0,1\}) &= \Omega \ . \end{aligned}$$

Da all diese Mengen in $\mathscr{A}$ liegen, ist f messbar. Verwenden wir hingegen $\{\emptyset, \Omega\}$ als σ-Algebra, so ist f nicht mehr messbar, da das Urbild von $\{0\}$ nicht in der σ-Algebra liegt.

Vor allem an diesem letzten Beispiel sehen wir, dass Messbarkeit stark von den verwendeten σ-Algebren abhängt.

Um unsere Reise durch die Maßtheorie fortsetzen zu können, benötigen wir noch das namensgebende Maß.

Definition 2.5 (Maß)**:** *Sei* $(\Omega, \mathscr{A})$ *ein Messraum und* $\mu : \mathscr{A} \to \overline{\mathbb{R}}$ *eine Abbildung. Gilt*

(M1) $\mu(\emptyset) = 0$

(M2) $\forall A \in \mathscr{A} : \mu(A) \geq 0$

(M3) $\forall (A_n)_{n\in\mathbb{N}_0} \subseteq \mathscr{A}$ *disjunkt* $: \mu\left(\bigsqcup_{n\in\mathbb{N}_0} A_n\right) = \sum_{n\in\mathbb{N}_0} \mu(A_n)$,

so ist μ *ein Maß und* $(\Omega, \mathscr{A}, \mu)$ *ein Maßraum. Existiert eine Folge* $(A_n)_{n\in\mathbb{N}_0} \subseteq \mathscr{A}$*, sodass deren Vereinigung* Ω *ist und* $\mu(A_n) < \infty$ *für alle* $n \in \mathbb{N}_0$ *ist, so nennt man* μ *sogar* σ*-endlich. Gilt zudem* $\mu(\Omega) = 1$*, so ist* μ *ein Wahrscheinlichkeitsmaß, welches man häufig mit* $\mathbb{P}$ *bezeichnen und* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum.*

Die Eigenschaften (M1) und (M3) bezeichnet man auch als *Nulltreue* respektive σ*-Additivität.* Die definierende Eigenschaft eines Wahrscheinlichkeitsmaßes bezeichnet man auch als *Normiertheit.*

Bemerkung 2.6 (σ-Endlichkeit von Wahrscheinlichkeitsmaß)**:** Sei $\mathbb{P}$ ein Wahrscheinlichkeitsmaß. Dann ist $\mathbb{P}$ insbesondere σ-endlich. Sei dazu $A_n = \Omega$ für alle $n \in \mathbb{N}_0$. Dann gilt offenbar

$$\bigcup_{n\in\mathbb{N}_0} A_n = \Omega \ .$$

Außerdem gilt für alle $n \in \mathbb{N}_0$

$$\mathbb{P}(A_n) = 1 < \infty \ ,$$

was zu zeigen war.

Wir werden nun einige Beispiele für Maße sehen.

Beispiel 2.7 (Maße)**:**

(i) Lebesgue-Maß λ: Dies ist das „natürliche“ Maß, dass wir schon aus der Schule kennt. Für $a < b$ aus $\mathbb{R}$ gilt

$$\lambda([a, b]) = b - a \ ,$$

was genau die Länge des Intervalls ist. Das Lebesgue-Maß ist ein Maß auf $(\mathbb{R}, \mathscr{B})$.

(ii) Zählmaß $\#$: Sei Ω höchstens abzählbar. Für ein $A \in \mathcal{P}(\Omega)$ gilt

$$\#(A) = |A| \ ,$$

was genau die Mächtigkeit der Menge ist. Das Zählmaß ist ein ein Maß auf $(\Omega, \mathcal{P}(\Omega))$.

(iii) Dirac-Maß $\delta_{\{\omega\}}$: Sei $(\Omega, \mathscr{A})$ Messraum. Für $A \in \mathscr{A}$ definieren wir

$$\delta_{\{\omega\}} := \begin{cases} 1 & , \omega \in A \\ 0 & , \omega \notin A \ . \end{cases}$$

Das Dirac-Maß zu $\omega \in \Omega$ ist sogar ein Wahrscheinlichkeitsmaß auf $(\Omega, \mathscr{A})$.

Die ersten beiden Maße sind die wichtigsten, da es dank der Analysis und insbesondere dem Hauptsatz der Differential- und Integralrechnung starke Werkzeuge für deren Berechnung gibt.

Wir haben nun alle nötigen Definitionen beisammen, um uns Zufallsvariablen sauber definieren zu können. Wir werden nun sehen, dass Zufallsvariablen eine spezielle Art messbarer Abbildungen sind.

Definition 2.8 (Zufallsvariable)**:** *Gegeben sei ein Wahrscheinlichkeitsraum* $(\Omega, \mathscr{A}, \mathbb{P})$ *und eine Abbildung* $X : (\Omega, \mathscr{A}, \mathbb{P}) \to (\mathbb{R}, \mathscr{B})$*. X heißt reelle Zufallsvariable, falls X Borel-messbar ist. Es soll also*

$$X^{-1}(B) \in \mathscr{A}$$

für alle $B \in \mathscr{B}$ *gelten.*

Wir wollen uns mit der maßtheoretischen Frage der Messbarkeit nicht weiter beschäftigen. Daher verzichten wir im Folgenden auf die explizite Nennung der Start- und Zielräume der Zufallsvariablen.

Nun werden wir eine Möglichkeit finden, wie wir mithilfe von Zufallsvariablen aus alten Wahrscheinlichkeitsmaßen neue bilden können.

Definition 2.9 (Bildmaß)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Durch*

$$\mathbb{P}_X : \mathscr{B} \to [0,1] : B \mapsto \mathbb{P}_X(B) := \mathbb{P}(X^{-1}(B))$$

ist das Bildmaß von $\mathbb{P}$ *unter X definiert.*

Wir können uns nun davon überzeugen, dass das Bildmaß auch tatsächlich ein Wahrscheinlichkeitsmaß ist.

Satz 2.10 (Bildmaß ist Wahrscheinlichkeitsmaß)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Dann ist* $\mathbb{P}_X$ *ein Wahrscheinlichkeitsmaß auf* $(\mathbb{R}, \mathscr{B})$*.*

Beweis :
Wir rechnen die nötigen Eigenschaften nach.

Zur Wohldefiniertheit:
Da X eine messbare Abbildung ist, ist für $B \in \mathscr{B}$ dann $X^{-1}(B) \in \mathscr{A}$. Da $\mathbb{P}$ ein Wahrscheinlichkeitsmaß auf $(\Omega, \mathscr{A})$ ist, können wir dies auf $[0,1]$ abbilden.

Zu (M1):
Betrachte

$$\mathbb{P}_X(\emptyset) = \mathbb{P}(X^{-1}(\emptyset)) \ .$$

Da das Urbild der leeren Menge die leere Menge selbst ist, folgt

$$\begin{aligned} &= \mathbb{P}(\emptyset) \\ &= 0 \ , \end{aligned}$$

da für $\mathbb{P}$ insbesondere (M1) gilt.

Zu (M2):
Sei $B \in \mathscr{B}$ eine Borel-Menge. Betrachte

$$\mathbb{P}_X(B) = \mathbb{P}(X^{-1}(B)) \ .$$

Da X messbar ist, existiert ein $A \in \mathscr{A}$, sodass $X^{-1}(B) = A$ ist. Damit erhalten wir

$$= \mathbb{P}(A) > 0 \ ,$$

da für $\mathbb{P}$ insbesondere (M2) gilt.

Zu (M3):
Seien $(B_n)_{n\in\mathbb{N}_0} \subset \mathscr{B}$ disjunkt. Betrachte

$$\begin{aligned}\mathbb{P}_X\left(\bigsqcup_{n\in\mathbb{N}_0} B_n\right) &= \mathbb{P}\left(X^{-1}\left(\bigsqcup_{n\in\mathbb{N}_0} B_n\right)\right)\\ &= \mathbb{P}\left(\bigsqcup_{n\in\mathbb{N}_0} X^{-1}(B_n)\right) \ .\end{aligned}$$

Da für $\mathbb{P}$ insbesondere (M3) gilt, folgt

$$\begin{aligned}&= \sum_{n\in\mathbb{N}_0} \mathbb{P}(X^{-1}(B_n))\\ &= \sum_{n\in\mathbb{N}_0} \mathbb{P}_X(B_n) \ .\end{aligned}$$

Zur Normiertheit:
Betrachte

$$\mathbb{P}_X(\mathbb{R}) = \mathbb{P}(X^{-1}(\mathbb{R})) \ .$$

Da das Urbild des gesamten Bildraums der gesamte Urbildraum ist, gilt

$$\begin{aligned}&= \mathbb{P}(\Omega)\\ &= 1 \ ,\end{aligned}$$

da $\mathbb{P}$ insbesondere normiert ist.

Somit ist $\mathbb{P}_X$ ein Wahrscheinlichkeitsmaß. □

Da wir nun bewiesen haben, dass $\mathbb{P}_X$ ein Wahrscheinlichkeitsmaß ist, können wir nun den umgangssprachlichen Begriff der „Wahrscheinlichkeit“ mathematisch definieren.

Bemerkung 2.11 (Wahrscheinlichkeit)**:** Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum. Zu einer Zufallsvariable X können wir $\mathbb{P}_X(B)$ als *Wahrscheinlichkeit* für das Ereignis $B \in \mathscr{B}$ interpretieren. Da $\mathbb{P}$ ein Wahrscheinlichkeitsmaß ist, ist $\mathbb{P}_X(A) \in [0,1]$. Dies ist eine direkte Folge aus (M2) und der Normiertheit.

2.2 Dichtefunktionen

Wir wollen uns weiter mit dem oben definierten Bildmaß beschäftigen. Um einfacher damit rechnen zu können, benötigen wir zuerst folgende

Definition 2.12 (Dichtefunktion)**:** *Sei $(\Omega, \mathscr{A}, \mu)$ ein Maßraum und $f : \Omega \to [0,\infty)$ eine messbare Funktion. Zu $A \in \mathscr{A}$ ist durch*

$$\nu(A) := \int \mathbb{1}_A f d\mu \ .$$

ein Maß definiert. Man nennt f dann Dichtefunktion oder kurz Dichte (density) von ν bezüglich μ.

Wenden wir diese Definition jetzt speziell auf Wahrscheinlichkeitsmaße an, so erhalten wir die folgende

Definition 2.13 (Wahrscheinlichkeitsdichte)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Ist $\varphi : \mathbb{R} \to [0, \infty)$ eine messbare Funktion und gilt für alle $B \in \mathscr{B}$*

$$\begin{aligned}\mathbb{P}_X(B) &= \int_{\mathbb{R}} \mathbb{1}_B \varphi \, \mathrm{d}\lambda \\ &= \int_B \varphi(x) \, \mathrm{d}\lambda(x) \ ,\end{aligned}$$

so heißt φ Lebesgue-Borel-Dichte von $\mathbb{P}_X$ oder kurz Lebesgue-Dichte von X. Ist $\varphi : E \to [0, \infty)$ eine messbare Funktion mit $E \subset \mathbb{R}$ höchstens abzählbar und gilt für alle $B \in \mathcal{P}(E)$

$$\begin{aligned}\mathbb{P}_X(B) &= \int_E \mathbb{1}_B \varphi \, \mathrm{d}\# \\ &= \sum_{b \in B} \varphi(b) \ ,\end{aligned}$$

so heißt φ Zähldichte von $\mathbb{P}_X$ oder einfach von X.

Da sich, wie wir noch sehen werden, Zufallsvariablen sehr unterschiedlich verhalten, je nachdem welche Dichte sie besitzen, benötigen wir für diese Bachelorarbeit noch folgende eher untypische

Definition 2.14 (Finite / Diskrete / Stetige Zufallsvariablen)**:** *Sei X eine reelle Zufallsvariable. Wir nennen X finit (finite), falls eine Zähldichte existiert und der Träger dieser Funktion endlich ist. Wir nennen X diskret (discrete), falls eine Zähldichte existiert und der Träger dieser Funktion abzählbar ist. Wir nennen X stetig (continuous), falls eine Lebesgue-Dichte existiert.*

Diese spezielle und eher untypische Wortwahl wird nun näher erläutern.

Bemerkung 2.15 (Wortwahl)**:** Normalerweise werden diskrete Zufallsvariablen als Zufallsvariablen mit Zähldichte definiert, ohne dabei näher auf die Mächtigkeit des Trägers einzugehen. Damit wäre die Menge der finiten Zufallsvariablen in der Menge der diskreten Zufallsvariablen enthalten. Da wir für diese Typen von Zufallsvariablen im Code sehr unterschiedliche Methoden entwickeln müssen, werden wir in dieser Bachelorarbeit diese Mengen scharf unterscheiden und aus der Menge der diskreten Zufallsvariablen alle Finiten rausschneiden.

Wir wollen nun jeweils ein Beispiel für diese Typen von Zufallsvariablen betrachten.

Beispiel 2.16 (Würfelwurf, Münzwurf und Regentropfen)**:**

(i) Wir modellieren mit X den Wurf eines sechsseitigen Würfels. Dann gilt

$$\mathrm{Im}(X) = \{1, 2, 3, 4, 5, 6\} \ .$$

Die Zähldichte ist also für $n \in \{1, \ldots, 6\}$ gegeben durch

$$\varphi(n) = \frac{1}{6} \sum_{i=1}^{6} \mathbb{1}_i(n) \ .$$

Stochastisch ist dies eine Gleichverteilung auf $\{1, \ldots, 6\}$. Der Würfelwurf ist also eine finite, reelle Zufallsvariable.

(ii) Wir werfen eine faire Münze. Die Zufallsvariable X sei die Anzahl der Würfe bis zum ersten Mal Kopf geworfen wird. Es gilt

$$\varphi(n) = 2^{-n}$$

für alle $n \in \mathbb{N}$. Dies ist also eine diskrete, reelle Zufallsvariable.

(iii) Wir betrachten einen ein Meter langen, eindimensionalen Strich und modellieren mit X die Position, auf der ein punktförmiger Regentropfen auftrifft. Die Lebesgue-Dichte ist also für $x \in \mathbb{R}$

$$\varphi(x) = \mathbb{1}_{[0,1]}(x) \ .$$

Stochastisch ist dies eine Gleichverteilung auf $[0, 1]$. Dies ist ein Beispiel für eine stetige, reelle Zufallsvariable.

Wir werden in späteren Kapiteln noch viele weitere Beispiele für die verschiedenen Typen von Zufallsvariablen kennenlernen.

Mit diesen Definitionen können wir beginnen den Code zu erläutern. Wir beginnen damit ein Objekt aus der `RandomVariable`-Klasse zu initialisieren, damit wir später damit weiterarbeiten können.

Code 2.17 (Initialisierung)**:** Wie wir gesehen haben, können wir eine Zufallsvariable durch ihre Dichtefunktionen definieren. Weiter benötigen wir noch den Typ der Zufallsvariable. Hierfür muss man dann `RandomVariableFinite`, `RandomVariableDiscrete` oder `RandomVariableContinuous` aufrufen, um ein entsprechendes Objekt zu erhalten. Die Initialisierung läuft bei allen Typen ähnlich ab.

```
def __init__(self, density, variable, force_density=False):
    self.density = density
    self.variable = variable
    self.force_density = force_density
    if force_density == False:
        self._is_density()
```

Mit dem `density` Argument definieren wir die Dichtefunktion der eingegebenen Zufallsvariable. Um den Nutzer nicht in der Wahl der Variablen einzuschränken, wird über das Attribut `variable` die verwendete Variable übergeben. Je nachdem, in welchem Typen man sich befindet, wird automatisch das Attribut `type` definiert, was dann entsprechende `'f'`, `'d'` oder `'c'` für finite, diskrete oder stetige Zufallsvariablen ist.

Es ist zu beachten, dass stetige und diskrete Zufallsvariablen noch ein `supp` Argument besitzen. Dies ist eine Liste, welche jeweils die Integrations- beziehungsweise Summationsgrenzen der verwendeten Variablen enthält. Vergibt man dieses Attribut nicht, so wird $(-\infty, \infty)$ beziehungsweise $[0, \infty)$ verwendet. Dieses Attribut ist vor allem wichtig für Dichten stetiger Zufallsvariablen, die eine Indikatorfunktion beinhalten, denn SymPy verträgt stückweise Funktionen (piecewise) nicht gut. Möchte man also eine Zufallsvariable mit der Dichte

$$\varphi : [0, \infty) \to \mathbb{R}$$

definieren, so verwendet man `supp = [sym.Integer(0), sym.oo]`.
Um zu überprüfen, ob das so definierte Wahrscheinlichkeitsmaß normiert ist, wird die Methode `is_density()` in der jeweiligen Unterklasse aufgerufen. Um diese verstehen zu können, benötigen wir noch ein wichtiges Werkzeug.

Wir werden nun typische Beispiele für das Erstellen von Zufallsvariablen betrachten.

Beispiel 2.18 (Erstellen von Zufallsvariablen)**:**

(i) Sei $X \sim \mathrm{Ber}(p)$ Bernoulli-verteilt mit Parameter $p \in (0, 1)$. Die Zähldichte ist dann

$$\mathbb{P}(X = 1) = p, \qquad \mathbb{P}(X = 0) = 1 - p \ .$$

Dies ist eine finite Zufallsvariable.

Zur Initialisierung definieren wir

```
n = sym.Symbol('n', integer=True, nonnegative=True)
p = sym.Symbol('p', real=True, positive=True)
density = {1: p, 0: 1 - p}
```

Dies ist ein Dictionary. Die Schlüssel (keys) sind jeweils die Werte der Zufallsvariable und die Werte (values) sind die zugehörige Wahrscheinlichkeiten. Es ist nicht nötig die `0` und die `1` als SymPy Objekte zu definieren, da bei der Initialisierung dieses Objektes die Funktion `_make_density` aufgerufen wird, die die Einträge im Dictionary in SymPy-Objekte verwandelt. Schaden tut es trotzdem nicht. Die Verwendung von SymPy-Objekten kann vor allem nützlich sein, wenn man rationale Werte verwenden möchte, da diese in Python sonst als Kommazahl gerundet werden. Um daraus nun ein `RandomVariable`-Objekt zu machen, definieren wir

```
rv = RandomVariableFinite(density, n)
```

Wir müssen die Variable `n` übergeben, um zu definieren, welche Variable wir für die Integration verwenden wollen.

(ii) Sei $X \sim \mathrm{Bin}(n, p)$ binomialverteilt mit Parameter $n \in \mathbb{N}$ und $p \in (0, 1)$. Die Zähldichte ist für $k \in \{0, \dots, n\}$ dann gegeben durch

$$\varphi(k) = \binom{n}{k} p^k (1-p)^{n-k} \ .$$

Dies ist eine diskrete Zufallsvariable. Zur Initialisierung definieren wir

```
n, k = sym.symbols('n, k', integer=True, nonnegative=True)
p = sym.Symbol('p', real=True, positive=True)
density = sym.binomial(n, k) * p**k * (1 - p)**(n - k)
```

Um daraus nun ein `RandomVariable`-Objekt zu machen, definieren wir

```
rv = RandomVariableDiscrete(density, k)
```

In der Konsole wird der Text `WARNING: Chopped up piecwise-function` ausgegeben. Dies verursacht allgemein keine weiteren Schwierigkeiten. Der Grund ist in diesem Fall, dass die Reihe für $p \in [0, 1]$ vereinfacht werden kann, für $p > 1$ jedoch nicht. Somit bildet SymPy eine stückweise Funktion mit diesen beiden Ästen.
Es sei erwähnt, dass Binomialverteilungen eigentlich keine diskreten Zufallsvariablen sind, da die Werte aus der endlichen Menge $\{0, \dots, n\}$ sind. Da aber

$$\binom{n}{k} = 0$$

ist für $k > n$ und SymPy sehr gut mit dem Binomialkoeffizient rechnen kann, können wir dies so verwenden. Wöllte man dies als echte finite Zufallsvariable definieren, so müsste man die Variable n fest wählen und damit dann ein Dictionary definieren. Man würde also einen Großteil der Flexibilität verlieren. Für $n = 1$ erhält man beispielsweise obige Bernoulli-Verteilung.

(iii) Sei $X \sim \mathrm{N}(0, 1)$ standardnormalverteilt. Die Lebesgue-Dichte ist dann für $x \in \mathbb{R}$

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \ .$$

Dies ist eine stetige Zufallsvariable.

Zur Initialisierung definieren wir

```
x = sym.Symbol('x', real=True)
density = 1 / sym.sqrt(2 * sym.pi) * sym.exp(- x**2 / 2)
```

Um daraus nun ein `RandomVariable`-Objekt zu machen, definieren wir

```
rv = RandomVariableContinuous(density, x)
```

(iv) Sei $X \sim \mathrm{Exp}(\lambda)$ exponentialverteilt mit Parameter $\lambda > 0$. Die Lebesgue-Dichte ist dann für $x \in \mathbb{R}$

$$\varphi(x) = \lambda \exp(-\lambda x) \mathbb{1}_{[0,\infty)}(x) \ .$$

Dies ist eine stetige Zufallsvariable. Zur Initialisierung definieren wir

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
```

Es ist zu beachten, dass wir als Variablennamen für den Parameter nicht `lambda` verwenden können, da dies in Python schon mit den Lambda-Funktionen belegt ist. Weiter sei erwähnt, dass wir hier keine stückweise Funktion mit SymPy bauen, da dies später zu großen Problemen führt. Um zu definieren, dass der Träger nur $[0, \infty)$ ist, benötigen wir bei der Erstellung des Objekts das Argument

```
supp = [sym.Integer(0), sym.oo]
```

Zusammen ergibt sich dann

```
rv = RandomVariableContinuous(density, x, supp=supp)
```

Wir werden später noch viele weitere `RandomVariable`-Objekte definieren.

Man könnte sich die Fragen stellen, wie Dichtefunktionen denn graphisch aussehen. Dazu kann man die folgende Methode verwenden.

Code 2.19 (`plot_density`)**:** Da die Dichten von finiten und diskreten Zufallsvariablen Funktion von diskreten Mengen nach $[0, \infty)$ sind und die Dichten von stetigen Zufallsvariablen von $\mathbb{R}$ nach $[0, \infty)$ abbilden, müssen wir hier die Typen wieder Unterscheiden. Einige Teile des Codes werden für alle Typen verwendet und daher nur für finite Zufallsvariablen beschrieben.

(i) Für finite Zufallsvariablen gilt

```
def plot_density(self, show=True, use_latex=True):
    if self._test_for_symbols():
        return
    x_values = list(self.density.keys())
    y_values = list(self.density.values())
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.scatter(x_values, y_values, marker='.')
    if show:
        plt.show()
    else:
        return fig, ax
```

Die Methode `_test_for_symbols` untersucht eine Dichte, ob mehr Symbole verwendet wurden, als das Symbol für `self.variable`. Solche Dichten können wir nicht plotten, da in diesen Funktionen mindestens ein Parameter zu viel auftaucht.

Die x- und y-Werte für den Plot sind die Keys und Values aus dem Dichte-Dictionary. Diese werden als Punkte mit `scatter` im Plot verteilt. Das Argument `use_latex` verwendet für die Beschriftungen des Plots LaTeX. Mit dem Argument `show` kann man sich entweder den Plot direkt anzeigen lassen oder die beiden matplotlib-Objekte `fig` und `ax` zurückgeben lassen, um noch eigene Änderungen, wie zum Beispiel das Hinzufügen eines Titels, vorzunehmen.

(ii) Für diskrete Zufallsvariablen gilt

```
def plot_density(self, lower=0, upper=10, show=True, use_latex=True):
    if self._test_for_symbols():
        return
    x_values = np.arange(lower, upper + 1, step=1, dtype=int)
    y_values = []
    for x_value in x_values:
        if x_value >= self.supp[0] and x_value <= self.supp[1]:
            y_value = float(self.density.subs(self.variable, x_value).evalf())
        else:
            y_value = 0
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.scatter(x_values, y_values, marker='.')
    if show:
        plt.show()
    else:
        return fig, ax
```

Anders als bei finiten Zufallsvariablen gibt es hier die Argumente `lower` und `upper`, welche die untere beziehungsweise obere Grenze, für die man die Dichte plotten möchte, angeben. Zuerst wird mit NumPy ein Array mit den ganzzahligen Werten zwischen den angegebenen Grenzen erzeugt. Um zu diesen x-Werten die y-Werte zu bestimmen wird mittels einer Schleife zuerst überprüft, ob die Dichte dort überhaupt Werte annimmt oder null ist. Werden nicht-verschwindende Werte angenommen, so wird der x-Wert in die Dichtefunktion eingesetzt, mittels SymPy evaluiert und anschließend in eine Python-Gleitkommazahl umgewandelt. Wie für finite Zufallsvariablen werden diese dann als Punkte im Plot verteilt.

(iii) Für stetige Zufallsvariablen gilt

```
def plot_density(self, lower=-5, upper=5, numpoints=100, show=True, use_latex=True):
    if self._test_for_symbols():
        return
    x_values = np.linspace(lower, upper, num=numpoints)
    y_values = []
    for x_value in x_values:
        if x_value > self.supp[0] and x_value < self.supp[1]:
            y_value = float(self.density.subs(self.variable, x_value).evalf())
        else:
            y_value = 0
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.plot(x_values, y_values)
    if show:
        plt.show()
    else:
        return fig, ax
```

Neben den Argumenten für die obere und untere Grenze gibt es nun das Argument `numpoints`, mit welchem man die Anzahl der berechneten Punkte verändern kann. Die x-Werte werden ähnlich wie bei den diskreten Zufallsvariablen mit NumPy bestimmt. Die Werte der Dichtefunktion werden anschließend wie oben durch Einsetzen bestimmt. Der einzige Unterschied ist, dass sie nicht mehr als Punkte geplottet werden, sondern mit `plot` verbunden werden.

An dieser Stelle sei erwähnt, dass dieser Code nicht sonderlich effizient ist, da die Auswertungen der Dichtefunktionen in SymPy stattfinden und SymPy diese Berechnungen nicht vektorisiert. Es gäbe die Möglichkeit SymPy-Ausdrücke mit `lambdify` in Ausdrücke zu verwandeln, die mit NumPy vektorisiert werden können. Dabei ergibt sich das Problem, dass es Funktionen in SymPy gibt, die in NumPy nicht existieren und dies führt dann zu Fehlern.

Wir können nun versuchen, die Dichten einiger Zufallsvariablen zu visualisieren.

Beispiel 2.20 (Plots von Dichten)**:** Zur Visualisierung verwenden wir die Zufallsvariablen aus obigem Beispiel.

(i) Sei $X \sim \text{Ber}(4/7)$. Wir verwenden

```
n = sym.Symbol('n', integer=True, nonnegative=True)
p = sym.Rational(4, 7)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
rv.plot_density()
```

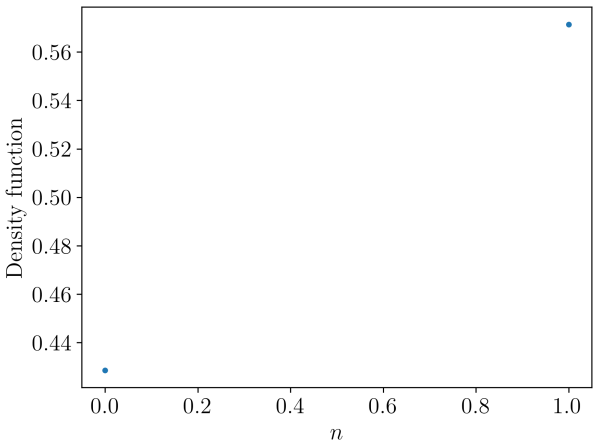

Abb. 1: Dichte einer Ber(4/7)-Verteilung

Wir können diesen Plot auch modifizieren. Mittels

```
fig, ax = rv.plot_density(show=False)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_position('zero')
ax.spines['left'].set_position('zero')
ax.set_ylim(0)
ax.set_xticks([0, 1])
ax.set_yticks([3/7, 4/7], [r'$\frac{3}{7}$', r'$\frac{4}{7}$'])
plt.savefig('Dichte Bernoulli 2.png', dpi=300)
```

erhalten wir

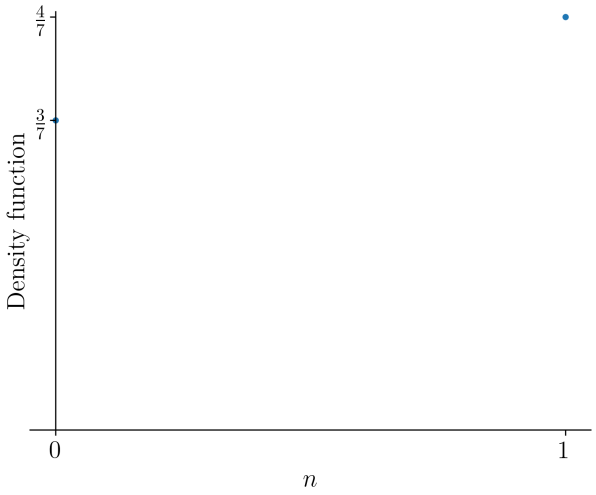

Abb. 2: Dichte einer Ber(4/7)-Verteilung

(ii) Sei $X \sim \mathrm{Bin}(8, 2/3)$. Wir verwenden

```
k = sym.symbols('k', integer=True, nonnegative=True)
n = sym.Integer(8)
p = sym.Rational(2, 3)
density = sym.binomial(n, k) * p**k * (1 - p)**(n - k)
rv = RandomVariableDiscrete(density, k)
rv.plot_density()
```

und erhalten

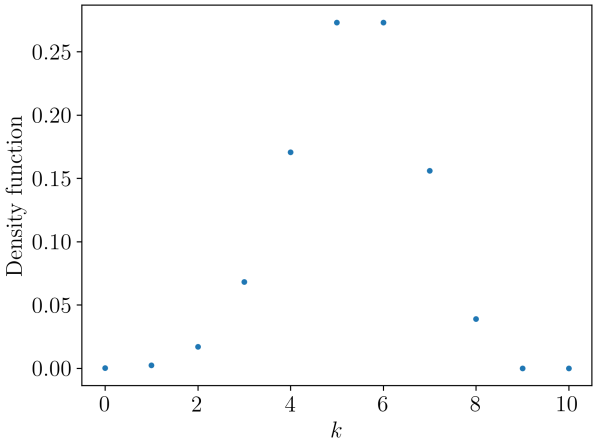

Abb. 3: Dichte einer $\mathrm{Bin}(8, 2/3)$-Verteilung

Dieser Plot ließe sich auch entsprechend verändern, worauf wir an dieser Stelle aber verzichten wollen.

(iii) Sei $X \sim \mathrm{N}(0, 1)$. Wir verwenden

```
mu = sym.Integer(0)
sigma = sym.Integer(1)
x = sym.symbols('x', real=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
rv.plot_density()
```

und erhalten

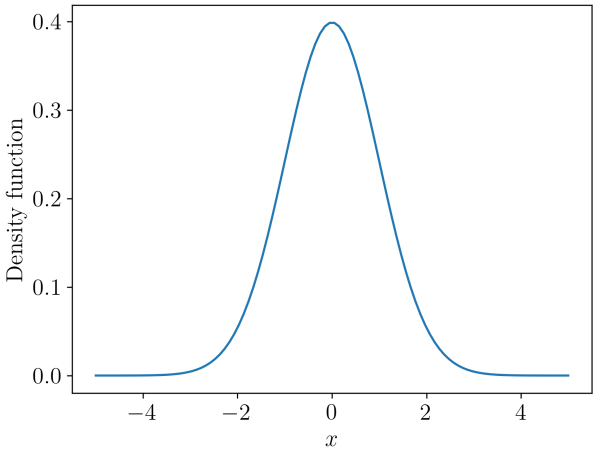

Abb. 4: Dichte einer $\mathrm{N}(0,1)$-Verteilung

(iv) Sei $X \sim \mathrm{Exp}(3)$. Diesen Plot wollen wir gleich etwas bearbeiten.

```
lamda = sym.Integer(3)
x = sym.symbols('x', real=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
fig, ax = rv.plot_density(numpoints=1000, show=False)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_position('zero')
ax.spines['left'].set_position('zero')
ax.set_xlim(-1, 4)
ax.set_ylim(0)
plt.show()
```

Wir erhalten

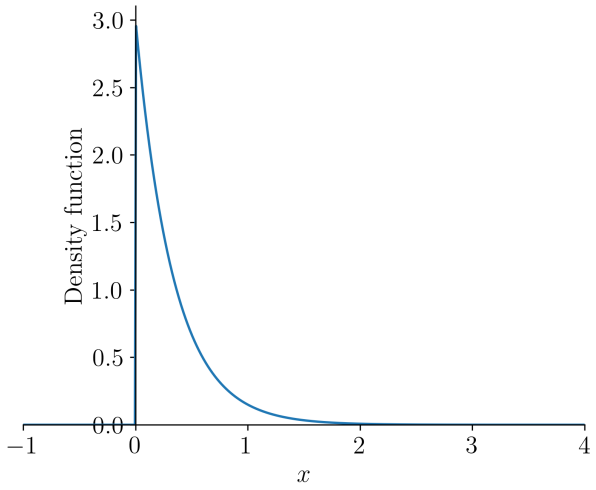

Abb. 5: Dichte einer $\mathrm{Exp}(3)$-Verteilung

An dieser Stelle sei erwähnt, dass die Senkrechte um die Null mathematisch nicht schön ist. Die Dichte macht an dieser Stelle einen Sprung und diese Unterscheidung wäre in matplotlib theoretisch möglich, hätte aber zu einer unnötigen Spaghettifizierung des Codes geführt.

Wir werden nun einen Satz betrachten und beweisen, mit dem wir Funktionen bezüglich des Bildmaßes integrieren können.

Satz 2.21 (Maß und Bildmaß)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable. Weiter sei* $f : \mathbb{R} \to \mathbb{R}$ *eine messbare Funktion, so dass* $f \circ X : \Omega \to \mathbb{R}$ $\mathbb{P}$*-integrierbar ist. Dann gilt*

$$\int f \, \mathrm{d}\mathbb{P}_X = \int f \circ X \, \mathrm{d}\mathbb{P} \ .$$

Beweis :
Wir beweisen diesen Satz mittels algebraischer Induktion. Sei $B \in \mathscr{B}$ und $\mathbb{1}_B$ die zugehörige Indikatorfunktion. Betrachte zunächst für $\omega \in \Omega$

$$\mathbb{1}_B \circ X(\omega) = 1 \Leftrightarrow X(\omega) \in B \Leftrightarrow \omega \in X^{-1}(B) \ .$$

Also gilt

$$\mathbb{1}_B \circ X = \mathbb{1}_{X^{-1}(B)} \ .$$

Betrachte nun

$$\begin{aligned}
\int \mathbb{1}_B \, \mathrm{d}\mathbb{P}_X &= \int_B \mathrm{d}\mathbb{P}_X \\
&= \mathbb{P}_X(B) \\
&= \mathbb{P}(X^{-1}(B)) \\
&= \int_{X^{-1}(B)} \mathrm{d}\mathbb{P} \\
&= \int \mathbb{1}_{X^{-1}(B)} \, \mathrm{d}\mathbb{P} \\
&= \int \mathbb{1}_B \circ X \, \mathrm{d}\mathbb{P} \ .
\end{aligned}$$

Sei nun f eine Linearkombination von Indikatorfunktionen mit $(B_n)_{n\in\mathbb{N}_0} \subset \mathscr{B}$ und $(a_n)_{n\in\mathbb{N}_0} \subset \mathbb{R}$. Dann gilt dank Linearität des Integrals

$$\int \sum_{n\in\mathbb{N}_0} a_n \mathbb{1}_{B_n} \, \mathrm{d}\mathbb{P}_X = \sum_{n\in\mathbb{N}_0} a_n \int \mathbb{1}_{B_n} \, \mathrm{d}\mathbb{P}_X \ .$$

Da dies nur Indikatorfunktionen sind, gilt nach dem ersten Schritt

$$\begin{aligned}
&= \sum_{n\in\mathbb{N}_0} a_n \int \mathbb{1}_{B_n} \circ X \, \mathrm{d}\mathbb{P} \\
&= \int \sum_{n\in\mathbb{N}_0} a_n \mathbb{1}_{B_n} \circ X \, \mathrm{d}\mathbb{P} \ .
\end{aligned}$$

Sei f nun eine positive, messbare Funktion. Dann existiert eine monoton steigende Folge $(f_n)_{n\in\mathbb{N}_0}$ von Linearkombination von Indikatorfunktionen mit $\lim_{n\to\infty} f_n = f$. Dann gilt

$$\int f \, \mathrm{d}\mathbb{P}_X = \int \lim_{n\to\infty} f_n \, \mathrm{d}\mathbb{P}_X \ .$$

Dank dem Satz von Lebesgue lassen sich Limes und Integration vertauschen und es gilt

$$= \lim_{n\to\infty} \int f_n \, \mathrm{d}\mathbb{P}_X \ .$$

Da dies Linearkombination von Indikatorfunktionen sind, gilt nach obigem

$$= \lim_{n\to\infty} \int f_n \circ X \, \mathrm{d}\mathbb{P} \ .$$

Nochmaliges Anwenden des Satzes von Lebesgue liefert

$$\begin{aligned} &= \int \lim_{n\to\infty} f_n \circ X \,\mathrm{d}\mathbb{P} \\ &= \int f \circ X \,\mathrm{d}\mathbb{P} \ . \end{aligned}$$

Sei nun f nur noch eine messbare Funktion. Dann gibt es positive, messbare Funktion f^+ und f^- mit $f = f^+ - f^+$ und es gilt

$$\begin{aligned} \int f \,\mathrm{d}\mathbb{P}_X &= \int f^+ - f^+ \,\mathrm{d}\mathbb{P}_X \\ &= \int f^+ \,\mathrm{d}\mathbb{P}_X - \int f^- \,\mathrm{d}\mathbb{P}_X \ . \end{aligned}$$

Da dies positive messbare Funktionen sind, folgt dank obigem

$$\begin{aligned} &= \int f^+ \circ X \,\mathrm{d}\mathbb{P} - \int f^- \circ X \,\mathrm{d}\mathbb{P} \\ &= \int f^+ \circ X - f^- \circ X \,\mathrm{d}\mathbb{P} \\ &= \int (f^+ - f^+) \circ X \,\mathrm{d}\mathbb{P} \\ &= \int f \circ X \,\mathrm{d}\mathbb{P} \ . \end{aligned}$$

Die Aussage gilt somit für alle messbaren Funktionen und wir sind fertig. □

Nun werden wir einen Satz beweisen, der es uns erlaubt Funktionen, die wir bezüglich des Bildmaßes integrieren wollen auf einem einfacheren Wege mithilfe der Dichtefunktion zu integrieren.

Satz 2.22 (Dichtesatz)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable mit Lebesgue-Dichte φ. Weiter sei $f : \mathbb{R} \to \mathbb{R}$ eine Funktion, sodass $f \cdot \varphi$ λ-integrierbar ist. Dann gilt*

$$\int f \,\mathrm{d}\mathbb{P}_X = \int f\varphi \,\mathrm{d}\lambda \ .$$

Eine analoge Aussage gilt für das Zählmaß.

Beweis :
Wir beweisen diesen Satz mittels algebraischer Induktion. Sei $B \in \mathscr{B}$ und $\mathbb{1}_B$ die zugehörige Indikatorfunktion. Betrachte

$$\int \mathbb{1}_B \,\mathrm{d}\mathbb{P}_X = \mathbb{P}_X(B) \ .$$

Da φ eine λ-Dichte des Maßes $\mathbb{P}_X$ ist, folgt direkt aus der Definition der Dichtefunktion

$$= \int \mathbb{1}_B \varphi \,\mathrm{d}\lambda \ .$$

Sei nun f eine Linearkombination von Indikatorfunktionen mit $(B_n)_{n\in\mathbb{N}_0} \subset \mathscr{B}$ und $(a_n)_{n\in\mathbb{N}_0} \subset \mathbb{R}$. Dann gilt dank Linearität des Integrals

$$\int \sum_{n\in\mathbb{N}_0} a_n \mathbb{1}_{B_n} \,\mathrm{d}\mathbb{P}_X = \sum_{n\in\mathbb{N}_0} a_n \int \mathbb{1}_{B_n} \,\mathrm{d}\mathbb{P}_X \ .$$

Da dies nur Indikatorfunktionen sind, gilt nach dem ersten Schritt

$$\begin{aligned}
&= \sum_{n\in\mathbb{N}_0} a_n \int \mathbb{1}_{B_n}\varphi \,\mathrm{d}\lambda \\
&= \int \sum_{n\in\mathbb{N}_0} a_n \mathbb{1}_{B_n}\varphi \,\mathrm{d}\lambda \\
&= \int \left(\sum_{n\in\mathbb{N}_0} a_n \mathbb{1}_{B_n}\right)\varphi \,\mathrm{d}\lambda \ .
\end{aligned}$$

Sei f nun eine positive, messbare Funktion. Dann existiert eine monoton steigende Folge $(f_n)_{n\in\mathbb{N}_0}$ von Linearkombinationen von Indikatorfunktionen mit $\lim_{n\to\infty} f_n = f$. Dann gilt

$$\int f \,\mathrm{d}\mathbb{P}_x = \int \lim_{n\to\infty} f_n \,\mathrm{d}\mathbb{P}_X \ .$$

Dank dem Satz von Lebesgue lassen sich Limes und Integration vertauschen und es gilt

$$= \lim_{n\to\infty} \int f_n \,\mathrm{d}\mathbb{P}_X \ .$$

Da dies Linearkombinationen von Indikatorfunktionen sind, gilt nach obigem

$$= \lim_{n\to\infty} \int f_n\varphi \,\mathrm{d}\lambda \ .$$

Nochmaliges Anwenden des Satzes von Lebesgue liefert

$$\begin{aligned}
&= \int \lim_{n\to\infty} f_n\varphi \,\mathrm{d}\lambda \\
&= \int f\varphi \,\mathrm{d}\lambda \ .
\end{aligned}$$

Sei nun f nur noch eine messbare Funktion. Dann gibt es positive, messbare Funktion f^+ und f^- mit $f = f^+ - f^+$ und es gilt

$$\begin{aligned}
\int f \,\mathrm{d}\mathbb{P}_X &= \int f^+ - f^- \,\mathrm{d}\mathbb{P}_X \\
&= \int f^+ \,\mathrm{d}\mathbb{P}_X - \int f^- \,\mathrm{d}\mathbb{P}_X \ .
\end{aligned}$$

Da dies positive messbare Funktionen sind, folgt dank obigem

$$\begin{aligned}
&= \int f^+\varphi \,\mathrm{d}\lambda - \int f^-\varphi \,\mathrm{d}\lambda \\
&= \int f^+\varphi - f^-\varphi \,\mathrm{d}\lambda \\
&= \int \left(f^+ - f^-\right)\varphi \,\mathrm{d}\lambda \\
&= \int f\varphi \,\mathrm{d}\lambda \ .
\end{aligned}$$

Die Aussage gilt somit für alle messbaren Funktionen und wir sind fertig. □

Die Kombination dieser beiden Sätze liefert das folgende Korollar. Dies liefert uns eine einfach Möglichkeit Maßintegrale mit bekannten Werkzeugen zu Riemann- beziehungsweise Lebesgue-Integralen und Reihen zu berechnen.

Korollar 2.23 (Dichtekorollar)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit Lebesgue-Dichte* φ*. Weiter sei* $f : \mathbb{R} \to \mathbb{R}$ *eine messbare Funktion, so dass* $f \circ X : \Omega \to \mathbb{R}$ $\mathbb{P}$*-integrierbar und* $f \cdot \varphi$ λ*-integrierbar ist. Dann gilt*

$$\int f \circ X \, \mathrm{d}\mathbb{P} = \int f\varphi \, \mathrm{d}\lambda \ .$$

Eine analoge Aussage gilt für das Zählmaß.

Beweis :
Wir verbinden die Aussagen der vorangegangenen Sätze. Betrachte mit dem Satz von Maß und Bildmaß

$$\begin{aligned} \int f \circ X \, \mathrm{d}\mathbb{P} &= \int f \, \mathrm{d}\mathbb{P}_X \\ &= \int f\varphi \, \mathrm{d}\lambda \ , \end{aligned}$$

dank dem gerade bewiesenen Dichtesatz. □

Bemerkung 2.24 (Dichtekorollar Summenform)**:** Wir können das obige Dichtkorollar in die Form für diskrete Zufallsvariablen schreiben. Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine diskrete Zufallsvariable mit Zähldichte φ. Dann gilt für $f : \mathbb{N}_0 \to \mathbb{R}$

$$\begin{aligned} \int f \circ X \, \mathrm{d}\mathbb{P} &= \int f\varphi \, \mathrm{d}\# \\ &= \sum_{n \in \mathbb{N}_0} f(n)\varphi(n) \ . \end{aligned}$$

Analog gilt dies auch für finite Zufallsvariablen.

Sei X nun eine stetige Zufallsvariable mit Dichte φ. Ist $f\varphi$ für stetige Zufallsvariablen sogar Riemann-integrierbar, so können wir die Integrale wie gewohnt mit dem Hauptsatz berechnen. Diese Voraussetzung nehmen wir im Folgenden häufig implizit an.

Wollen wir also die Funktion einer Zufallsvariable bezüglich des Wahrscheinlichkeitsmaßes $\mathbb{P}$ berechnen, so genügt es entsprechende Lebesgue-Integral oder die Summe zu berechnen, falls diese existiert. Wir können nun eine Methode definieren, die genau dieses Korollar anwendet.

Code 2.25 (`integrate_random_variable`)**:** Ziel dieser Methode ist es Integrale von $f(X)$ zu bestimmen. Sei X eine reelle Zufallsvariable mit Dichte φ und f eine Funktion, die wir mit `expr` bezeichnen. Da sich das Zähl- und Lebesgue-Maß grundlegend in ihrer Berechnung unterscheiden, ist diese Methode für jeden Typ von Zufallsvariable separat definiert.

(i) Für finite Zufallsvariablen gilt

```
def integrate_random_variable(self, expr, lower=-sym.oo, upper=sym.oo):
    integral = sym.Integer(0)
    for key in self.density.keys():
        if key >= lower and key <= upper:
            integral += expr.subs(self.variable, key) * self.density[key]
    integral = sym.simplify(integral)
    return integral
```

Wir lassen hier untere und obere Grenzen für die Summation zu, welche standardmäßig $-\infty$ und ∞ sind. Da die Dichte einer finiten Zufallsvariable aus einem Dictionary besteht, können wir über deren Schlüssel iterieren. Ist der Schlüssel innerhalb des Integrationsintervalls, ersetzen wir mithilfe der `subs`-Methode in jeder Iteration die Variable der Funktion durch den entsprechenden Schlüssel und multiplizieren dies mit der Wahrscheinlichkeit diesen Wert zu erhalten, was `self.density[key]` entspricht. Zuletzt lassen wir SymPy diesen Ausdruck vereinfachen und anschließend ausgeben.

(ii) Für diskrete Zufallsvariablen gilt

```
def integrate_random_variable(self, expr, lower=sym.Integer(0), upper=sym.oo):
    lower = sym.Max(lower, self.supp[0])
    upper = sym.Min(upper, self.supp[1])
    integral = sym.summation(expr * self.density, (self.variable, lower, upper)).doit()
    integral = RandomVariable.clean_piecewise(integral)
    integral = sym.simplify(integral)
    return integral
```

Wir lassen hier untere und obere Grenzen für die Summation zu, welche standardmäßig 0 und ∞ sind. Diese Summationsgrenzen werden dann noch mit den Grenzen für die Werte der Zufallsvariable verarbeitet. Mithilfe von SymPy bilden wir die Summe aus Funktion und Dichte und summieren von der unteren bis zur oberen Grenze. Als nächstes wird aus der Hauptklasse die `clean_piecewise`-Funktion aufgerufen. Diese überprüft einen Ausdruck, ob dieser zur SymPy `Piecewise`-Klasse gehört. Dies ist das SymPy Objekt für stückweise definiert Funktionen. Ist der Ausdruck aus dieser Klasse, so nimmt die Funktion immer den ersten Zweig dieser Funktion, da dieser meist das gesuchte Ergebnis ist und gibt eine Warnung in der Konsole aus. Sollte man nicht wollen, dass stückweise Funktionen gekürzt werden, so kann man die `no_chopping`-Funktion aufrufen. Es ist zu beachten, dass dann die meisten Funktionen nicht mehr funktionieren, da SymPy mit stückweisen Funktionen nicht richtig weiterrechnen kann. Zum Schluss wird wieder versucht das Ergebnis zu vereinfachen.

(iii) Für stetige Zufallsvariablen gilt

```
def integrate_random_variable(self, expr, lower=-sym.oo, upper=sym.oo):
    lower = sym.Max(lower, self.supp[0])
    upper = sym.Min(upper, self.supp[1])
    integral = sym.integrate(expr * self.density, (self.variable, lower, upper)).doit()
    integral = RandomVariable.clean_piecewise(integral)
    integral = sym.simplify(integral)
    return integral
```

Wir lassen auch hier obere und untere Grenzen zu, welche standardmäßig auf ∞ und $-\infty$ gesetzt werden. Da wir für die Dichten von stetigen Zufallsvariablen die Indikatorfunktionen weggelassen haben, müssen wir diese in einem ersten Schritt mit dem Träger der Dichte vergleichen. Anschließend integrieren wir mit SymPy das Produkt aus Funktion und Dichte von der unteren zur oberen Grenze. Nach der Entfernung von stückweisen Funktionen wird dies wieder vereinfacht und ausgegeben.

Der besonders elegante Teil besteht darin, dass wir in der Aufforderung für die Berechnung quasi die maßtheoretische Integration einer Zufallsvariable verwenden können, da der Code das Dichtekorollar automatisch anwendet und die Dichtefunktion dazu multipliziert.

Wir können nun versuchen mit der obigen Methode eine Funktion einer Zufallsvariable zu integrieren.

Beispiel 2.26 (Funktionen von Zufallsvariablen)**:**

(i) Sei X eine finite Zufallsvariable mit

$$\begin{aligned}\mathbb{P}(X=1) &= p\\ \mathbb{P}(X=2) &= q\\ \mathbb{P}(X=3) &= 1-p-q\end{aligned}$$

mit $p, q \in (0,1)$. Wir berechnen nun mit $E = \{1,2,3\}$

$$\begin{aligned}\int X^2 + 2X \,\mathrm{d}\mathbb{P} &= \sum_{n\in E} (n^2+2n)\mathbb{P}(X=n)\\ &= (1^2+2\cdot 1)\mathbb{P}(X=1) + (2^2+2\cdot 2)\mathbb{P}(X=2) + (3^2+2\cdot 3)\mathbb{P}(X=3)\\ &= (1+2)p + (4+4)q + (9+6)(1-p-q)\\ &= 3p + 8q + 15 - 15p - 15q\\ &= 15 - 12p - 7q\end{aligned}$$

Dies können wir nun folgendermaßen mit der oben definierten Funktion berechnen lassen.

```
n = sym.Symbol('n', integer=True, positive=True)
p, q = sym.symbols('p, q', real=True, positive=True)
density = {1: p, 2: q, 3: 1 - p - q}
rv = RandomVariableFinite(density, n)
polynom = rv.integrate_random_variable(n**2 + 2 * n)
```

Als Ergebnis erhalten wir `-12*p - 7*q + 15`.

(ii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Betrachte nun mit dem Dichtekorollar

$$\begin{aligned}\int \sin(X) \,\mathrm{d}\mathbb{P} &= \int_{-\infty}^{\infty} \sin(x)\lambda \exp(-\lambda x)\mathbb{1}_{[0,\infty)}(x)\,\mathrm{d}x\\ &= \lambda \int_0^{\infty} \sin(x)\exp(-\lambda x)\,\mathrm{d}x\ .\end{aligned}$$

Weiter betrachten wir das Integral

$$\alpha = \int_0^{\infty} \sin(x)\exp(-\lambda x)\,\mathrm{d}x$$

Mit partieller Integration erhalten wir

$$\begin{aligned}&= \left[\frac{1}{\lambda}\sin(x)\exp(-\lambda x)\right]_0^{\infty} + \frac{1}{\lambda}\int_0^{\infty}\cos(x)\exp(-\lambda x)\,\mathrm{d}x\\ &= [0-0] + \frac{1}{\lambda}\int_0^{\infty}\cos(x)\exp(-\lambda x)\,\mathrm{d}x\\ &= \frac{1}{\lambda}\int_0^{\infty}\cos(x)\exp(-\lambda x)\,\mathrm{d}x\ .\end{aligned}$$

Nochmal partielle Integration liefert

$$\begin{aligned}
&= \left[-\frac{1}{\lambda^2}\cos\exp(-\lambda x)\right]_0^\infty - \frac{1}{\lambda^2}\int_0^\infty \sin(x)\exp(-\lambda x)\,\mathrm{d}x \\
&= \left[-0 + \frac{1}{\lambda^2}\right] - \frac{1}{\lambda^2}\int_0^\infty \sin(x)\exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{1}{\lambda^2} - \frac{1}{\lambda^2}\alpha\ .
\end{aligned}$$

Zusammengefasst ist also

$$\alpha = \frac{1}{\lambda^2} - \frac{1}{\lambda^2}\alpha\ .$$

Umformen liefert

$$\begin{aligned}
(\lambda^2+1)\alpha &= 1 \\
\alpha &= \frac{1}{\lambda^2+1}\ .
\end{aligned}$$

Damit ist dann insgesamt

$$\int \sin(X)\,\mathrm{d}\mathbb{P} = \frac{\lambda}{\lambda^2+1}\ .$$

Dies können wir folgendermaßen berechnen lassen

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
sine = rv.integrate_random_variable(sym.sin(x))
```

Wir erhalten analog `lambda/(lambda**2 + 1)`.

(iii) Sei $X \sim \mathrm{N}(0,1)$ standardnormalverteilt. Wir können mit dieser Methode beispielsweise

$$\int \log(X)\,\mathrm{d}\mathbb{P}$$

berechnen. Es ist zu beachten, dass log den natürlichen Logarithmus bezeichnet.

```
x = sym.Symbol('x', real=True)
density = 1 / sym.sqrt(2 * sym.pi) * sym.exp(- x**2 / 2)
rv = RandomVariableContinuous(density, x)
natural_logarithm = rv.integrate_random_variable(sym.log(x))
```

Wir erhalten

$$\begin{aligned}
\int \log(X)\,\mathrm{d}\mathbb{P} &= -\frac{\log(2)}{2} - \frac{\gamma}{2} + \frac{i\pi}{2} \\
&= \frac{1}{2}\left(-\log(2) - \gamma + i\pi\right)\ .
\end{aligned}$$

Dies ist ein sehr interessantes Ergebnis, da hier einige wichtige mathematische Konstanten auftauchen. Die eulersche Zahl e implizit im natürliche Logarithmus von 2, die Euler–Mascheroni-Konstante γ, die komplexe Einheit i und die Kreiszahl π. Dies können wir leider nicht auf elementare Weise nachrechnen.

Wir werden später in vielen Funktionen diese `integrate_random_variable`-Methode benötigen.

Mit dieser Integrationsmethode können wir nun die `_is_density`-Methode definieren, welche bei der Initialisierung einer Zufallsvariable aufgerufen wird.

Code 2.27 (`_is_density`)**:** Mit dieser Methode überprüfen wir, ob die Wahrscheinlichkeit des gesamten Raumes gleich eins ist. Dies ist folgendermaßen implementiert

```
def _is_density(self):
    total = self.integrate_random_variable(sym.Integer(1))
    if not total.equals(sym.Integer(1)):
        print('WARNING: Density not standardized!')
    return total
```

Mithilfe der gerade definieren Integrationsmethode wird über die Einsfunktion integriert. Wir bilden also im stetigen Fall

$$\begin{aligned}\int_{\mathbb{R}} 1\varphi(x)\,\mathrm{d}x &= \int_{\mathbb{R}} \varphi(x)\,\mathrm{d}x \\ &= \mathbb{P}_X(\mathbb{R})\ .\end{aligned}$$

Für ein Wahrscheinlichkeitsmaß war dies nach Definition eins. Streng genommen müssten wir noch nachrechnen, dass die Dichtefunktion überall nicht-negativ ist. Hierfür gibt es leider keine einfache und effiziente Methode, weshalb diese Überprüfung nicht stattfindet. Diese Methode dient allgemein nur dazu, dem Nutzer offensichtliche Fehler bei der Eingabe der Dichte mitzuteilen. Ist die Dichte nicht normiert, so wird eine entsprechende Warnung in der Konsole ausgegeben. Sollte man diese Überprüfung unterdrücken wollen, so muss man bei der Definition der Zufallsvariable zusätzlich `force_denisty=True` übergeben, was die gesamte Überprüfung überspringt. Hier wurde bewusst kein Python-Fehler verwendet. Dieser würde das Programm sofort beenden und es gibt leider Zufallsvariablen, deren Dichten SymPy nicht zu eins vereinfachen kann. Sei dazu $X \sim \mathrm{Log}(p)$ mit $p \in (0,1)$ logarithmischverteilt. Die Dichte ist gegeben durch

$$\varphi(n) = -\frac{p^n}{n}\frac{1}{\log(1-p)}$$

für alle $n \in \mathbb{N}$. Wir können daraus nun ein `RandomVariable`-Objekt machen.

```
p = sym.Symbol('p', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = - p**n / n * 1 / sym.log(1 - p)
rv = RandomVariableDiscrete(density, n, supp=[sym.Integer(1), sym.oo])
```

Wir erhalten nun die Meldung `WARNING: Density not standardized!`. Dies liegt vermutlich daran, dass SymPy die Summe nicht zusammenfassen möchte, da wir nicht festlegen können, dass $p < 1$ ist.

Da uns Dichtefunktionen die Arbeit erleichtern, wollen wir nun allgemein wissen, wann eine solche Funktion existiert. Für diesen zentralen Satz der Maßtheorie benötigen wir zunächst die folgende

Definition 2.28 (Nullmenge)**:** *Sei $(\Omega, \mathscr{A}, \mu)$ ein Maßraum und $A \in \mathscr{A}$ eine Menge. Gilt*

$$\mu(A) = 0\ ,$$

so ist A eine μ-Nullmenge oder nur Nullmenge.

Nun können wir einige typische Beispiele für Nullmengen betrachten.

Beispiel 2.29 (Nullmenge)**:**

(i) Betrachten wir $(\Omega, \mathscr{A}, \delta_{\{\omega\}})$ mit einem $\omega \in \Omega$, so ist jede Menge $A \in \mathscr{A}$ mit $\omega \notin A$ eine $\delta_{\{\omega\}}$-Nullmenge.

(ii) Betrachten wir $(\mathbb{R}, \mathscr{B}, \lambda)$, so ist jede Einpunktmenge eine Nullmenge. Sei dazu $x \in \mathbb{R}$. Dann gilt

$$\begin{aligned}\lambda(\{x\}) &= \lambda([x,x]) \\ &= x - x \\ &= 0 \ .\end{aligned}$$

(iii) Betrachte wieder $(\mathbb{R}, \mathscr{B}, \lambda)$. Dann ist jede abzählbare Menge eine Nullmenge. Sei nun $A = \{x_n \in \Omega \mid n \in \mathbb{N}_0\}$. Dann gilt

$$\lambda(A) = \lambda\left(\bigsqcup_{n\in\mathbb{N}_0} \{x_n\}\right) \ .$$

Aufgrund der σ-Additivität gilt

$$= \sum_{n\in\mathbb{N}_0} \lambda(\{x_n\}) \ .$$

Nach (ii) sind dies Nullmenge, womit gilt

$$\begin{aligned}&= \sum_{n\in\mathbb{N}_0} 0 \\ &= 0 \ .\end{aligned}$$

Insbesondere sind $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{Q}$ Lebesgue-Nullmengen.

Definition 2.30 (Absolute Stetigkeit)**:** *Seien μ und ν Maße auf $(\Omega, \mathscr{A})$. Gilt für alle $A \in \mathscr{A}$ mit $\mu(A) = 0$ auch $\nu(A) = 0$, so ist ν absolutstetig bezüglich μ und wir schreiben $\nu \ll \mu$.*

Mithilfe dieser Definitionen können wir nun den folgenden Satz formulieren, der uns dabei hilft, verstehen zu können, wann Dichten existieren.

Satz 2.31 (Radon-Nikodym)**:** *Seien μ und ν zwei σ-endliche Maße auf $(\Omega, \mathscr{A})$. Dann sind die folgenden Aussagen äquivalent.*

(i) *ν ist absolutstetig bezüglich μ.*

(ii) *ν hat eine Dichte bezüglich μ.*

Dies ist ein zentraler Satz der Maß- und Wahrscheinlichkeitstheorie. Der Satz von Radon-Nikodym ist nicht nur eine Existenz-Aussage sondern er liefert uns auch eine „Bauanleitung“, wie wir eine solche Dichte finden können. Darauf wollen wir aber im Folgenden nicht weiter eingehen.

Nun können wir einige Beispiele zur Anwendung des Satzes von Radon-Nikodym betrachten.

Beispiel 2.32 (Anwendungen Radon-Nikodym)**:**

(i) Gegeben sei die Zufallsvariable X aus dem Regenbeispiel. Die Lebesgue-Dichte ist

$$\varphi(x) = \mathbb{1}_{[0,1]}(x) \ .$$

Nach dem Satz von Radon-Nikodym ist das Bildmaß $\mathbb{P}_X$ absolutstetig bezüglich dem Lebesgue-Maß λ und alle Lebesgue-Nullmengen sind auch Nullmengen bezüglich dem Bildmaß $\mathbb{P}_X$. Wir können also folgern, dass die Wahrscheinlichkeit, dass der Regentropfen auf einem bestimmten Punkt auftritt, null ist.

(ii) Gegeben sei ein Dirac-Maß zu $x \in \mathbb{R}$. Nach dem Beispiel zu Nullmengen ist $\{x\}$ eine Lebesgue-Nullmenge. Da aber $\delta_{\{x\}}(\{x\}) = 1$ ist, ist $\delta_{\{x\}}$ nicht absolutstetig bezüglich λ. Aus dem Satz von Radon-Nikodym folgt, dass das Dirac-Maß keine Lebesgue-Dichte besitzen kann.

(iii) Sei $X \sim \text{Ber}(p)$ mit $p \in (0,1)$. Da

$$\mathbb{P}_X(1) = p \neq 0$$

ist, kann X nach dem Satz von Radon-Nikodym keine Lebesgue-Dichte besitzen. X hat dafür eine Zähldichte.

Es gibt auch Zufallsvariablen, die weder eine Lebesgue- noch eine Zähldichte besitzen.

Im Folgenden werden wir uns ausschließlich mit Zufallsvariablen beschäftigen, die eine Zähl- oder Lebesgue-Dichte besitzen.

2.3 Verteilungsfunktionen

In diesem Abschnitt werden wir uns mit einer wichtigen Funktion beschäftigen, die für jede reelle Zufallsvariable existiert.

Definition 2.33 (Verteilungsfunktion)**:** *Sei X eine reelle Zufallsvariable. Durch*

$$F_X : \mathbb{R} \to [0,1] : x \mapsto F_X(x) := \mathbb{P}(X \leq x)$$

ist die Verteilungsfunktion definiert.

Wir werden nun einige Eigenschaften von Verteilungsfunktionen beweisen.

Satz 2.34 (Eigenschaften Verteilungsfunktion)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Die Verteilungsfunktion F_X hat dann die folgenden Eigenschaften.*

(i) *F_X ist rechtsseitig stetig.*

(ii) *F_X ist monoton wachsend.*

(iii) $\lim_{x \to -\infty} F_X(x) = 0$.

(iv) $\lim_{x \to \infty} F_X(x) = 1$.

Beweis :
Zu (i):
Sei $x \in \mathbb{R}$ beliebig und $(x_n)_{n\in\mathbb{N}_0}$ eine monoton fallende Folge mit Grenzwert x. Betrachte

$$\lim_{n\to\infty} F_X(x) = \lim_{n\to\infty} \mathbb{P}(X \leq x_n) \ .$$

Da $\mathbb{P}$ als Wahrscheinlichkeitsmaß insbesondere stetig von oben ist, kann man den Limes reinziehen und erhält

$$\begin{aligned} &= \mathbb{P}(X \leq x) \\ &= F_X(x) \ . \end{aligned}$$

Somit ist F_X rechtsseitig stetig.

Zu (ii):
Sei $x_1 \leq x_2$ aus $\mathbb{R}$. Betrachte

$$\begin{aligned} F_X(x_2) - F_X(x_1) &= \mathbb{P}(X \leq x_2) - \mathbb{P}(X \leq x_1) \\ &= \int_{\mathbb{R}} \mathbb{1}_{(-\infty,x_2]} \, \mathrm{d}\mathbb{P}_X - \int_{\mathbb{R}} \mathbb{1}_{(-\infty,x_2]} \, \mathrm{d}\mathbb{P}_X \ . \end{aligned}$$

Mit der Linearität des Integrals gilt

$$= \int_{\mathbb{R}} \mathbb{1}_{(-\infty,x_2]} - \mathbb{1}_{(-\infty,x_1]} \, \mathrm{d}\mathbb{P}_X \ .$$

Wir werden gleich sehen, dass folgenden Umformung eine Identität ist

$$\begin{aligned} &= \int_{\mathbb{R}} \mathbb{1}_{[x_1,x_2]} \, \mathrm{d}\mathbb{P}_X \\ &= \mathbb{P}(X \in [x_1, x_2]) \geq 0 \ , \end{aligned}$$

dank (M2). Umformen liefert

$$F_X(x_2) \geq F_X(x_1) \ .$$

Zum Beweis der oben verwendeten Identität, betrachte die folgende Tabelle:

	$\mathbb{1}_{(-\infty,x_1]}(x)$	$\mathbb{1}_{(-\infty,x_2]}(x)$	$\mathbb{1}_{(-\infty,x_2]}(x) - \mathbb{1}_{(-\infty,x_1]}(x)$	$\mathbb{1}_{[x_1,x_2]}(x)$
$x \in (-\infty, x_1]$	1	1	0	0
$x \in (x_1, x_2)$	0	1	1	1
$x \in [x_2, \infty)$	0	0	0	0

Da die letzten beiden Spalten immer übereinstimmen gilt für alle $x \in \mathbb{R}$

$$\mathbb{1}_{(-\infty,x_2]}(x) - \mathbb{1}_{(-\infty,x_1]}(x) = \mathbb{1}_{[x_1,x_2]}(x) \ .$$

Zu (iii):
Betrachte

$$\begin{aligned} \lim_{x\to-\infty} F_X(x) &= \lim_{x\to-\infty} \mathbb{P}(X \leq x) \\ &= \lim_{x\to-\infty} \int \mathbb{1}_{(-\infty,x]} \, \mathrm{d}\mathbb{P}_X \ . \end{aligned}$$

Mit dem Satz von Lebesgue können wir Integration und Limesbildung vertauschen und es gilt

$$\begin{aligned} &= \int \lim_{x\to-\infty} \mathbb{1}_{(-\infty,x]} \,\mathrm{d}\mathbb{P}_X \\ &= \int \mathbb{1}_{\emptyset} \,\mathrm{d}\mathbb{P}_X \\ &= \mathbb{P}_X(\emptyset) = 0 \ , \end{aligned}$$

da $\mathbb{P}_X$ als Maß insbesondere nulltreu ist.

Zu (iv):
Betrachte

$$\begin{aligned} \lim_{x\to\infty} F_X(x) &= \lim_{x\to\infty} \mathbb{P}(X \le x) \\ &= \lim_{x\to\infty} \int \mathbb{1}_{(-\infty,x]} \,\mathrm{d}\mathbb{P}_X \ . \end{aligned}$$

Mit dem Satz von Lebesgue können wir Integration und Limesbildung vertauschen und es gilt

$$\begin{aligned} &= \int \lim_{x\to\infty} \mathbb{1}_{(-\infty,x]} \,\mathrm{d}\mathbb{P}_X \\ &= \int \mathbb{1}_{\mathbb{R}} \,\mathrm{d}\mathbb{P}_X \\ &= \mathbb{P}_X(\mathbb{R}) = 1 \ , \end{aligned}$$

da $\mathbb{P}_X$ ein Wahrscheinlichkeitsmaß ist. □

Wir werden nun eine Möglichkeit finden, diese Verteilungsfunktion auf einfache Weise zu berechnen.

Bemerkung 2.35 (Berechnung der Verteilungsfunktion)**:** Sei X eine stetige Zufallsvariable mit Dichte φ. Dann gilt dank dem Dichtekorollar

$$\begin{aligned} F_X(x) &= \mathbb{P}(X \le x) \\ &= \int \mathbb{1}_{\{\omega : X(\omega) \le x\}}(\omega) \,\mathrm{d}\mathbb{P}(\omega) \\ &= \int \mathbb{1}_{X\in(-\infty,x]} \,\mathrm{d}\mathbb{P} \\ &= \int \mathbb{1}_{(-\infty,x]} \varphi \,\mathrm{d}x \\ &= \int_{-\infty}^{x} \varphi \,\mathrm{d}x \ . \end{aligned}$$

Wir können also einfach die Einsfunktion bis zum Wert x integrieren. Für diskrete Zufallsvariablen berechnen wir

$$F_X(x) = \sum_{n=0}^{\lfloor x \rfloor} \varphi(n) \ .$$

Es sei an dieser Stelle nochmal erwähnt, dass die Verteilungsfunktion eine Funktion von $\mathbb{R}$ nach $[0,1]$ ist. Wir können also jeden reellen Wert einsetzen und nicht nur Werte, die X annehmen kann. Deshalb müssen wir hier die Abrundungsfunktion $\lfloor\cdot\rfloor$ verwenden.

Wir können nun eine Methode schreiben, welche zu einer Zufallsvariable die Verteilungsfunktion berechnet.

Code 2.36 (`distribution_function`)**:** Die Berechnung der Verteilungsfunktion ist leider etwas komplizierter. Aufgrund dessen müssen wir hier wieder die verschiedenen Typen unterscheiden.

(i) Für finite Zufallsvariablen gilt

```
def distribution_function(self):
    sortable = True
    keys = list(self.density.keys())
    for key in keys:
        if isinstance(key, sym.Symbol):
            print('WARNING: Can\'t sort values.')
            sortable = False
            break
    if sortable:
        keys = sorted(keys)
    cumulative_probability = self.density[keys[0]]
    distribution_function = {keys[0]: cumulative_probability}
    keys.pop(0)
    for key in keys:
        cumulative_probability += self.density[key]
        distribution_function.update({key: cumulative_probability})
    return distribution_function
```

Im ersten Schritt wird überprüft, ob die Werte der Zufallsvariable, also die Keys des Dictionaries, sortierbar sind. Sind sie nicht sortierbar, so wird die Verteilungsfunktion aufgrund der gegebenen Reihenfolge berechnet. Nun wird die erste kumulierte Wahrscheinlichkeit festgelegt und diese mit dem entsprechenden Wert der Zufallsvariable der Verteilungsfunktion hinzugefügt. Zuletzt wird dieser Wert aus der Liste der Werte entfernt. Im nächsten Schritt wird über alle anderen Werte iteriert. Dazu wird die Wahrscheinlichkeit für den entsprechenden Wert der kumulierte Wahrscheinlichkeit hinzuaddiert und das Dictionary um das entsprechende Paar erweitert. Man erhält am Schluss also ein Dictionary. Die Keys sind die ersten Werte, ab denen die sich im Value befindende kumulierte Wahrscheinlichkeit angenommen wird. Dies sind also jeweils nach links geschlossene halboffene Intervalle. Insbesondere ist auf dem offenen Intervall von $-\infty$ bis zum ersten Wert die kumulierte Wahrscheinlichkeit (implizit) null.

(ii) Für diskrete Zufallsvariablen gilt

```
def distribution_function(self, value=None):
    if value == None:
        t = sym.Symbol('t', real=True)
        upper = sym.Min(self.supp[1], sym.floor(t))
        distribution_function = self.integrate_random_variable(sym.Integer(1), upper=upper
)
        return distribution_function
    else:
        value = sym.sympify(value)
        upper = sym.Min(self.supp[1], sym.floor(value))
        distribution_function = self.integrate_random_variable(sym.Integer(1), upper=upper
)
        distribution_function = float(distribution_function.evalf())
        return distribution_function
```

Die Berechnung der Verteilungsfunktion für diskrete Zufallsvariablen läuft über zwei verschiedene Arten ab. Ruft man die Methode ohne weiteres Argument auf, so ergibt sich die obere Grenze über die Abrundungsfunktion $\lfloor t \rfloor$ und das Minimum mit der oberen Schranke für den Wertebereich. Anschließend wird über die Eins summiert bis zu dieser oberen Grenze.

Verwendet man als Argument eine Zahl, so wird diese ebenfalls abgerundet und das Minimum mit der oberen Grenze gebildet. Auch hier wird die Summe über die Eins gebildet. Anders als zuvor wird dieser Wert nun evaluiert und in eine Gleitkommazahl umgewandelt. Diese zweite Methode wird später für den Plot der Verteilungsfunktion und die Simulation wichtig.

(iii) Für stetige Zufallsvariablen gilt

```
def distribution_function(self):
    t = sym.Symbol('t', real=True)
    distribution_function = self.integrate_random_variable(sym.Integer(1), upper=t)
    return distribution_function
```

Diese Berechnung ist viel näher an der Definition der Verteilungsfunktion. Dabei wird die Einsfunktion mithilfe der Integrationsmethode integriert, wobei die obere Grenze die Variable der Verteilungsfunktion ist.

Wir können nun einige Verteilungsfunktionen berechnen.

Beispiel 2.37 (Verteilungsfunktionen)**:** Da diskrete Zufallsvariablen Treppenfunktionen als Verteilungsfunktionen haben und sich die Summen nicht weiter zusammenfassen lassen, werden wir nur finite und stetige Beispiele betrachten.

(i) Sei $X \sim \mathrm{Ber}(p)$ mit $p \in (0,1)$ Bernoulli-verteilt. Wir finden die folgende Verteilungsfunktion

$$F_X(x) = \begin{cases} 0 & , x \in (-\infty, 0) \\ 1-p & , x \in [0,1) \\ 1 & , x \in [1, \infty) \end{cases} .$$

Mittels

```
n = sym.Symbol('n', integer=True,positive=True)
p = sym.Symbol('p', real=True, positive=True)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
distribution_function = rv.distribution_function()
```

erhalten wir `{0: 1 - p, 1: 1}`. Dies entspricht genau dem oben Berechneten.

(ii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Für $x \geq 0$ betrachten wir deshalb

$$\begin{aligned} F_X(x) &= \mathbb{P}(X \leq x) \\ &= \int_{-\infty}^{x} \lambda \exp(-\lambda x) \mathbb{1}_{[0,\infty)}(x) \, \mathrm{d}x \\ &= \int_{0}^{x} \lambda \exp(-\lambda x) \, \mathrm{d}x \\ &= [-\exp(-\lambda x)]_0^x \\ &= -\exp(-\lambda x) + 1 \ . \end{aligned}$$

Nach den Eigenschaften der Verteilungsfunktion ist für $x < 0$ dann $F_X(x) = 0$. Um den oberen Teil der Verteilungsfunkion mit SymPy auszurechnen, verwenden wir

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
distribution_function = rv.distribution_function()
```

und erhalten `1 - exp(-lambda*t)` wie oben.

(iii) Sei X auf $[0,1]$ stetig gleichverteilt. Betrachte zu $x \in [0,1]$

$$\begin{aligned} F_X(x) &= \mathbb{P}(X \leq x) \\ &= \int_{-\infty}^{x} \mathbb{1}_{[0,1]}(x)\,\mathrm{d}x \\ &= \int_{0}^{x} 1\,\mathrm{d}x \\ &= [x]_0^x \\ &= x\ . \end{aligned}$$

Nach den Eigenschaften der Verteilungsfunktion ist $F_X(x) = 0$ für $x < 0$ und $F(x) = 1$ für $x > 1$. Mittels

```
x = sym.Symbol('x', real=True)
density = sym.Integer(1)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.Integer(1)])
distribution_function = rv.distribution_function()
```

erhalten wir `Min(1, t)`, was gleichbedeutend ist mit Obigem. Wir können SymPy bei der Integration leider nicht mitteilen, dass die obere Grenze kleiner eins ist, weshalb es zu diesem Minimum kommt.

Vor allem diese letzte Verteilungsfunktion wird bei der Simulation noch sehr wichtig werden. Zu den anderen Beispielen werden wir die Verteilungsfunktion gleich noch visualisieren.

Wie bei den Dichtefunktionen könnte man sich hier natürlich auch eine Visualisierung wünschen. Diese ist im Folgenden beschrieben.

Code 2.38 (`plot_distribution_function`)**:** Wie bei `plot_density` müssen wir hier die verschiedenen Typen von Zufallsvariablen unterscheiden. Teile der Methoden wurden dort schon erläutert, weshalb diese hier nicht wiederholt werden.

(i) Für finite Zufallsvariablen gilt

```
def plot_distribution_function(self, show=True, use_latex=True):
    if self._test_for_symbols():
        return
    distribution_function = self.distribution_function()
    keys = list(distribution_function.keys())
    min_value = min(keys)
    max_value = max(keys)
    distance = max_value - min_value
    lower = min_value - 0.1 * distance
    upper = max_value + 0.1 * distance
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Distribution function')
    ax.hlines(y=0, xmin=lower, xmax=min_value, color='tab:blue', linewidth=2)
    ax.hlines(y=1, xmin=max_value, xmax=upper, color='tab:blue',linewidth=2)
    for num, key in enumerate(keys):
        if num < len(keys) - 1:
            ax.hlines(y=distribution_function[key], xmin=keys[num], xmax=keys[num+1],
    color='tab:blue', linewidth=2)
    if show:
        plt.show()
    else:
        return fig, ax
```

Zuerst lassen wir mit der gerade definierten `distribution_function`-Methode die Verteilungsfunktion berechnen. Zu den x-Werten der Sprungstellen berechnen wir nun Minimum, Maximum sowie deren Abstand. Um die linke und rechte Grenze des Plots zu bestimmen, geben wir jeweils 10% der Länge als Puffer dazu. Als nächstes bilden wir zwei horizontale Linien auf den Höhen Null und Eins. Wir iterieren nun über alle restlichen Werte und fügen die entsprechenden Horizontalen ein.

(ii) Für diskrete Zufallsvariablen gilt

```
def plot_distribution_function(self, lower=0, upper=10, show=True, use_latex=True):
    if self._test_for_symbols():
        return
    lower = int(np.floor(lower))
    upper = int(np.ceil(upper))
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Distribution function')
    for num in range(lower, upper + 1):
        propability = self.distribution_function(value=num)
        ax.hlines(y=propability, xmin=num - 1, xmax=num, color='tab:blue', linewidth=2)
    if show:
        plt.show()
    else:
        return fig, ax
```

Da wir bei dieser Methode eine untere und obere Grenze selbst wählen können, müssen wir in einem ersten Schritt diese entsprechend runden, um später sinnvoll iterieren zu können. Standardmäßig wird die Verteilungsfunktion auf dem Intervall $[0, 10]$ geplottet. Wir iterieren über alle ganzen Zahlen von der unteren Grenze bis zur oberen Grenze eingeschlossen. Die kumulierte Wahrscheinlichkeit erhalten wir von der `distribution_function`-Methode im numerischen Modus. Damit können wir dann die passenden Horizontalen plotten.

(iii) Für stetige Zufallsvariablen gilt

```
def plot_distribution_function(self, lower=-5, upper=5, numpoints=100, show=True,
    use_latex=True):
    if self._test_for_symbols():
        return
    distribution_function = self.distribution_function()
    t = sym.Symbol('t', real=True)
    x_values = np.linspace(lower, upper, num=numpoints)
    y_values = []
    for x_value in x_values:
        if x_value < self.supp[0]:
            y_value = 0
        elif x_value > self.supp[1]:
            y_value = 1
        else:
            y_value = float(distribution_function.subs(t, x_value).evalf())
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Distribution function')
    ax.plot(x_values, y_values)
    if show:
        plt.show()
    else:
        return fig, ax
```

Als erstes lassen wir die Verteilungsfunktion berechnen. Anschließen bestimmen wir mit NumPy äquidistante x-Werte zwischen der unteren und oberen Grenze. Diese lassen sich beliebig ändern. Standardmäßig wird das Intervall $[-5, 5]$ geplottet. Um die kumulierte Wahrscheinlichkeit zu bestimmen iterieren wir über alle x-Werte. Zuerst wird überprüft, ob dieser Wert kleiner ist als der Träger. In diesem Fall verschwindet die kumulierte Wahrscheinlichkeit. Ist der Wert größer als der Träger, so ist diese eins. Ansonsten setzen wir in die Verteilungsfunktion den x-Wert ein und werten diese dort aus. Zum Schluss plotten wir unsere so berechneten x- und y-Werte.

An dieser Stelle sei noch darauf hingewiesen, dass die Verteilungsfunktion strenggenommen rechtsseitig stetig ist und wir dies an den Anfängen und Enden der Linien entsprechend kennzeichnen könnten. Darauf wurde verzichtet, um die Plots übersichtlicher zu gestalten.

Im folgenden Beispiel werden wir uns zu den Beispielen, für die wir bereits die Dichtefunktion visualisiert haben, auch die Verteilungsfunktion plotten.

Beispiel 2.39 (Plots von Verteilungsfunktionen)**:** Auf Änderungen am Plot wollen wir an dieser Stelle verzichten. Wir fügen `rv.plot_distribution_function` an die entsprechenden Codeschnipsel an.

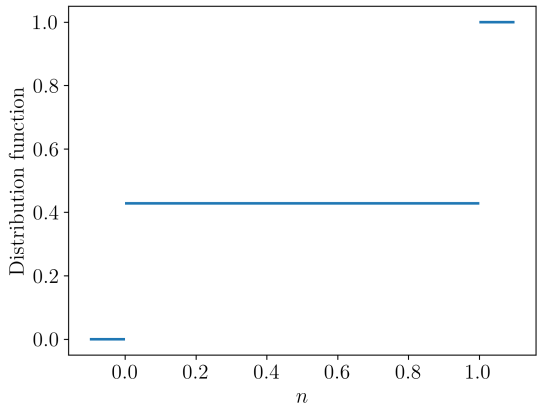

Abb. 6: Verteilungsfunktion einer Ber(4/7)-Verteilung

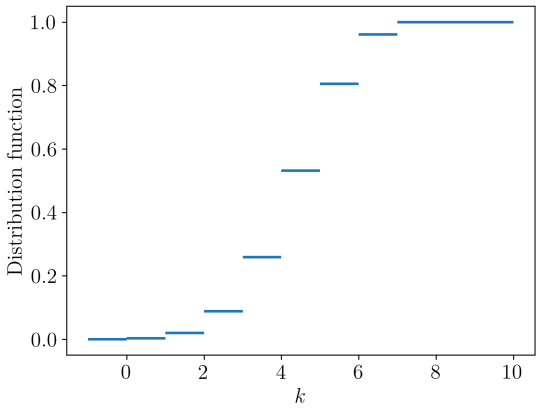

Abb. 7: Verteilungsfunktion einer Bin(8, 2/3)-Verteilung

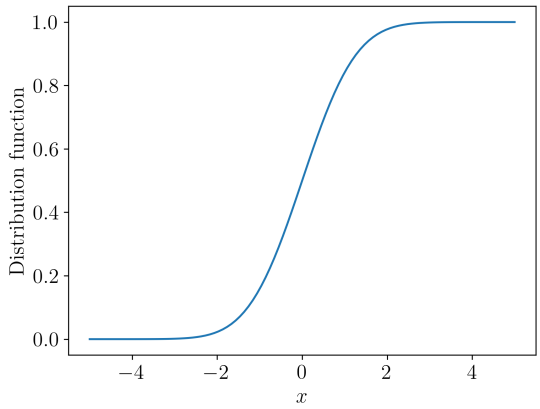

Abb. 8: Verteilungsfunktion einer N(0, 1)-Verteilung

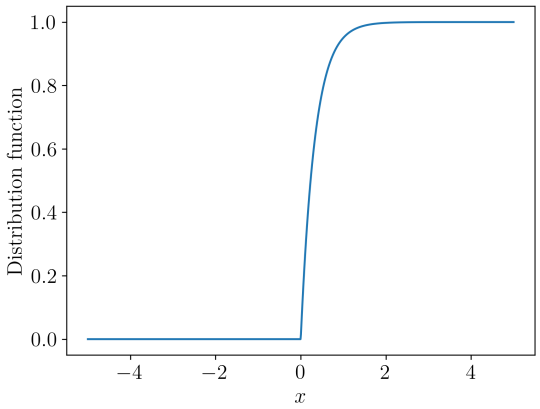

Abb. 9: Verteilungsfunktion einer Exp(3)-Verteilung

Wir sehen bei allen Verteilungsfunktionen, dass sie nur Werte aus $[0, 1]$ annehmen und insbesondere die Grenzwerte 0 und 1 für $x \to -\infty$ und $x \to \infty$ haben. Außerdem sehen wir, dass Verteilungsfunktionen monoton wachsend sind. Bei der Bernoulli- und Binomialverteilung sehen wir deren stückweise stetige Natur. Die stetigen Verteilungen sind sogar stetig. Dies sind genau die Aussagen, die wir im Satz über die Eigenschaften der Verteilungsfunktion gezeigt haben.

3 Momente

3.1 Erwartungswerte

Interpretiert man eine Zufallsvariable als ein Experiment mit einem zufälligen Ausgang, so könnte man sich überlegen, welche Methoden sich zur Berechnung des Erwartungswertes anbieten.

Beispiel 3.1 (Erwartungswert Würfel)**:** Wir betrachten wieder das Würfelbeispiel. Naiv würde man berechnen, dass man mit einer Wahrscheinlichkeit von 1/6 die Zahl Eins würfelt, mit 1/6 die Zwei und so weiter. Wir berechnen also

$$1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5 \ .$$

Ist $E = \{1, \dots, 6\}$ unser Ergebnisraum, so berechnen wir

$$\mathbb{E}(X) = \sum_{n \in E} n \mathbb{P}(X = n) \ .$$

Vergleichen wir dies mit unserem Dichtekorollar, so könnte man auf die Idee kommen, als zu integrierende Funktion die Identität zu wählen.

Dieses Beispiel erheben wir nun zur

Definition 3.2 (Erwartungswert)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Der Erwartungswert von X (mean) ist definiert durch*

$$\mathbb{E}(X) := \int X \, \mathrm{d}\mathbb{P} \ .$$

Dieses Maßintegral können wir so nicht direkt berechnen.

Bemerkung 3.3 (Berechnung des Erwartungswertes)**:** Wenden wir nun das Dichtekorollar an, so erhalten wir für finite und diskrete Zufallsvariable

$$\mathbb{E}(X) = \sum_{n \in \mathbb{N}_0} n \varphi(n) \ .$$

Für stetige Zufallsvariable erhalten wir ähnlich

$$\mathbb{E}(X) = \int_{\mathbb{R}} x \varphi(x) \, \mathrm{d}x \ .$$

Auf diese Weise werden wir den Erwartungswert berechnen.

Mit dieser neuen Definition können wir das Regenbeispiel betrachten.

Beispiel 3.4 (Erwartungswert Regen)**:** Sei X so verteilt, wie im Regenbeispiel. Wir können nun berechnen, wo der Regentropfen im Mittel auftrifft. Betrachte also

$$\begin{aligned} \mathbb{E}(X) &= \int_{\mathbb{R}} x \mathbb{1}_{[0,1]}(x) \, \mathrm{d}x \\ &= \int_0^1 x \, \mathrm{d}x \\ &= \left[\frac{x^2}{2} \right]_0^1 \\ &= \frac{1}{2} \ . \end{aligned}$$

Der Regentropfen kommt also im Mittel in der Mitte auf. Dies entspricht auch unserer Erwartung.

Auf die Implementierung wollen wir im Moment noch verzichten, da wir im folgenden Kapitel eine allgemeinere Methode zur Momentebestimmung definieren werden. Außerdem werden wir in einem späteren Kapitel noch eine zweite Methode zur Berechnung der Momente finden.

Beispiel 3.5 (Interpretation Erwartungswert)**:** Wie wir gerade bemerkt haben, scheint der Erwartungswert den durchschnittlichen Wert einer Zufallsvariable zu beschreiben. Den Erwartungswert können wir beispielsweise in der Dichtefunktion visualisieren. Hierfür verwenden wir die Beispiele, für die wir die Dichtefunktion bereits visualisiert haben.

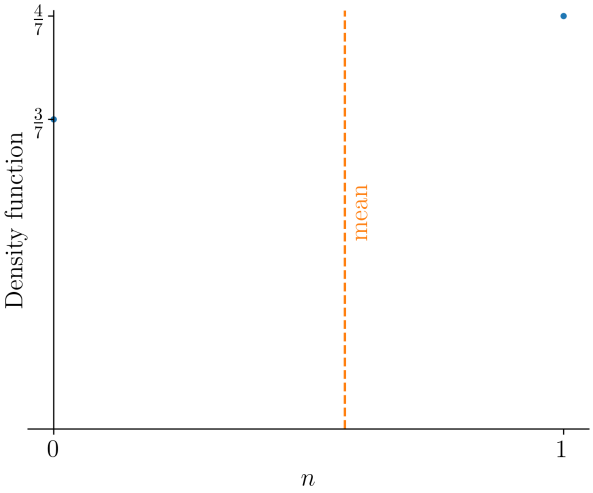

Abb. 10: Dichte einer Ber(4/7)-Verteilung

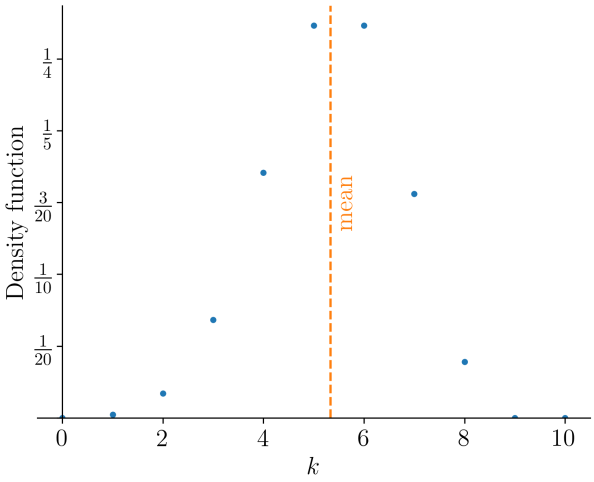

Abb. 11: Dichte einer Bin(8, 2/3)-Verteilung

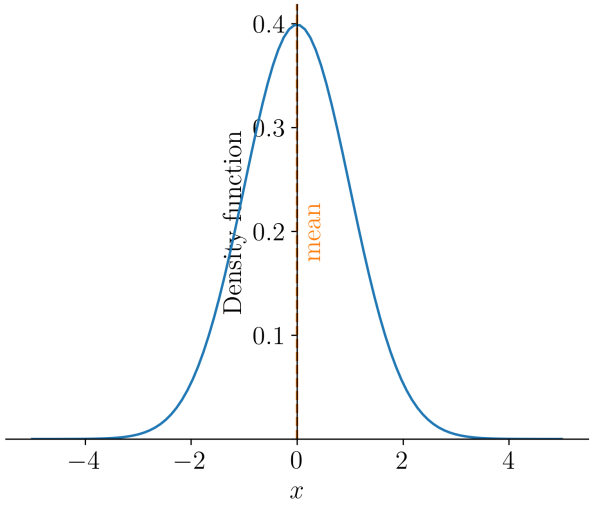

Abb. 12: Dichte einer N(0, 1)-Verteilung

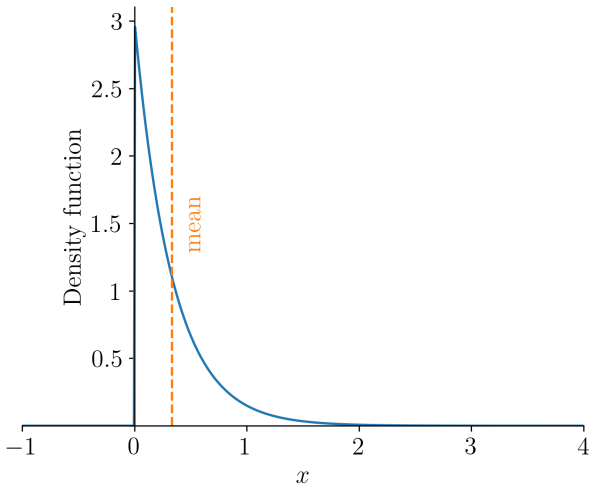

Abb. 13: Dichte einer Exp(3)-Verteilung

Wir sehen, dass der Erwartungswert tatsächlich die Verteilung teilt. Bei der finiten und diskreten Verteilung fällt auf, dass der Erwartungswert nicht unbedingt ein Wert sein muss, den die Zufallsvariable tatsächlich annimmt. Hierzu könnte man den Median (median) oder Modus (mode) berechnen.

Da die Dichtefunktionen von Normalverteilungen symmetrisch um den Erwartungswert sind, wollen wir diese noch gesondert betrachten.

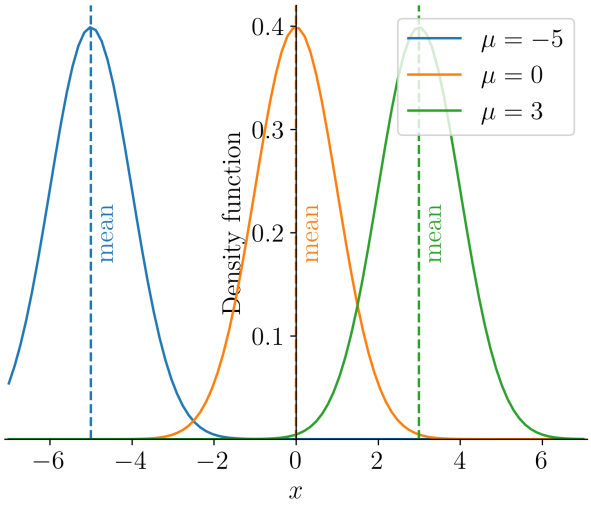

Abb. 14: Dichte einer $\mathrm{N}(\mu, 1)$-Verteilung

Wir sehen also, dass Symmetriezentrum und Erwartungswert bei diesen Normalverteilungen zusammenfallen. Diese Aussage gilt sogar allgemein.

Satz 3.6 (Erwartungswert symmetrischer Verteilungen)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine integrierbare Zufallsvariable, sodass die Dichte symmetrisch um* μ *ist. Dann gilt*

$$\mathbb{E}(X) = \mu \ .$$

Beweis :
Aufgrund der Symmetrie gilt

$$f(\mu + x) = f(\mu - x)$$

für alle $x \in \mathbb{R}$. Nach Definition gilt

$$\begin{aligned} \mathbb{E}(X) &= \int X \, \mathrm{d}\mathbb{P} \\ &= \int_{\mathbb{R}} x f(x) \, \mathrm{d}x \ . \end{aligned}$$

Aufgrund der Integrierbarkeit können wir dieses Integral am Symmetriezentrum trennen und es gilt

$$= \int_{-\infty}^{\mu} x f(x) \, \mathrm{d}x + \int_{\mu}^{\infty} x f(x) \, \mathrm{d}x \ .$$

Wir werden nun diese beiden Integral getrennt voneinander betrachten. Betrachte zunächst

$$\begin{aligned} \int_{-\infty}^{\mu} x f(x) \, \mathrm{d}x &= \int_{-\infty}^{\mu} (\mu - (\mu - x)) f(x) \, \mathrm{d}x \\ &= \int_{-\infty}^{\mu} \mu f(x) - (\mu - x) f(x) \, \mathrm{d}x \ . \end{aligned}$$

Dank der Linearität folgt

$$= \mu \int_{-\infty}^{\mu} f(x) \, \mathrm{d}x - \int_{-\infty}^{\mu} (\mu - x) f(x) \, \mathrm{d}x \ .$$

Das Integral im vorderen Summanden können wir auch mittels der Verteilungsfunktion F_X ausdrücken

$$= \mu F_X(\mu) - \int_{-\infty}^{\mu} (\mu - x) f(x)\,\mathrm{d}x \ .$$

Auf eine ähnliche Weise betrachten wir das andere Integral

$$\begin{aligned}\int_{\mu}^{\infty} x f(x)\,\mathrm{d}x &= \int_{\mu}^{\infty} (\mu + (x-\mu)) f(x)\,\mathrm{d}x \\ &= \int_{\mu}^{\infty} \mu f(x) + (x-\mu) f(x)\,\mathrm{d}x \ .\end{aligned}$$

Mithilfe der Linearität folgt wieder

$$= \mu \int_{\mu}^{\infty} f(x)\,\mathrm{d}x + \int_{\mu}^{\infty} (x-\mu) f(x)\,\mathrm{d}x \ .$$

Im vorderen Integral finden wir die Gegenwahrscheinlichkeit zu vorigem Ereigniss. Wir können dies also umschreiben zu

$$= \mu(1 - F_X(\mu)) + \int_{\mu}^{\infty} (x-\mu) f(x)\,\mathrm{d}x \ .$$

Setzen wir diese Rechnung nun in die Berechnung des Erwartungswertes ein, so erhalten wir

$$\begin{aligned}\mathbb{E}(X) &= \mu F_X(\mu) - \int_{-\infty}^{\mu} (\mu - x) f(x)\,\mathrm{d}x + \mu(1 - F_X(\mu)) + \int_{\mu}^{\infty} (x-\mu) f(x)\,\mathrm{d}x \\ &= \mu - \int_{-\infty}^{\mu} (\mu - x) f(x)\,\mathrm{d}x + \int_{\mu}^{\infty} (x-\mu) f(x)\,\mathrm{d}x \ .\end{aligned}$$

Aufgrund der Symmetrie der Dichtefunktion sind die beiden Integrale gleich und es gilt

$$= \mu \ .$$

Wir können diesen Satz auch auf eine deutlich elegantere und kürzere Art beweisen. Da X symmetrisch bezüglich μ ist, haben $X - \mu$ und $\mu - X$ dieselbe Verteilung. Es gilt also

$$\mathbb{E}(X - \mu) = \mathbb{E}(\mu + X) \ .$$

Damit gilt

$$\begin{aligned}0 &= \mathbb{E}(X - \mu) - \mathbb{E}(\mu - X) \\ &= \mathbb{E}(X - \mu - \mu + X) \\ &= \mathbb{E}(2X - 2\mu) \\ &= 2\mathbb{E}(X) - 2\mu\end{aligned}$$

und

$$\mathbb{E}(X) = \mu$$

durch Umformen. □

3.2 Höhere Momente

Wir können die Definition des Erwartungswertes nun verallgemeinern.

Definition 3.7 (n-tes Moment)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Zu $n \in \mathbb{N}$ definiert man das n-te Moment von X (moment) durch*

$$\mathbb{E}(X^n) := \int X^n \, \mathrm{d}\mathbb{P} \ .$$

Auf die Bemerkung zur Berechnung wollen wir an dieser Stelle verzichten. Wir werden dies sofort programmieren.

Code 3.8 (`_moment_integration`)**:** Wollen wir die Berechnung des n-ten Moments in Python implementieren, so müssen wir zuerst das Dichtekorollar anwenden und erhalten ein analoges Ergebnis zum Erwartungswert. Wir müssen nur X durch X^n beziehungsweise x durch x^n ersetzen. Da wir für die verschiedenen Typen jeweils eine Integrationsmethode definiert haben, ist die Berechnung des n-ten Moments sehr einfach.

```
def _moment_integration(self, n):
    moment = self.integrate_random_variable(self.variable**n)
    return moment
```

Diese Methode geht zurück in die jeweilige Unterklasse und integriert beziehungsweise summiert dann die n-te Potenz der Variable und multipliziert dies anschließend mit der Dichte. Wie definiert, wird dies dann entsprechend vereinfacht und aufgeräumt. An dieser Stelle sieht man wunderschön die Eleganz, die wir durch die klasseninterne `integrate_random_variable`-Methode gewonnen haben.

Diese Methode können wir gleich ausprobieren.

Beispiel 3.9 (n-tes Moment Regen)**:** Wir wollen nun für das Regenbeispiel das n-te Moment berechnen. Betrachte

$$\begin{aligned}
\mathbb{E}(X^n) &= \int_{\mathbb{R}} x^n \mathbb{1}_{[0,1]}(x) \, \mathrm{d}x \\
&= \int_0^1 x^n \, \mathrm{d}x \\
&= \left[\frac{x^{n+1}}{n+1} \right]_0^1 \\
&= \frac{1}{n+1} - 0 \\
&= \frac{1}{n+1} \ .
\end{aligned}$$

Für $n = 1$ erhalten wir den zuvor berechneten Erwartungswert von $1/2$. Wir können nun unserem Ergebnis überprüfen

```
x = sym.Symbol('x', real=True)
n = sym.Symbol('n', integer=True, positive=True)
density = sym.Integer(1)
rv = RandomVariableContinuous(density, x, supp=[sym.Integer(0), sym.Integer(1)])
moment = rv.moment(n)
```

und wir erhalten tatsächlich `1/(n + 1)`.

Wir haben nun schon an einigen Stellen als Voraussetzung den Begriff der Integrierbarkeit benötigt. Dies wollen wir an dieser Stelle definieren.

Definition 3.10 (Integrierbarkeit und n-tes absolutes Moment)**:** *Sei $(\Omega, \mathcal{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Man nennt X n-mal integrierbar, falls gilt*

$$\|X\|_n := \mathbb{E}(|X|^n)^{\frac{1}{n}} < \infty \ .$$

Man bezeichnet $\mathbb{E}(|X|^n)$ auch als n-tes absolutes Moment (absolute moment).

Wir werden die Implementierung dieser Methode nicht vorführen, da im Vergleich zur vorigen Methode nur `self.variable**n` durch `sym.Abs(self.variable)**n` ersetzt wird. Auf ein Beispiel wollen wir jedoch nicht verzichten.

Beispiel 3.11 (n-tes absolutes Moment Exponentialverteilung)**:** Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Wir berechnen das erste absolute Moment von Hand. Betrachte

$$\begin{aligned}\mathbb{E}(|X|) &= \int_{\mathbb{R}} |x|\, \lambda \exp(-\lambda x) \mathbb{1}_{[0,\infty)}\, \mathrm{d}x \\ &= \int_0^\infty \lambda\, |x| \exp(-\lambda x)\, \mathrm{d}x \ .\end{aligned}$$

Da wir x nur auf $[0, \infty)$ betrachten und dies dort nichtnegativ ist, können wir den Betrag weglassen und erhalten

$$= \int_0^\infty \lambda x \exp(-\lambda x)\, \mathrm{d}x \ .$$

Mit partieller Integration erhalten wir

$$\begin{aligned}&= [-x \exp(-\lambda x)]_0^\infty + \int_0^\infty 1 \exp(-\lambda x)\, \mathrm{d}x \\ &= [-0 + 0] + \int_0^\infty \exp(-\lambda x)\, \mathrm{d}x \\ &= \left[-\frac{1}{\lambda} \exp(-\lambda x)\right]_0^\infty \\ &= -0 + \frac{1}{\lambda} \\ &= \frac{1}{\lambda} \ .\end{aligned}$$

Verwenden wir nun

```
lamda = sym.symbols('lambda', real=True, positive=True)
x = sym.symbols('x', real=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, supp=[sym.Integer(0), sym.oo])
absolute_moment = rv.absolute_moment(1)
```

so erhalten wir ebenso `1/lambda`. Wollten wir nun aber allgemein für $n \in \mathbb{N}$ das n-te absolute Moment berechnen, so würden wir nach einer kurzen Rechnung von Hand feststellen, dass wir n-mal partiell integrieren müssen. Viel einfacher ist es an den folgenden Teil anzufügen

```
n = sym.symbols('n', integer=True, nonnegative=True)
general_absolute_moment = rv.absolute_moment(n)
```

Wir erhalten `factorial(n)/lambda**n`. Also ist

$$\mathbb{E}(|X|^n) = \frac{n!}{\lambda^n}$$

Somit existieren zur Exponentialverteilung absolute Momente beliebiger Ordnung. Insbesondere stimmen die absoluten und die rohen Momente überein.

Satz 3.12 (Integrierbarkeit)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Ist X n-mal integrierbar, so ist X auch m-mal integrierbar mit $m \leq n$.*

Beweis :
Dies folgt aus der allgemeinen Schachtelung der L^p- beziehungsweise $\mathcal{L}^p$-Räume. □

Beispiel 3.13 (Nicht-integrierbare Familie)**:** Wir werden uns in diesem Beispiel mit einer Familie von Zufallsvariablen beschäftigen, die nur bis zu einem bestimmten Grad integrierbar sind. Sei zu $k \in \mathbb{N}_{>1}$ die Dichte von X_k gegeben durch

$$\varphi_k(x) = c_k \frac{1}{1+x^k} \mathbb{1}_{[0,\infty)}(x)$$

für alle $x \in \mathbb{R}$. Wir werden in einem ersten Schritt die Konstanten c_k bestimmen, damit diese Funktionen Dichten sind. Betrachte also

$$\begin{aligned} \mathbb{P}_{X_k}(\mathbb{R}) &= \int_{\mathbb{R}} c_k \frac{1}{1+x^k} \mathbb{1}_{[0,\infty)}(x) \, \mathrm{d}x \\ &= c_k \int_0^\infty \frac{1}{1+x^k} \, \mathrm{d}x \ . \end{aligned}$$

Dieses Integral ist leider nur für $k = 1, 2$ einfach lösbar. Wir verwenden also die Bibliothek.

```
x = sym.Symbol('x', real=True)
k = sym.Symbol('k', integer=True, positive=True)
density = 1 / (1 + x**k)
rv = RandomVariableContinuous(density, x, supp=[sym.Integer(0), sym.oo], force_density=True)
value = rv._is_density()
```

Wir erhalten

$$\mathbb{P}_{X_k}(\mathbb{R}) = c_k \cdot \frac{\pi}{k \sin\left(\frac{\pi}{k}\right)} \ .$$

Da dies eins sein muss, erhalten wir äquivalent

$$c_k = \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \ .$$

Also ist

$$\varphi_k(x) = \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \frac{1}{1+x^k} \mathbb{1}_{[0,\infty)}(x)$$

eine Wahrscheinlichkeitsdichte. Diese ist nicht-negativ, da alle Faktoren nicht-negativ sind. Insbesondere ist auch der Sinus-Term nicht-negativ, da $\pi/k \in [0, \pi]$ ist für alle $k \in \mathbb{N}_{>1}$ und der Sinus dort nicht-negativ ist. Verwenden wir also

```
density = k / sym.pi * sym.sin(sym.pi / k) * 1 / (1 + x**k)
rv = RandomVariableContinuous(density, x, supp=[sym.Integer(0), sym.oo])
value = rv._is_density()
```

so erhalten wir `1` und können auf `force_density` verzichten. Die Warnung wegen dem Auflösen stückweisen Funktion erscheint, da $k > 1$ sein muss und wir dies mit SymPy leider nicht definieren können. Die Zufallsvariable X_k ist nur $(k-2)$-mal integrierbar. Betrachte für $n > k - 2$ mit dem Dichtekorollar

$$\begin{aligned} \mathbb{E}\left(|X_k|^n\right) &= \int_{\mathbb{R}} |x^n| \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \frac{1}{1+x^k} \mathbb{1}_{[0,\infty)}(x) \, \mathrm{d}x \\ &= \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \int_0^\infty \frac{|x^n|}{1+x^k} \, \mathrm{d}x \ . \end{aligned}$$

Da wir nur über die positive reelle Halbachse integrieren, können wir im Zähler den Betrag weglassen und wir erhalten

$$= \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \int_0^\infty \frac{x^n}{1+x^k} \,\mathrm{d}x \ .$$

Dies können wir Abschätzen

$$\begin{aligned} &\geq \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \int_0^\infty \frac{x^n}{x^k} \,\mathrm{d}x \\ &= \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \int_0^\infty x^{n-k} \,\mathrm{d}x \ . \end{aligned}$$

Integration liefert

$$= \frac{k}{\pi} \sin\left(\frac{\pi}{k}\right) \left[\begin{cases} \log(x), & n = k-1 \\ \frac{1}{n-k+1} x^{n-k+1}, & n > k-1 \end{cases} \right]_0^\infty \ .$$

Beide Fälle divergieren. Somit ist diese Zufallsvariable für $n > k-2$ nicht n-integrierbar. Verwenden wir SymPy zur Bestimmung des $(k-2)$-ten absoluten Moments, so erhalten wir mit

```
moment = rv.absolute_moment(k - 2)
```

den Wert `1`, weshalb die Zufallsvariable $(k-2)$-mal integrierbar ist. Für die Abschätzung betrachten wir

$$\frac{x^n}{1+x^k} \geq \frac{x^n}{x^k} \ .$$

Erweitern wir zum gemeinsamen Nenner, so erhalten wir

$$\frac{x^n x^k}{(1+x^k)x^k} \geq \frac{x^n(1+x^k)}{x^k(1+x^k)} \ .$$

Das Ungleichungszeichen bleibt erhalten, da sowohl $1+x^k$ als auch x^k auf $[0,\infty)$ nicht-negativ sind. Bringen wir beides auf dieselbe Seite, so gilt

$$\begin{aligned} 0 &\geq \frac{x^n(1+x^k) - x^n x^k}{x^k(1+x^k)} \\ &\geq \frac{x^n + x^n x^k - x^n x^k}{x^k(1+x^k)} \\ &\geq \frac{x^n}{x^k(1+x^k)} \ , \end{aligned}$$

da alle Faktoren nicht-negativ sind. An dieser Stelle sei nochmals eine Warnung bezüglich dem Auflösen von stückweisen Funktionen gegeben. Verwenden wir direkt

```
n = sym.Symbol('n', integer=True, positive=True)
moment = rv.absolute_moment(n)
```

so erhalten wir neben der Warnung `-sin(pi/k)/sin(pi*(k + n + 1)/k)`. Setzen wir hier $n = k$, so erhielten wir fälschlicherweise `-1`.

Verwenden wir

```
RandomVariable.no_chopping()
moment = rv.absolute_moment(n)
```

so erhalten wir die folgende, richtige Funktion für die absoluten Momente

$$\mathbb{E}\left(|X_k|^n\right) = \begin{cases} -\frac{\sin\left(\frac{\pi}{k}\right)}{\sin\left(\frac{\pi(k+n+1)}{k}\right)}, & \frac{n+1}{k} < 1 \\ \frac{k\sin\left(\frac{\pi}{k}\right)\int_0^\infty \frac{|x|^n}{x^k+1}\,\mathrm{d}x}{\pi}, & \text{sonst .} \end{cases}$$

Die Bedingung ist $n < k-1$, wie von uns gefordert. Nach dem Satz zur Integrierbarkeit hätte es mittels Negation schon genügt zu zeigen, dass $\mathbb{E}(|X_k|^{k-1})$ nicht endlich ist. Somit sind auch alle höheren absoluten Momente nicht mehr endlich. Man sollte sich immer davon überzeugen, ob das Ergebnis von SymPy Sinn ergibt, falls man diese Warnung bekommt. Dennoch ist es sehr sinnvoll, stückweise Funktionen zu kürzen, da dies in den meisten Fällen natürliche Beschränkungen wie $p \in (0,1)$ oder ähnliches sind.

Man könnte vermuten, dass es einen Satz gibt, der besagt, dass zu einer n-fach integrierbaren Zufallsvariable auch das n-te Moment existiert. Hierzu betrachten wir das folgende Gegenbeispiel.

Beispiel 3.14 (Cauchy-Verteilung)**:** Sei X Cauchy-verteilt. Seien dazu $x_0 \in \mathbb{R}$ und $\gamma > 0$. Die Dichte ist dann für $x \in \mathbb{R}$ gegeben durch

$$\varphi(x) = \frac{1}{\pi\gamma}\left(1 + \frac{(x-x_0)^2}{\gamma^2}\right)^{-1} .$$

Im einfachsten Fall von $\gamma = 1$ und $x_0 = 0$ finden wir

$$\begin{aligned} \varphi(x) &= \frac{1}{\pi}\left(1+x^2\right)^{-1} \\ &= \frac{1}{\pi}\frac{1}{1+x^2} . \end{aligned}$$

Dies ist fast ein Vertreter der obigen nicht-integrierbaren Familie. Die Cauchy-Verteilung ist aber auf ganz $\mathbb{R}$ und nicht nur auf $[0,\infty)$ definiert. Es lässt sich zeigen, dass $\|X\|_1 < \infty$ ist und X somit integrierbar ist. Betrachten wir hingeben $\mathbb{E}(X)$, so ist dies nicht definiert. Auch SymPy hat mit diesem Integral Probleme. Verwenden wir

```
x = sym.Symbol('x', real=True)
x_0 = sym.Integer(0)
gamma = sym.Integer(1)
density = 1 / (sym.pi * gamma) * 1 / (1 + (x - x_0)**2 / gamma**2)
rv = RandomVariableContinuous(density, x)
absolute_moment = rv.absolute_moment(1)
```

so lässt SymPy den Ausdruck für das erste absolute Moment stehen ohne ihn weiter auszuwerten. Verwenden wir nun

```
absolute_moment = sym.N(absolute_moment)
```

zur numerischen Berechnung, so erhalten wir `100`. Dieser Wert ist auf jeden Fall endlich. Lassen wir nun mit

```
mean = rv.mean()
```

den Erwartungswert berechnen, so erhalten wir `nan`. Für eine Cauchy-verteilte Zufallsvariable scheinen also die absoluten Momente endlich zu sein, ohne dass die entsprechenden rohen Moment existieren.

Auf die Frage der Integrierbarkeit wollen wir im Folgenden verzichten. Wir versuchen mit dem Programm entsprechende Moment zu berechnen und falls diese endlich sind, gibt SymPy uns hoffentlich das entsprechende Ergebnis. Ansonsten erhalten wir andere SymPy-Objekte, die wir nun kurz besprechen werden.

Bemerkung 3.15 (Unbestimmte Werte) [Tea23]**:** SymPy unterscheidet die unbestimmte Werte auf die folgenden Arten.

(i) `nan`: Für nicht-definierte Ausdrücke, denen man keinen Wert (indeterminate form) zuordnen kann, wie $0/0$ oder $\infty - \infty$, gibt SymPy ein `nan`-Objekt zurück. Dies steht für keine Zahl (not a number). Es sei erwähnt, dass die klassischerweise nicht-definierten Ausdrücke 0^0 und ∞^0 wie in Python üblich zu Eins ausgewertet werden.

(ii) `oo`: Für nicht-endliche Werte gibt SymPy `oo` für ∞ oder `-oo` für $-\infty$ zurück. Dies ist beispielsweise der Fall für $\int_0^\infty x \,\mathrm{d}x$ respektive $\int_{-\infty}^0 x \,\mathrm{d}x$.

(iii) `zoo`: Für nicht-endliche Werte im komplexen gibt SymPy `zoo` zurück. Dies ist eine komplexe Zahl von unendlichem Betrag mit unbekannter Phase. Dies tritt beispielsweise bei $1/0$ auf. Wir erhalten ein solches Objekt, falls SymPy auf dem Weg der Berechnung irgendwann versucht etwas im Komplexen zu berechnen oder zu vereinfachen. Dies tritt also häufig im Zusammenhang mit der Exponentialfunktion auf.

Wir können nun eine weitere Art von Momenten definieren.

Definition 3.16 (Zentralmoment)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine n-mal integrierbare Zufallsvariable. Zu $n \in \mathbb{N}$ definiert man das n-te Zentralmoment von X (central moment) durch*

$$\mu_n := \mathbb{E}\left((X - \mathbb{E}(X))^n\right) \ .$$

Code 3.17 (`_central_moment_integration`)**:** Auch hier ist die Implementierung dank Vorarbeit sehr einfach.

```
def _central_moment_integration(self, n):
    mean = self.mean()
    central_moment = self.integrate_random_variable((self.variable - mean)**n)
    return central_moment
```

Betrachten wir nun beispielsweise eine stetige Zufallsvariable mit Dichte φ. Dann gilt

$$\begin{aligned}\mu_n &= \mathbb{E}\left((X - \mathbb{E}(X))^n\right) \\ &= \int (X - \mathbb{E}(X))^n \,\mathrm{d}\mathbb{P} \ .\end{aligned}$$

Verwenden wir nun das Dichtekorollar, so erhalten wir mit $\mu = \mathbb{E}(X)$

$$= \int_{\mathbb{R}} (x - \mu)^n \varphi(x) \,\mathrm{d}x \ .$$

Die Integrationsmethode berechnet dasselbe. Die `mean`-Methode bestimmt den Erwartungswert. Dieser wird entweder mittels `_moment_integration` oder mit einer anderen Methode berechnet, welche wir noch später entwickeln werden.

Beispiel 3.18 (Zentralmomente)**:**

(i) Sei $X \sim \mathrm{Ber}(p)$ mit $p \in (0,1)$ Bernoulli-verteilt. Wir werden nun das sechste Zentralmomente berechnen. Betrachte mit $E = \{0,1\}$

$$\mu_6 = \sum_{n \in E} (n - \mathbb{E}(X))^6 \, \mathbb{P}(X = n) \ .$$

Wir kennen bereites $\mathbb{E}(X) = p$, es folgt also

$$\begin{aligned}
&= \sum_{n \in E} (n-p)^6 \, \mathbb{P}(X = n) \\
&= (0-p)^6 \, \mathbb{P}(X=0) + (1-p)^6 \mathbb{P}(X=1) \\
&= p^6(1-p) + (1-p)^6 p \\
&= p^5 p(1-p) + (1-p)^6 p(1-p) \\
&= p^6 - p^7 + (p^6 - 6p^5 + 15p^4 - 20p^3 + 15p^2 - 6p + 1)p \\
&= p^6 - p^7 + p^7 - 6p^6 + 15p^5 - 20p^4 + 15p^3 - 6p^2 + p \\
&= -5p^6 + 15p^5 - 20p^4 + 15p^3 - 6p^2 + p \ .
\end{aligned}$$

Verwenden wir

```
p = sym.Symbol('p', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
sixth_central_moment = rv.central_moment(6)
sixth_central_moment = sym.expand(sixth_central_moment)
```

so erhalten wir ebenfalls `-5*p**6 + 15*p**5 - 20*p**4 + 15*p**3 - 6*p**2 + p`.

(ii) Sei nun $X \sim \mathrm{Poiss}(\lambda)$ mit $\lambda > 0$ Poisson-verteilt. Die Zähldichte ist für $n \in \mathbb{N}_0$ gegeben durch

$$\varphi(n) = \frac{\lambda^n}{n!} \exp(-\lambda) \ .$$

Wir bestimmen das zweite Zentralmoment. Betrachte also nach [Soc23]

$$\begin{aligned}
\mathbb{E}\left(X(X-1)\right) &= \sum_{n=0}^{\infty} n(n-1) \mathbb{P}(X=n) \\
&= \sum_{n=0}^{\infty} n(n-1) \frac{\lambda^n}{n!} \exp(-\lambda) \ .
\end{aligned}$$

Wir können die Terme für $n = 0$ und $n = 1$ weglassen, da diese null sind. Damit folgt

$$\begin{aligned}
&= \exp(-\lambda) \sum_{n=2}^{\infty} n(n-1) \frac{\lambda^n}{n \cdot (n-1) \cdot (n-2)!} \\
&= \exp(-\lambda) \sum_{n=2}^{\infty} \frac{\lambda^2 \lambda^{n-2}}{(n-2)!} \\
&= \lambda^2 \exp(-\lambda) \sum_{n=2}^{\infty} \frac{\lambda^{n-2}}{(n-2)!} \ .
\end{aligned}$$

Ein Indexshift führt nun zu

$$= \lambda^2 \exp(-\lambda) \sum_{n=0}^{\infty} \frac{\lambda^n}{(n)!} \ .$$

Die hinter Summe ist genau die Definition der Exponentialfunktion, womit folgt

$$\begin{aligned} &= \lambda^2 \exp(-\lambda) \exp(\lambda) \\ &= \lambda^2 \ . \end{aligned}$$

Betrachte nun

$$\begin{aligned} \mathbb{E}\left(X(X-1)\right) &= \mathbb{E}(X^2 - X) \\ &= \mathbb{E}(X^2) - \mathbb{E}(X) \ . \end{aligned}$$

Mit $\mathbb{E}(X) = \lambda$ folgt dann

$$= \mathbb{E}(X^2) - \lambda$$

und äquivalent

$$\mathbb{E}(X^2) = \mathbb{E}\left(X(X-1)\right) + \lambda \ .$$

Einsetzen der obigen Rechnung liefert

$$= \lambda^2 + \lambda \ .$$

Wir finden damit

$$\begin{aligned} \mu_2 &= \mathbb{E}(X^2) - \mathbb{E}(X)^2 \\ &= \lambda^2 + \lambda - \lambda^2 \\ &= \lambda \ . \end{aligned}$$

Mit

```
lamda = sym.Symbol('lambda', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = lamda**n / sym.factorial(n) * sym.exp(- lamda)
rv = RandomVariableDiscrete(density, n)
second_central_moment = rv.central_moment(2)
```

erhalten wir ebenfalls `lambda`.

(iii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Wir berechnen nun das dritte Zentralmoment. Betrachte

$$\mu_3 = \mathbb{E}\left((X - \mathbb{E}(X))^3\right) \ .$$

Verwenden wir $\mathbb{E}(X) = 1/\lambda$ und das Dichtekorollar, so erhalten wir

$$\begin{aligned} &= \int_{-\infty}^{\infty} \left(x - \frac{1}{\lambda}\right)^3 \lambda \exp(-\lambda x) \mathbb{1}_{[0,\infty)}(x) \, \mathrm{d}x \\ &= \int_{0}^{\infty} \lambda \left(x - \frac{1}{\lambda}\right)^3 \exp(-\lambda x) \, \mathrm{d}x \ . \end{aligned}$$

Partielle Integration liefert

$$
\begin{aligned}
&= \left[-\left(x - \frac{1}{\lambda}\right)^3 \exp(-\lambda x) \right]_0^\infty + \int_0^\infty 3\left(x - \frac{1}{\lambda}\right)^2 \exp(-\lambda x)\,\mathrm{d}x \\
&= \left[-0 + \frac{1}{\lambda^3} \right] + \int_0^\infty 3\left(x - \frac{1}{\lambda}\right)^2 \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{1}{\lambda^3} + \int_0^\infty 3\left(x - \frac{1}{\lambda}\right)^2 \exp(-\lambda x)\,\mathrm{d}x\ .
\end{aligned}
$$

Nochmal partielle Integration liefert

$$
\begin{aligned}
&= \frac{1}{\lambda^3} + \left[-\frac{3}{\lambda}\left(x - \frac{1}{\lambda}\right)^2 \exp(-\lambda x) \right]_0^\infty + \int_0^\infty \frac{6}{\lambda}\left(x - \frac{1}{\lambda}\right) \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{1}{\lambda^3} + \left[-0 + \frac{3}{\lambda}\frac{1}{\lambda^2} \right] + \int_0^\infty \frac{6}{\lambda}\left(x - \frac{1}{\lambda}\right) \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{1}{\lambda^3} + \frac{3}{\lambda^3} + \int_0^\infty \frac{6}{\lambda}\left(x - \frac{1}{\lambda}\right) \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{2}{\lambda^3} + \int_0^\infty \frac{6}{\lambda}\left(x - \frac{1}{\lambda}\right) \exp(-\lambda x)\,\mathrm{d}x\ .
\end{aligned}
$$

Ein letztes Mal partielle Integration liefert

$$
\begin{aligned}
&= \frac{2}{\lambda^3} + \left[-\frac{6}{\lambda^2}\left(x - \frac{1}{\lambda}\right) \exp(-\lambda x) \right]_0^\infty + \int_0^\infty \frac{6}{\lambda^2} \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{2}{\lambda^3} + \left[-0 - \frac{6}{\lambda^2}\frac{1}{\lambda} \right] + \int_0^\infty \frac{6}{\lambda^2} \exp(-\lambda x)\,\mathrm{d}x \\
&= \frac{2}{\lambda^3} - \frac{6}{\lambda^3} + \int_0^\infty \frac{6}{\lambda^2} \exp(-\lambda x)\,\mathrm{d}x \\
&= -\frac{4}{\lambda^3} + \int_0^\infty \frac{6}{\lambda^2} \exp(-\lambda x)\,\mathrm{d}x\ .
\end{aligned}
$$

Dies können wir nun endlich mit dem Hauptsatz integrieren

$$
\begin{aligned}
&= -\frac{4}{\lambda^3} + \left[-\frac{6}{\lambda^3} \exp(-\lambda x) \right]_0^\infty \\
&= -\frac{4}{\lambda^3} + \left[-0 + \frac{6}{\lambda^3} \right] \\
&= -\frac{4}{\lambda^3} + \frac{6}{\lambda^3} \\
&= \frac{2}{\lambda^3}
\end{aligned}
$$

Mittels

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
third_central_moment = rv.central_moment(3)
```

erhalten wir ebenfalls `2/lambda**3`.

Wir werden nun sehen, dass einige Zentralmomente für beliebige Verteilungen gleich sind.

Bemerkung 3.19 (Triviale Zentralmomente)**:** Das nullte Zentralmoment einer Zufallsvariable ist immer eins. Betrachte

$$\begin{aligned}\mu_0 &= \mathbb{E}\left((X - \mathbb{E}(X))^0\right) \\ &= \mathbb{E}(1) \\ &= \int \varphi(x)\,\mathrm{d}x\end{aligned}$$

Da φ eine Dichtefunktion einer Zufallsvariable ist, gilt nach Definition von Wahrscheinlichkeitsmaßen

$$= 1 \ .$$

Diese Aussage können wir auch fast vollständig mit SymPy „beweisen“.

```
x = sym.Symbol('x', real=True)
density = sym.Function('varphi')(x)
rv = RandomVariableContinuous(density, x, force_density=True)
zeroth_central_moment = rv.central_moment(0)
```

Wir erhalten dann `Integral(varphi(x), (x, -oo, oo))`, was das Integral über die Dichte ist. Da wir eine Wahrscheinlichkeitsdichte verwendet haben, ist dies wie gefordert eins. Diesen letzten Schritt haben wir leider selbst machen müssen, da wir SymPy nicht erklären können, dass die Dichtefunktion normiert ist.

Das erste Zentralmoment einer integrierbaren Zufallsvariable verschwindet immer. Betrachte

$$\begin{aligned}\mu_1 &= \mathbb{E}\left((X - \mathbb{E})^1\right) \\ &= \mathbb{E}\left(X - \mathbb{E}(X)\right) \ .\end{aligned}$$

Da das Integral linear ist, folgt

$$\begin{aligned}&= \mathbb{E}(X) - \mathbb{E}(X) \\ &= 0 \ .\end{aligned}$$

Dies können wir leider nicht mit SymPy „beweisen“, da es keine Möglichkeit gibt SymPy zu erklären, dass der Erwartungswert endlich ist. Somit möchte SymPy das Integral nicht auseinanderziehen und vereinfachen.

Das erste nicht-triviale Zentralmoment ist also das zweite. Dieses bekommt sogar einen speziellen Namen.

Definition 3.20 (Varianz und Standardabweichung)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle, zweimal integrierbare Zufallsvariable. Die Varianz (variance) ist definiert durch*

$$\begin{aligned}\mathrm{Var}(X) &:= \mu_2 \\ &= \mathbb{E}\left((X - \mathbb{E}(X))^2\right) \ .\end{aligned}$$

Damit definiert man die Standardabweichung (standard deviation) durch

$$\sigma := \sqrt{\mathrm{Var}(X)} \ .$$

Man kann also auch $\mathrm{Var}(X) = \sigma^2$ *schreiben.*

Auf die Implementierung dieser neuen Definitionen wollen wir an dieser Stelle verzichten, da wir die allgemeineren Funktionen bereits definiert haben.

Beispiel 3.21 (Interpretation Standardabweichung) [Soc23]**:** Wie bei der Interpretation des Erwartungswertes wollen wir an dieser Stelle ebenfalls versuchen dieses Zentralmoment zu interpretieren. Sei $X \sim \mathrm{N}(\mu, \sigma)$ mit $\mu \in \mathbb{R}$ und $\sigma > 0$ normalverteilt. Die Dichte ist gegeben durch

$$\varphi(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) .$$

Um den riesigen Vorteil von SymPy zu zeigen, werden wir die Berechnung an dieser Stelle einmal von Hand durchführen. Betrachte nun

$$\begin{aligned}
\mathrm{Var}(X) &= \mathbb{E}\left((X - \mathbb{E}(X))^2\right) \\
&= \int_{-\infty}^{\infty} (x-\mu)^2 \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) \mathrm{d}x \\
&= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} (x-\mu)^2 \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) \mathrm{d}x .
\end{aligned}$$

Substituieren wir nun $z = x - \mu$, so ist $\mathrm{d}z = \mathrm{d}x$ und wir erhalten

$$= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} z^2 \exp\left(-\frac{1}{2}\frac{z^2}{\sigma^2}\right) \mathrm{d}z .$$

Substituieren wir nun $z = \sqrt{2}\sigma x$, so ist $\mathrm{d}z = \sqrt{2}\sigma\,\mathrm{d}x$ und wir erhalten

$$\begin{aligned}
&= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} (\sqrt{2}\sigma x)^2 \exp\left(-\frac{1}{2}\frac{(\sqrt{2}\sigma x)^2}{\sigma^2}\right) \sqrt{2}\sigma\,\mathrm{d}x \\
&= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} 2\sigma^2 x^2 \exp\left(-\frac{1}{2}\frac{2\sigma^2 x^2}{\sigma^2}\right) \sqrt{2}\sigma\,\mathrm{d}x .
\end{aligned}$$

Durch Kürzen erhalten wir nun

$$= \frac{2\sigma}{\sqrt{\pi}} \int_{-\infty}^{\infty} x^2 \exp\left(x^2\right) \mathrm{d}x .$$

Der Integrand ist symmetrisch um Null. Wir können das Integral also umschreiben zu

$$\begin{aligned}
&= \frac{2\sigma}{\sqrt{\pi}} 2 \int_{0}^{\infty} x^2 \exp\left(x^2\right) \mathrm{d}x \\
&= \frac{4\sigma}{\sqrt{\pi}} \int_{0}^{\infty} x^2 \exp\left(x^2\right) \mathrm{d}x .
\end{aligned}$$

Substituieren wir nun $z = x^2$, so ist $\mathrm{d}x = 1/(2\sqrt{z})\,\mathrm{d}z$ und wir erhalten

$$\begin{aligned}
&= \frac{4\sigma}{\sqrt{\pi}} \int_{0}^{\infty} z \exp(z) \frac{1}{2\sqrt{z}} \mathrm{d}z \\
&= \frac{2\sigma}{\sqrt{\pi}} \int_{0}^{\infty} z^{\frac{3}{2}-1} \exp(z)\,\mathrm{d}z .
\end{aligned}$$

Verwenden wir nun die Definition der Gammafunktion mit $\Gamma(k) = \int_0^\infty z^{k-1} \exp(-z)\,\mathrm{d}z$, so erhalten wir für $k = 3/2$

$$
\begin{aligned}
&= \frac{2\sigma}{\sqrt{\pi}} \Gamma\left(\frac{3}{2}\right) \\
&= \frac{2\sigma}{\sqrt{\pi}} \frac{\sqrt{\pi}}{2} \\
&= \sigma \ .
\end{aligned}
$$

Die Standardabweichung ist dann entsprechend σ. Verwenden wir nun SymPy, so benötigen wir

```
x = sym.Symbol('x', real=True)
mu = sym.Symbol('mu', real=True)
sigma = sym.Symbol('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
variance = rv.variance()
std = rv.standard_deviation()
```

Wir erhalten ebenso `sigma**2` und `sigma`. Diese Berechnung hat je nach Computerleistung nur wenige Sekunden gedauert, während die händische Kalkulation deutlich mehr Zeit in Anspruch nimmt.

Anhand einer Normalverteilung kann man die Bedeutung der Varianz beziehungsweise der Standardabweichung leicht verstehen. Sie ist ein Maß für die Abweichung vom Erwartungswert. Wir werden nun im folgenden Bild sehen, wie die Größer der Standardabweichung den Plot der Dichtefunktion ändert.

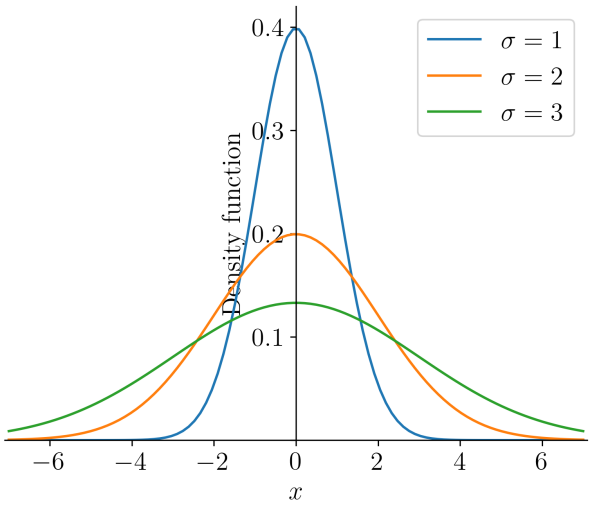

Abb. 15: Dichte einer $\mathrm{N}(0, \sigma)$-Verteilung

Wie oben beschrieben führen große Standardabweichungen zu einer größeren Streuung um den Erwartungswert, welcher hier in der Null liegt.

Bemerkung 3.22 (Berechnung Varianz)**:** In der obigen Definition ist die Varianz als ein Zentralmoment definiert. Wir können sie auch mittels der rohen Momente berechnen. Betrachte

$$
\begin{aligned}
\mathrm{Var}(X) &= \mathbb{E}\left((X - \mathbb{E}(X))^2\right) \\
&= \mathbb{E}\left(X^2 - 2X\mathbb{E}(X) + \mathbb{E}(X)^2\right) \ .
\end{aligned}
$$

Mit der Linearität des Erwartungswertes folgt

$$\begin{aligned} &= \mathbb{E}(X^2) - 2\mathbb{E}(X)\mathbb{E}(X) + \mathbb{E}(X)^2 \\ &= \mathbb{E}(X^2) - 2\mathbb{E}(X)^2 + \mathbb{E}(X)^2 \\ &= \mathbb{E}(X^2) - \mathbb{E}(X)^2 \ . \end{aligned}$$

Wir können die Varianz auch durch das zweite Moment minus das erste Moment zu Quadrat berechnen.

Diese Aussage gilt sogar allgemeiner.

Satz 3.23 (Darstellung der Zentralmomente)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle,* n*-mal integrierbare Zufallsvariable. Dann lässt sich das* n*-te Zentralmoment als Polynom der rohen Momente bis zum Grad* n *darstellen. Es gilt*

$$\mu_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} \mathbb{E}(X)^k \mathbb{E}(X^{n-k}) \ .$$

Beweis :
Betrachte

$$\mu_n = \mathbb{E}\left((X - \mathbb{E}(X))^n\right) \ .$$

Verwenden wir den binomische Lehrsatz, so erhalten wir

$$\begin{aligned} &= \mathbb{E}\left(\sum_{k=0}^{n} \binom{n}{k} X^{n-k} (-\mathbb{E}(X))^k\right) \\ &= \mathbb{E}\left(\sum_{k=0}^{n} (-1)^k \binom{n}{k} X^{n-k} \mathbb{E}(X)^k\right) \ . \end{aligned}$$

Da X n-mal integrierbar ist flogt mit der Linearität des Erwartungswertes

$$= \sum_{k=0}^{n} (-1)^k \binom{n}{k} \mathbb{E}(X)^k \mathbb{E}(X^{n-k}) \ ,$$

was zu zeigen war. □

Bemerkung 3.24 (Variationskoeffizient) [Wik24a]**:** Mit dem Begriff der Standardabweichung können wir den neuen Begriff des *Variationskoeffizienten* (coefficient of variation) definieren. Es gilt, falls $\mathbb{E}(X) \neq 0$ ist

$$\mathrm{VarK}(X) := \frac{\sigma}{\mathbb{E}(X)} \ .$$

In der Bibliothek ist er unter `coefficient_of_variation` zu finden und wird unter Aufruf der Methoden `mean` und `standard_deviation` berechnet.

Beispiel 3.25 (Variationskoeffizient)**:** Wir werden nun für ein Beispiel den Variationskoeffizient berechnen. Sei $X \sim \mathrm{N}(\mu, \sigma)$ normalverteilt mit $\mu \in \mathbb{R}$ und $\sigma > 0$. Den Erwartungswert werden wir an einer späteren Stelle noch berechnen. Die Standardabweichung kennen wir bereits. Es gilt

$$\begin{aligned} \mathbb{E}(X) &= \mu \\ \mathrm{Var}(X) &= \sigma^2 \ . \end{aligned}$$

Damit können wir den Variationskoeffizienten berechnen

$$\mathrm{VarK}(X) = \frac{\sigma}{\mu} \ .$$

Um uns manuelle Berechnung zu ersparen, können wir auch

```
x = sym.symbols('x', real=True)
mu = sym.symbols('mu', real=True)
sigma = sym.symbols('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
coefficient_of_variation = rv.coefficient_of_variation()
```

verwenden und erhalten ebenfalls `sigma/mu`.

Mithilfe der Standardabweichung können wir nun die Zentralmomente entsprechend standardisieren.

Definition 3.26 (Standardmoment)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine* n*-mal integrierbare Zufallsvariable mit Standardabweichung* $\sigma \neq 0$*. Zu* $n \in \mathbb{N}_{>1}$ *ist das* n*-te Standardmoment von* X *(standardised moment) definiert durch*

$$\begin{aligned}\tilde{\mu}_n &:= \frac{\mu_n}{\sigma^n} \\ &= \frac{\mathbb{E}\left((X - \mathbb{E}(X))^n\right)}{\mathbb{E}\left((X - \mathbb{E}(X))^2\right)^n} \ .\end{aligned}$$

Code 3.27 (`_standard_moment_integration`)**:** Die Implementierung lehnt sich eher an die erste Schreibweise in der obigen Definition an.

```
def _standard_moment_integration(self, n):
    central_moment = self.central_moment(n, use_integration=True)
    standard_deviation = self.standard_deviation(use_integration=True)
    standard_moment = central_moment / standard_deviation**n
    standard_moment = sym.simplify(standard_moment)
    return standard_moment
```

Wir bestimmen zuerst mit den entsprechenden Methoden das Zentralmoment und die Standardabweichung. Wie in der Definition dividieren wir diese und lassen dies nach Möglichkeit noch mit SymPy vereinfachen.

Bemerkung 3.28 (Triviale Standardmomente)**:** Ähnlich wie bei den Zentralmomenten ist das nullte Standardmoment immer eins. Betrachte

$$\begin{aligned}\tilde{\mu}_0 &= \frac{\mu_0}{\sigma^0} \\ &= \frac{1}{1} \\ &= 1.\end{aligned}$$

Für diese Tatsache können wir wieder einen „Beweis“ mit SymPy führen.

```
x = sym.Symbol('x', real=True)
f = sym.Function('f')(x)
rv = RandomVariableContinuous(f, x, force_density=True)
zeroth_standardized_moment = rv.standard_moment(0)
```

Wir erhalten wie im Bemerkung zu den Zentralmomenten `Integral(f(x), (x, -oo, oo))`, was aufgrund der Normiertheit einer Dichte genau eins ist.

Das erste Standardmoment ist null. Es gilt

$$\begin{aligned}\tilde{\mu}_1 &= \frac{\mu_1}{\sigma^1} \\ &= \frac{0}{\sigma} \\ &= 0 \ .\end{aligned}$$

Wie bei den Zentralmomenten können wir dies vermutlich aus ähnlichen Gründen nicht „beweisen“.

Zu guter Letzt ist das zweite Standardmoment immer eins. Betrachte dazu

$$\begin{aligned}\tilde{\mu}_2 &= \frac{\mu_2}{\sigma^2} \\ &= \frac{\sigma^2}{\sigma^2} \\ &= 1\end{aligned}$$

Dies können wir wieder „beweisen“ lassen. Durch das Anfügen von

```
5 second_standardized_moment = rv.standard_moment(2)
```

erhalten wir `1`.

Bemerkung 3.29 (Spezielle Standardmomente) [Wik24b]**:** Analog zur Varianz gibt es auch hier Standardmomente, die einen speziellen Namen haben. Das dritte Standardmoment bezeichnet man als Schiefe (skewness) und das vierte als Wölbung (kurtosis). Weniger gebräuchlich sind die Begriffe für das fünfte und sechste Standardmoment, nämlich *Hyperschiefe* (hyperskewness) respektive Hypertailedness (hypertailedness). Diese Methoden sind unter den entsprechenden englischen Begriffen implementiert. Auch hier möchte ich auf eine Vorstellung des Codes verzichten, da einfach die `standard_moment`-Methode mit Parameter `3` und `4` beziehungsweise `5` und `6` aufgerufen wird.

Wir werden nun anhand einiger Beispiele diese Standardmomente berechnen.

Beispiel 3.30 (Standardmomente)**:**

(i) Sei $X \sim \mathrm{Ber}(p)$ Bernoulli-verteilt mit $p \in (0,1)$. Wir berechnen die Hypertailedness. Betrachte also

$$\tilde{\mu}_6 = \frac{\mu_6}{\sigma^6} \,.$$

μ_6 haben wir oben bereits berechnet und $\sigma^2 = p(1-p)$. Damit folgt

$$\begin{aligned}&= \frac{p^5 p(1-p) + (1-p)^6 p(1-p)}{\sqrt{p(1-p)}^6} \\ &= \frac{p^5 p(1-p) + (1-p)^6 p(1-p)}{(p(1-p))^3} \\ &= \frac{p^5 + (1-p)^6}{(p(1-p))^2} \\ &= \frac{p^6 - 5p^5 + 15p^4 - 20p^3 + 15p^2 - 6p + 1}{p^4 - 2p^3 + p^2} \,.\end{aligned}$$

Durch

```
1 p = sym.Symbol('p', real=True, positive=True)
2 n = sym.Symbol('n', integer=True, nonnegative=True)
3 density = {1: p, 0: 1 - p}
4 rv = RandomVariableFinite(density, n)
5 hypertailedness = rv.hypertailedness()
```

erhalten wir auch `(p**5 - (p - 1)**5)/(p**2*(p - 1)**2)`.

(ii) Sei $X \sim \text{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Wir berechnen die Schiefe. Betrachte also

$$\tilde{\mu}_3 = \frac{\mu_3}{\sigma^3} \ .$$

Wir haben bereits $\mu_3 = 2/\lambda^3$ berechnet. Mit $\sigma = 1/\lambda$ folgt

$$\begin{aligned} &= \frac{\frac{2}{\lambda^3}}{\frac{1}{\lambda^3}} \\ &= \frac{2\lambda^3}{\lambda^3} \\ &= 2 \ . \end{aligned}$$

Mit

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
skewness = rv.skewness()
```

erhalten wir dasselbe Ergebnis. Es ist interessant, dass alle Exponentialverteilungen unabhängig von der Wahl von λ gleich schief sind.

(iii) Sei $X \sim \text{N}(\mu, \sigma)$ mit $\mu \in \mathbb{R}$ und $\sigma > 0$ normalverteilt. Wir werden an dieser Stelle Schiefe und Wölbung nicht von Hand berechnen. Mit

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
skewness = rv.skewness()
kurtosis = rv.kurtosis()
```

erhalten wir `0` und `3`.

Weitere Beispiele lassen sich einfach mit dem Code selbst ausrechnen.

Wir wollen an dieser Stelle mögliche Interpretationen für die Schiefe und Wölbung finden.

Beispiel 3.31 (Interpretation Schiefe und Wölbung)**:** Die Schiefe ist ein Maß dafür, wie symmetrisch oder asymmetrisch eine Verteilung ist. Ist die Schiefe positive, so liegt mehr Masse rechts vom Erwartungswert. Umgekehrtes gilt für negative Schiefen. Wie wir in obigem Beispiel gesehen haben, verschwindet die Schiefe der Normalverteilung. Dies liegt daran, dass die Dichte symmetrisch um den Erwartungswert ist. Außerdem haben wir oben gesehen, dass die Schiefe einer Exponentialverteilung immer zwei ist. An der Dichtefunktion können wir schon erahnen, dass die Schiefe positive sein muss.

Die Wölbung ist ein Maß dafür, wie steil oder flach sich die Verteilung um den Erwartungswert verhält. Da die Wölbung einer Normalverteilung immer drei ist, kann man andere Verteilungen mit der Normalverteilung vergleichen, indem man den *Exzess* (excess kurtosis) definiert. Diesen erhält man, indem man drei von der Wölbung der Zufallsvariable abzieht. Er dient als Vergleichswert, wie schnell die Enden einer Verteilung gegen Null abfallen und ist unter `excess_kurtosis` zu finden. Verschwindet der Exzess, so bezeichnet man die Verteilung als *mesokurtisch* (mesocurtic). Dies ist zum Beispiel die Normalverteilung. Ist der Exzess positiv beziehungsweise negativ, so bezeichnet man die Verteilung als *leptokurtisch* (leptokurtic) respektive *platykurtisch* (platykurtic). Eine Methode, die den Exzess einer parameterfreien Verteilung interpretiert, lässt sich unter `interpret_excess_kurtosis` finden.

(i) Betrachte eine (μ, σ)-Normalverteilung. Verwenden wir

```
x = sym.Symbol('x', real=True)
mu = sym.Symbol('mu', real=True)
sigma = sym.Symbol('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
rv.interpret_excess_kurtosis()
```

so erhalten wir `The Distribution is mesokurtic`, wie erwartet.

(ii) Sei nun $X \sim \text{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Wir würden erwarten, dass der Exzess negativ ist, da eine Exponentialfunktion nur in $\exp(-x)$ und die Normalverteilung mit $\exp(-x^2)$ schneller abfällt. Mit

```
lamda = sym.symbols('lambda', real=True, positive=True)
x = sym.Symbol('x', real=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
rv.interpret_excess_kurtosis()
```

erhalten wir `The Distribution is leptokurtic`, wie vorhergesagt.

(iii) Um ein platykurtisches Beispiel zu finden, benötigen wir eine Dichte, die schneller als $\exp(-x^2)$ abfällt. Wir können beispielsweise $\exp(-x^4)$ verwenden. Wir beginnen mit

```
x = sym.Symbol('x', real=True)
density = sym.exp(- x**4)
rv = RandomVariableContinuous(density, x)
value = rv._is_density()
```

und erhalten `gamma(1/4)/2` als Ergebnis der nicht-normierten Dichte. Eine Wahrscheinlichkeitsdichte ist also

$$\varphi(x) = \frac{2}{\Gamma\left(\frac{1}{4}\right)} \exp(-x^4) \ .$$

Mit

```
density = sym.Integer(2) / sym.gamma(sym.Rational(1, 4)) * sym.exp(- x**4)
rv = RandomVariableContinuous(density, x)
rv.interpret_excess_kurtosis()
```

erhalten wir `The Distribution is platykurtic`. Dies ist genau so, wie wir die Verteilung „gebaut“ haben.

Im nächsten Abschnitt werden wir die versprochene andere Möglichkeit finden, mit der man Momente berechnen kann.

3.3 Momenterzeugende Funktionen

Wir beginnen dieses Kapitel mit der folgenden

Definition 3.32 (Momenterzeugende Funktion)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Die momenterzeugende Funktion von X (moment-generating function) ist definiert durch*

$$M_X(t) := \mathbb{E}\left(\exp(tX)\right) \ .$$

Insbesondere ist $M_X : \mathbb{D} \to \mathbb{R} : t \mapsto M_X(t)$ für $\mathbb{D} := \{t \in \mathbb{R} \mid M_X(t) < \infty\}$ eine Abbildung.

Wir werden nun die momenterzeugende Funktion implementieren.

Code 3.33 (`moment_generating_function`)**:** Dank der allgemeinen Integrationsmethode ist auch an dieser Stelle nicht viel zu implementieren.

```
def moment_generating_function(self):
    if hasattr(self, 'MGF'):
        moment_generating_function = self.MGF
    else:
        t = sym.Symbol('t', real=True)
        moment_generating_function = self.integrate_random_variable(sym.exp(t * self.variable))
        self.MGF = moment_generating_function
    return moment_generating_function
```

Neu ist, dass wir zuvor noch das Symbol `t` definieren müssen. Da dieses Symbol hier verwendet wird, sollte man in der Definition seiner Dichtefunktion kein `t` verwenden, da dies sonst zu großen Schwierigkeiten führt. Weiter prüfen wir zuerst, ob das Attribut `MGF` schon belegt ist. In diesem Attribut wird die momenterzeugende Funktion nach der ersten Berechnung gespeichert, um Berechnungen zu beschleunigen.

Zur momenterzeugenden Funktion gibt es auch eine Visualisierungsmethode. Diese ist unter `plot_moment_generating_function` zu finden und funktioniert wie `plot_denisty` für stetige Zufallsvariablen. Auf Beispiele zur Visualisierung wollen wir verzichten.

Um zu zeigen, wieso in der Dichte kein `t` verwendet werden sollte, definieren wir beispielsweise folgendermaßen eine Exponentialverteilung

```
t = sym.Symbol('t', real=True)
lamda = sym.Symbol('lamda', real=True, positive=True)
density = lamda * sym.exp(- lamda * t)
rv = RandomVariableContinuous(density, t, [sym.Integer(0), sym.oo])
moment_generating_function = rv.moment_generating_function()
```

SymPy liefert uns `lamda*Integral(exp(t**2)*exp(-lamda*t), (t, 0, oo))` für die momenterzeugende Funktion. Dieses Integral kann SymPy nicht berechnen, da es aufgrund des t^2 Terms divergiert. Wir werden im folgenden Beispiel die richtige momenterzeugende Funktion berechnen.

Bemerkung 3.34 (Momenterzeugende Funktion in Null)**:** Sei X eine reelle Zufallsvariable und M_X eine um Null existierende momenterzeugende Funktion. Betrachte

$$\begin{aligned} M_X(0) &= \mathbb{E}(\exp(0 \cdot x)) \\ &= \mathbb{E}(\exp(0)) \\ &= \mathbb{E}(1) \\ &= \int 1 \, \mathrm{d}\mathbb{P} \, . \end{aligned}$$

Da $\mathbb{P}$ ein Wahrscheinlichkeitsmaß ist, folgt aus der Normiertheit

$$= 1 \, .$$

Dies werden wir später noch benötigen. Diese Aussage können wir wieder mit SymPy „beweisen“. Mit

```
x = sym.Symbol('x', real=True)
t = sym.Symbol('t', real=True)
f = sym.Function('f')(x)
rv = RandomVariableContinuous(f, x, force_density=True)
moment_generating_function = rv.moment_generating_function()
value = moment_generating_function.subs(t, sym.Integer(0))
```

erhalten wir `Integral(f(x), (x, -oo, oo))`, was eins ist.

Wir werden nun einige Beispiele für momenterzeugende Funktionen berechnen.

Beispiel 3.35 (Momenterzeugende Funktionen)**:**

(i) Sei $X \sim \mathrm{Ber}(p)$ mit $p \in (0,1)$ Bernoulli-verteilt. Wir berechnen mit $E = \{0,1\}$

$$\begin{aligned} M_X(t) &= \int \exp(tX)\,\mathrm{d}\mathbb{P} \\ &= \sum_{n\in E} \exp(tn)\mathbb{P}(X=n) \\ &= \exp(t\cdot 0)\mathbb{P}(X=0) + \exp(t\cdot 1)\mathbb{P}(X=1) \\ &= \exp(0)(1-p) + \exp(t)p \\ &= (1-p) + p\exp(t)\ . \end{aligned}$$

Um dies mit dem Programm zu berechnen, verwenden wir

```
p = sym.Symbol('p', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
moment_generating_function = rv.moment_generating_function()
```

und erhalten ebenfalls `p*exp(t) - p + 1`.

(ii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Wir berechnen

$$\begin{aligned} M_X(t) &= \int \exp(tX)\,\mathrm{d}\mathbb{P} \\ &= \int_{-\infty}^{\infty} \exp(tx)\lambda\exp(-\lambda x)\mathbb{1}_{[0,\infty)}(x)\,\mathrm{d}x \\ &= \lambda\int_0^{\infty} \exp\left((t-\lambda)x\right)\mathrm{d}x \\ &= \lambda\left[\frac{1}{t-\lambda}\exp\left((t-\lambda)x\right)\right]_0^{\infty} \\ &= \lambda\left[0 - \frac{1}{t-\lambda}\right] \\ &= \frac{\lambda}{\lambda - t}\ . \end{aligned}$$

Durch

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
moment_generating_function = rv.moment_generating_function()
```

erhalten wir ebenfalls `lambda/(lambda - t)`.

(iii) Sei nun $X \sim \mathrm{N}(\mu,\sigma)$ mit $\mu \in \mathbb{R}$ und $\sigma > 0$ normalverteilt. Diese momenterzeugende Funktion wollen wir nicht von Hand berechnen, da dies ziemlich kompliziert ist. SymPy schafft dies folgendermaßen problemlos.

```
x = sym.symbols('x', real=True)
mu = sym.symbols('mu', real=True)
sigma = sym.symbols('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
moment_generating_function = rv.moment_generating_function()
```

Wir erhalten `exp(t*(mu + sigma**2*t/2))`, wie in [Soc23] nachzulesen ist.

Wir werden nun einen Satz beweisen, der es uns erlaubt die momenterzeugenden Funktion als eine Potenzreihe der Momente darzustellen.

Satz 3.36 (Potenzreihendarstellung der momenterzeugenden Funktion) [MS05]**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable. Ist* M_X *die momenterzeugende Funktion und existiert ein* $\varepsilon > 0$*, sodass* $(-\varepsilon, \varepsilon) \subseteq \mathbb{D}$*, so gilt*

$$M_X(t) = \sum_{n=0}^{\infty} \frac{t^n}{n!} \mathbb{E}(X^n) \ .$$

Beweis [MS05]*:*
Betrachte die Potenzreihe der Exponentialfunktion

$$\exp(tc) = \sum_{n=0}^{\infty} \frac{t^n}{n!} c^n \ .$$

Wir erkennen eine deutliche Ähnlichkeit zu dem, was wir zeigen wollen. Wir müssen also beweisen, dass wir Erwartungswert und Summation vertauschen dürfen. Sei für $N \in \mathbb{N}$ durch

$$f_N(X) := \sum_{n=0}^{N} \frac{t^n}{n!} X^n$$

eine Folge von Zufallsvariablen definiert. Dann gilt nach obiger Potenzreihe

$$\begin{aligned} M_X(t) &= \mathbb{E}\left(\exp(tX)\right) \\ &= \mathbb{E}\left(\lim_{N\to\infty} f_N(X)\right) \ . \end{aligned}$$

Da nach Definition $\exp(tX)$ überall auf $\mathbb{D}$ $\mathbb{P}_X$-integierbar ist, können wir mit dem Satz von Lebesgue Limes und Integration vertauschen und erhalten

$$\begin{aligned} &= \lim_{N\to\infty} \mathbb{E}\left(f_N(X)\right) \\ &= \lim_{N\to\infty} \mathbb{E}\left(\sum_{n=0}^{N} \frac{t^n}{n!} X^n\right) \ . \end{aligned}$$

Dank Linearität des Erwarungswertes folgt für diese endliche Summe

$$\begin{aligned} &= \lim_{N\to\infty} \sum_{n=0}^{N} \frac{t^n}{n!} \mathbb{E}(X^n) \\ &= \sum_{n=0}^{\infty} \frac{t^n}{n!} \mathbb{E}(X^n) \ , \end{aligned}$$

was zu zeigen war. □

Damit können wir nun mit der momenterzeugenden Funktion die Momente berechnen.

Korollar 3.37 (Momente mit momenterzeugenden Funktion)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable. Ist* M_X *die momenterzeugende Funktion und existiert ein* $\varepsilon > 0$*, sodass* $(-\varepsilon, \varepsilon) \subseteq \mathbb{D}$*, so gilt*

$$\mathbb{E}(X^n) = \left[\frac{\mathrm{d}^n}{\mathrm{d}t^n} M_X(t)\right]_{t=0}$$

und alle Momente sind endlich.

Beweis :
Betrachte zuerst

$$\frac{\mathrm{d}^n}{\mathrm{d}t^n}t^k = \begin{cases} \frac{k!}{(n-k)!}t^{k-n}, & n < k \\ n!, & n = k \\ 0, & n > k \end{cases}.$$

Setzen wir $t = 0$ ein, so gilt

$$\left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}t^k\right]_{t=0} = \begin{cases} 0, & n < k \\ n!, & n = k \\ 0, & n > k \end{cases}.$$

Mit dem Satz über die Potenzreihendarstellung der momenterzeugenden Funktion gilt

$$\left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}M_X(t)\right]_{t=0} = \left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}\sum_{k=0}^{\infty}\frac{t^k}{k!}\mathbb{E}(X^k)\right]_{t=0}.$$

Da eine Potenzreihe im inneren ihre Konvergenzradiuses unendlich oft differenzierbar ist, folgt durch Vertauschen von Differentiation und Summenbildung

$$= \left[\sum_{k=0}^{\infty}\frac{\mathrm{d}^n}{\mathrm{d}t^n}\frac{t^k}{k!}\mathbb{E}(X^k)\right]_{t=0}.$$

Nach obiger Betrachtung bleibt nur für Summanden mit $n = k$ der Faktor $n!$ übrig. Es folgt also

$$\begin{aligned} &= \frac{n!}{n!}\mathbb{E}(X^n) \\ &= \mathbb{E}(X^n), \end{aligned}$$

was zu zeigen war. □

Dieses Korollar werden wir als Methode zur alternative Berechnung der Momente implementieren.

Code 3.38 (`_moment_generating`)**:** Wir haben nun gesehen, dass wir mithilfe der momenterzeugenden Funktion die Momente einer Zufallsvariable erzeugen können. Da wir hier einige Werkzeuge von SymPy verwenden, die wir bisher noch nicht benötigt haben, wollen wir hier den Code besprechen.

```
def _moment_generating(self, n):
    t = sym.Symbol('t', real=True)
    moment_generating_function = self.moment_generating_function()
    moment = sym.diff(moment_generating_function, (t, n))
    moment = moment.subs(t, sym.Integer(0))
    moment = sym.simplify(moment)
    return moment
```

Zuerst lassen wir uns die momenterzeugende Funktion bestimmen. Diese leiten wir mithilfe von SymPy `n` mal nach `t` ab. Anschließend setzen wir für `t` den Wert `0` ein und versuchen dies zu vereinfachen. Es ist es wichtig in der Ableitung `(t, n)` zu klammern, falls man für `n` ein Symbol verwenden möchte.

Dazu betrachten wir das folgende kurze Beispiel.

```
x = sym.Symbol('x', real=True)
n = sym.Symbol('n', integer=True, positive=True)
f = sym.exp(x**2)
wrong = sym.Derivative(f, x, n)
right = sym.Derivative(f, (x, n))
```

Die `Derivative`-Methode ist eine noch nicht ausgeführte Version der `diff` Methode. Wir erhalten im ersten Fall

$$\frac{\mathrm{d}^2}{\mathrm{d}x\,\mathrm{d}n}\exp(x^2)\ ,$$

was wir nicht wollen und im zweiten Fall

$$\frac{\mathrm{d}^n}{\mathrm{d}x^n}\exp(x^2)\ .$$

Mit dieser Methode ist es leider nicht möglich das allgemeine n-te Moment zu berechnen, da SymPy keine Funktion n-mal ableiten möchte. Selbst

```
x = sym.Symbol('x', real=True)
n = sym.Symbol('n', integer=True, postive=True)
expr = sym.exp(x)
derivative = sym.diff(expr, (x, n))
```

bleibt unevaluiert.

Mithilfe dieser neuen Methode werden wir nun einige Momente berechnen.

Beispiel 3.39 (Momente mit momenterzeugenden Funktionen)**:** Wir verwenden die momenterzeugenden Funktionen aus obigem Beispiel

(i) Sei $X \sim \mathrm{Ber}(p)$ mit $p \in (0,1)$ Bernoulli-verteilt. Die momenterzeugende Funktion ist dann

$$M_X(t) = (1-p) + p\exp(t)\ .$$

Wir berechnen beispielsweise das vierte Moment

$$\begin{aligned}\mathbb{E}(X^4) &= \left[\frac{\mathrm{d}^4}{\mathrm{d}t^4}M_X(t)\right]_{t=0}\\ &= \left[\frac{\mathrm{d}^4}{\mathrm{d}t^4}(1-p)+p\exp(t)\right]_{t=0}\ .\end{aligned}$$

Der erste Summand fällt beim Ableiten weg, da dort kein t vorkommt und der hintere verändert sich nicht. Somit gilt

$$\begin{aligned}&= [p\exp(t)]_{t=0}\\ &= p\exp(0)\\ &= p\end{aligned}$$

Verwenden wir

```
p = sym.Symbol('p', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
fourth_moment = rv._moment_generating(4)
```

so erhalten wir ebenfalls `p`.

(ii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Die momenterzeugende Funktion ist

$$M_X(t) = \frac{\lambda}{\lambda - t} \ .$$

Wir berechnen das zweite Moment

$$\begin{aligned} \mathbb{E}(X^2) &= \left[\frac{\mathrm{d}^2}{\mathrm{d}t^2} M_X(t) \right]_{t=0} \\ &= \left[\frac{\mathrm{d}^2}{\mathrm{d}t^2} \frac{\lambda}{\lambda - t} \right]_{t=0} \ . \end{aligned}$$

Einmal Ableiten liefert

$$= \left[\frac{\mathrm{d}}{\mathrm{d}t} \frac{\lambda}{(\lambda - t)^2} \right]_{t=0}$$

und nochmal Ableiten liefert dann

$$\begin{aligned} &= \left[\frac{2\lambda}{(\lambda - t)^3} \right]_{t=0} \\ &= \frac{2\lambda}{(\lambda - 0)^3} \\ &= \frac{2}{\lambda^2} \ . \end{aligned}$$

Durch

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
second_moment = rv._moment_generating(2)
```

erhalten wir `2/lambda**2`.

(iii) Sei nun $X \sim \mathrm{N}(\mu, \sigma)$ mit $\mu \in \mathbb{R}$ und $\sigma > 0$ normalverteilt. Die momenterzeugende Funktion ist

$$\begin{aligned} M_X(t) &= \exp\left(t \left(\mu + \frac{\sigma^2 t}{2} \right) \right) \\ &= \exp\left(\mu t + \frac{\sigma^2}{2} t^2 \right) \ . \end{aligned}$$

Hier wollen wir nur das erste Moment bestimmen. Betrachte also

$$\begin{aligned} \mathbb{E}(X) &= \left[\frac{\mathrm{d}}{\mathrm{d}t} M_X(t) \right]_{t=0} \\ &= \left[\frac{\mathrm{d}}{\mathrm{d}t} \exp\left(\mu t + \frac{\sigma^2}{2} t^2 \right) \right]_{t=0} \ . \end{aligned}$$

Die Kettenregel liefert

$$\begin{aligned} &= \left[(\mu + \sigma^2 t) \exp\left(\mu t + \frac{\sigma^2}{2} t^2 \right) \right]_{t=0} \\ &= (\mu + \sigma^2 \cdot 0) \exp(0) \\ &= \mu \ . \end{aligned}$$

Das Programm liefert mit

```
x = sym.symbols('x', real=True)
mu = sym.symbols('mu', real=True)
sigma = sym.symbols('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
first_moment = rv._moment_generating(1)
```

ebenfalls `mu`.

Mittels Integration erhalten wir jeweils dasselbe Ergebnis.

Diese Methoden zur Berechnung der Momente wollen wir nun zusammenfassen.

Code 3.40 (`moment`)**:**

```
def moment(self, n, use_integration=True):
    if use_integration == True:
        moment = self._moment_integration(n)
    else:
        moment = self._moment_generating(n)
    return moment
```

Standardmäßig wird für die Berechnung von Momenten Integration verwenden, da dies besser funktioniert. Wir werden anschließend die beiden Methoden in ihrer Geschwindigkeit vergleichen. Sollte man die Berechnung mithilfe der momenterzeugenden Funktion bevorzugen, kann man einfach das Argument `use_integration=False` verwenden.

Satz 3.41 (Verschiebesatz)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit momenterzeugender Funktion* M_X*. Für* $a, b \in \mathbb{R}$ *gilt*

$$M_{aX+b}(t) = \exp(bt) M_X(at) \ .$$

Beweis :
Betrachte

$$\begin{aligned} M_{aX+b}(t) &= \mathbb{E}\left(\exp((aX+b)t)\right) \\ &= \int \exp((aX+b)t)\, \mathrm{d}\mathbb{P} \\ &= \int \exp(aXt+bt)\, \mathrm{d}\mathbb{P} \\ &= \int \exp(aXt)\exp(bt)\, \mathrm{d}\mathbb{P} \\ &= \int \exp(aX(\omega)t)\exp(bt)\, \mathrm{d}\mathbb{P}(\omega) \ . \end{aligned}$$

Da der hintere Faktor nicht von ω abhängt, lässt sich dieser einfach vor das Integral ziehen und es gilt

$$\begin{aligned} &= \exp(bt) \int \exp(aXt)\, \mathrm{d}\mathbb{P} \\ &= \exp(bt) \int \exp(X(at))\, \mathrm{d}\mathbb{P} \\ &= \exp(bt) M_X(at) \ , \end{aligned}$$

was zu zeigen war. □

Wenden wir diesen Satz jetzt auf eine zentrierte und standardisierte Zufallsvariable an, so erhalten wir das folgende

Korollar 3.42 (Zentral- und Standardmomenterzeugende Funktion)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine Zufallsvariable. Weiter seien* μ *und* σ *der Erwartungswert und die Standardabweichung. Für*

$$\overline{X} = X - \mu$$

gilt dann

$$M_{\overline{X}}(t) = \exp(-\mu t) M_X(t) \ .$$

Für

$$\tilde{X} = \frac{X - \mu}{\sigma}$$

gilt dann

$$M_{\tilde{X}}(t) = \exp\left(-\frac{\mu}{\sigma} t\right) M_X\left(\frac{1}{\sigma} t\right) \ .$$

Diese beiden Funktionen können wir dann als zentral- beziehungsweise standardmomenterzeugende Funktionen bezeichnen.

Beweis :
Betrachte zunächst

$$\overline{X} = X - \mu \ .$$

Mit dem Verschiebesatz folgt für $a = 1$ und $b = -\mu$

$$M_{\overline{X}}(t) = \exp(-\mu t) M_X(t) \ .$$

Für

$$\begin{aligned} \tilde{X} &= \frac{X - \mu}{\sigma} \\ &= \frac{X}{\sigma} - \frac{\mu}{\sigma} \end{aligned}$$

folgt ähnlich für $a = 1/\sigma$ und $b = -\mu/\sigma$

$$M_{\tilde{X}}(t) = \exp\left(-\frac{\mu}{\sigma}\right) M_X\left(\frac{t}{\sigma}\right)$$

dank dem obigen Satz. □

Auch zu diesen beiden Funktionen können wir Methoden zur Berechnung implementieren.

Code 3.43 (`standardized_moment_generating_function`)**:** Wir implementieren die zentral- beziehungsweise standardmomenterzeugenden Funktionen wie oben erarbeitet. Da dies für beide Fälle ähnlich abläuft, werden wir dies nur für Letztere zeigen.

```
def standardized_moment_generating_function(self):
    if hasattr(self, 'SMGF'):
        standardized_moment_generating_function = self.SMGF
    else:
        t = sym.Symbol('t', real=True)
        mu = self.mean()
        sigma = self.standard_deviation()
        moment_generating_function = self.moment_generating_function().subs(t, t / sigma)
        standardized_moment_generating_function = sym.exp(- mu / sigma * t) *
    moment_generating_function
        standardized_moment_generating_function = sym.simplify(
    standardized_moment_generating_function)
        self.SMGF = standardized_moment_generating_function
    return standardized_moment_generating_function
```

Als erstes überprüfen wir, wie bei der momenterzeugenden Funktion, ob das Attribut `SMGF` schon belegt ist. Ansonsten lassen wir in einem ersten Schritt Erwartungswert, Standardabweichung und momenterzeugende Funktion berechnen. Nach obigen Satzes bilden wir dann die standard-momenterzeugenden Funktionen und vereinfachen diese. Mithilfe dieser Funktionen können wir nun ebenfalls Zentral- und Standardmomente berechnen. Der Code für `_central_moment_generating` und `_standard_moment_generating` ist analog zu `_moment_generating` und wird daher nicht vorgeführt.

An dieser Stelle betrachten wir eine eng mit der momenterzeugenden Funktion verwandte Funktion, die uns ein weiteres Set an Charakteristika liefert.

Definition 3.44 (Kumulantenerzeugende Funktion und Kumulanten)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit momenterzeugender Funktion* M_X*. Die Kumulantenerzeugende Funktion von* X *(cumulant-generating function) ist definiert durch*

$$\begin{aligned} K_X(t) &:= \log\left(M_X(t)\right) \\ &= \log\left(\mathbb{E}\left(\exp(tX)\right)\right) \ . \end{aligned}$$

Insbesondere ist $K_X : \mathbb{D} \to \mathbb{C} : t \mapsto K_X(t)$ *eine Abbildung. Die* n*-te Kumulante (cumulant) ist dann definiert durch*

$$\kappa_n := \left[\frac{\mathrm{d}^n}{\mathrm{d}t^n} K_X(t)\right]_{t=0} \ .$$

Code 3.45 (`cumulant_generating_function`)**:** Die Implementierung der kumulantenerzeugenden Funktion ist exakt nach obiger Definition

```
def cumulant_generating_function(self):
    if hasattr(self, 'CGF'):
        cumulant_generating_function = self.CGF
    else:
        moment_generating_function = self.moment_generating_function()
        cumulant_generating_function = sym.log(moment_generating_function)
        cumulant_generating_function = sym.simplify(cumulant_generating_function)
        self.CHF = cumulant_generating_function
    return cumulant_generating_function
```

und die der Kumulanten ist analog zu Definition von `_moment_generating`.

Bemerkung 3.46 (Einfache Kumulanten)**:** Gegeben sei eine Zufallsvariable X mit momenterzeugender Funktion M_X. Betrachte nun

$$\begin{aligned} \kappa_1 &= \left[\frac{\mathrm{d}}{\mathrm{d}t} K_X(t)\right]_{t=0} \\ &= \left[\frac{\mathrm{d}}{\mathrm{d}t} \log(M_X(t))\right]_{t=0} \ . \end{aligned}$$

Mit der Kettenregel folgt

$$\begin{aligned} &= \left[\frac{1}{M_X(t)} M_X'(t)\right]_{t=0} \\ &= \frac{1}{M_X(0)} M_X'(0) \ . \end{aligned}$$

Dies ist wohldefiniert, da wie zuvor berechnet $M_X(0) = 1$ ist. Nach der Potenzreihendarstellung der momenterzeugenden Funktion finden wir rechts den Erwatungswert und es gilt

$$= \mathbb{E}(X) \ .$$

Diesen Beweis können wir mit SymPy andeuten. Durch

```
x = sym.Symbol('x', real=True)
f = sym.Function('f')(x)
rv = RandomVariableContinuous(f, x, force_density=True)
first_cumulant = rv.cumulant(1)
mean = rv.mean()
solution = first_cumulant - mean
solution = sym.simplify(solution)
```

erhalten wir `(1 - Integral(f(x), (x, -oo, oo)))*Integral(x*f(x), (x, -oo, oo))/Integral(f(x), (x, -oo, oo))`. Da f eine Dichte ist, wird der vordere Faktor zu Null. Damit ist dann gezeigt, dass Erwartungswert und erster Kumulante gleich sind.

Wir betrachten nun den zweiten Kumulanten

$$\begin{aligned}\kappa_2 &= \left[\frac{\mathrm{d}^2}{\mathrm{d}t^2} K_X(t)\right]_{t=0} \\ &= \left[\frac{\mathrm{d}^2}{\mathrm{d}t^2} \log(M_X(t))\right]_{t=0} .\end{aligned}$$

Mit der Kettenregel folgt

$$= \left[\frac{\mathrm{d}}{\mathrm{d}t} \frac{1}{M_X(t)} M_X'(t)\right]_{t=0} .$$

Mit Produkt- und Kettenregel folgt

$$\begin{aligned}&= \left[-\frac{1}{M_X(t)^2} M_X'(t) M_X'(t) + \frac{1}{M_X(t)} M_X''(t)\right]_{t=0} \\ &= \left[\frac{1}{M_X(t)} M_X''(t) - \frac{1}{M_X(t)^2} M_X'(t)^2\right]_{t=0} \\ &= \frac{1}{M_X(0)} M_X''(0) - \frac{1}{M_X(0)^2} M_X'(0)^2 .\end{aligned}$$

Mit einer ähnlichen Argumentation zu oben folgt

$$= \mathbb{E}(X^2) - \mathbb{E}(X)^2$$

und wir finden

$$= \mathrm{Var}(X) .$$

Dies lässt sich leider nicht sinnvoll mit SymPy „beweisen“.

Wir berechnen nun einige kumulantenerzeugende Funktionen.

Beispiel 3.47 (Kumulantenerzeugende Funktionen)**:** Wir verwenden die Zufallsvariablen, für die wir schon die momenterzeugende Funktion berechnet haben.

(i) Sei $X \sim \mathrm{Ber}(p)$ mit $p \in (0,1)$ Bernoulli-verteilt. Die momenterzeugende Funktion ist dann

$$M_X(t) = (1-p) + p\exp(t) .$$

Damit ergibt sich die kumulantenerzeugende Funktion durch

$$\begin{aligned}K_X(t) &= \log(M_X(t)) \\ &= \log\left((1-p) + p\exp(t)\right) .\end{aligned}$$

Durch

```
p = sym.Symbol('p', real=True, positive=True)
n = sym.Symbol('n', integer=True, nonnegative=True)
density = {1: p, 0: 1 - p}
rv = RandomVariableFinite(density, n)
cumulant_generating_function = rv.cumulant_generating_function()
```

erhalten wir ebenfalls `log(p*exp(t) - p + 1)`.

(ii) Sei $X \sim \mathrm{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Die momenterzeugende Funktion ist

$$M_X(t) = \frac{\lambda}{\lambda - t}$$

Für die kumulantenerzeugende Funktion gilt dann

$$\begin{aligned} K_X(t) &= \log(M_X(t)) \\ &= \log\left(\frac{\lambda}{\lambda - t}\right) \\ &= \log(\lambda) - \log(\lambda - t) \ . \end{aligned}$$

Mittels

```
x = sym.Symbol('x', real=True)
lamda = sym.Symbol('lambda', real=True, positive=True)
density = lamda * sym.exp(- lamda * x)
rv = RandomVariableContinuous(density, x, [sym.Integer(0), sym.oo])
cumulant_generating_function = rv.cumulant_generating_function()
```

erhalten wir ebenso `log(lambda/(lambda - t))`.

(iii) Sei nun $X \sim \mathrm{N}(\mu, \sigma)$ mit $\mu \in \mathbb{R}$ und $\sigma > 0$ normalverteilt. Die momenterzeugenden Funktion ist

$$M_X(t) = \exp\left(\mu t + \frac{\sigma^2}{2} t^2\right) \ .$$

Damit ergibt sich die folgende kumulantenerzeugende Funktion

$$\begin{aligned} K_X(t) &= \log(M_X(t)) \\ &= \log\left(\exp\left(\mu t + \frac{\sigma^2}{2} t^2\right)\right) \\ &= \mu t + \frac{\sigma^2}{2} t^2 \ . \end{aligned}$$

Durch

```
x = sym.symbols('x', real=True)
mu = sym.symbols('mu', real=True)
sigma = sym.symbols('sigma', real=True, positive=True)
density = 1 / (sigma * sym.sqrt(2 * sym.pi)) * sym.exp(- (x - mu)**2 / (2 * sigma**2))
rv = RandomVariableContinuous(density, x)
cumulant_generating_function = rv.cumulant_generating_function()
```

erhalten wir `t*(2*mu + sigma**2*t)/2`, was dasselbe ist. Es ist interessant, dass SymPy sich für diese doch etwas komplizierte Vereinfachung entscheidet. Verwenden wir nun

```
cumulant_generating_function = sym.expand(cumulant_generating_function)
```

so erhalten wir `mu*t + sigma**2*t**2/2` wie oben.

Auf die Berechnung von Kumulanten dieser Zufallsvariablen wollen wir verzichten. Die entsprechende Methode ist unter `cumulant` zu finden. Sie funktioniert analog zu `_moment_generating`.

3.4 Laufzeitvergleiche

In diesem Kapitel wollen wir die Dauer der Berechnung der Momente mit verschiedenen Methoden vergleichen. Für alle Berechnung wurden möglichst gleiche Voraussetzungen geschaffen, indem möglichst alle Programme im Hintergrund geschlossen wurden.

Beispiel 3.48 (Laufzeit Exponentialverteilung)**:** Sei $X \sim \text{Exp}(\lambda)$ mit $\lambda > 0$ exponentialverteilt. Der folgende Graph stellt die Dauer für die Berechnung der ersten hundert Momente dar.

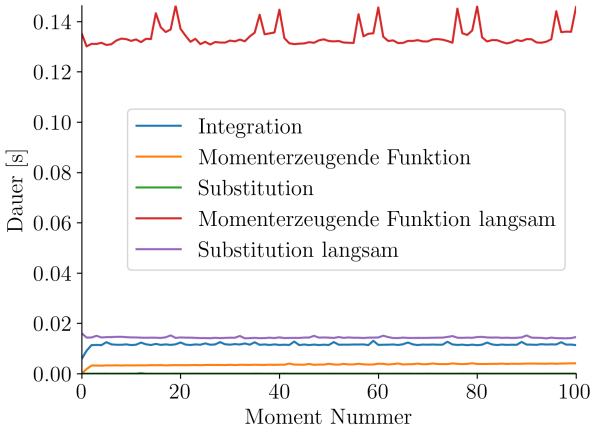

Abb. 16: Dauer der Berechnung der ersten hundert Momente einer Exp(λ)-Verteilung

Für die Erstellung der Daten lassen wir die ersten hundert Momente hundertmal berechnen. Für Integration verwenden wir die `_moment_integration-` und für momenterzeugende Funktion die `_moment_generating`-Methode. Für die langsam-Kurven wird vor jeder Berechnung das `MGF`-Attribut gelöscht. Somit muss dort jedes Mal die momenterzeugende Funktion neu berechnet werden. Für Substitution wird zu Beginn (ohne Messung der Zeit) das n-te Momente berechnet und anschließend für n der entsprechende Wert eingesetzt. Im langsamen Fall wird das n-te Moment für jede Iteration von neuem berechnet.

Interessant sind die Spitzen bei $[15, 19]$, $[36, 40]$, $[56, 60]$ sowie $[76, 80]$ und $[96, 100]$. Dies dauert circa 10% länger, als die anderen Berechnungen. Es kann nicht am Ableiten liegen, da sonst die ähnliche Doppelspitzen auch in der Kurve für die optimierte Berechnung mit der momenterzeugenden Funktion zu finden wären. Der Garbage Collector kann mit ziemlicher Sicherheit ausgeschlossen werden, da nach jeder Berechnung der Inhalt der Variable mit `del` gelöscht wird. Verwenden wir einen anderen Computer, so erhalten wir

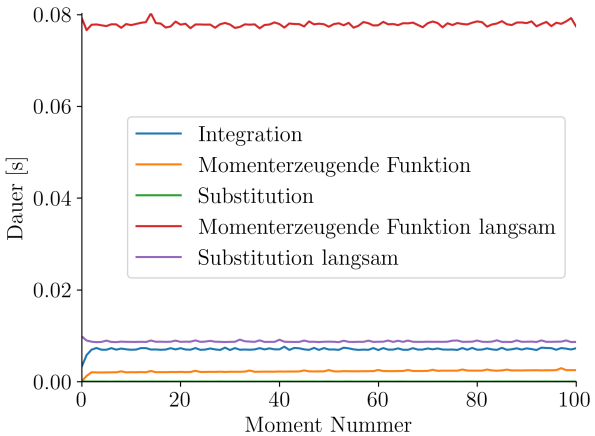

Abb. 17: Dauer der Berechnung der ersten hundert Momente einer Exp(λ)-Verteilung

An dieser Grafik lässt sich feststellen, dass das erste System um etwa die Hälfte langsamer ist als das zweite. Zudem treten die angesprochenen Spitzen nicht erneut auf. Daraus lässt sich schließen, dass diese Anomalie vermutlich nichts mit dem Programm selbst zu tun hat.

Für die folgende Diskussion werden wir uns mit den ersten Daten beschäftigen. Um den Graphen zu vereinfachen, ist die folgende Grafik linear interpoliert.

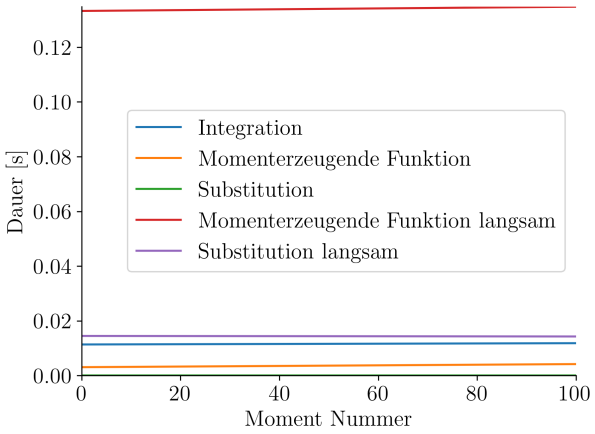

Abb. 18: Interpolierte Dauer der Berechnung der ersten hundert Momente einer Exp(λ)-Verteilung

Betrachten wir den Unterschied zwischen den beiden Geraden für die momenterzeugende Funktion, so erhalten wir eine Dauer von 0.1310 Sekunden. Lassen wir die momenterzeugende Funktion eintausendmal berechnen, so erhalten wir im Mittel einen Wert von 0.1362 Sekunden. Dies stimmt also mit obigem Abstand überein. Da die Berechnung der momenterzeugende Funktion scheinbar eine „teure“ Berechnung ist, ist es sinnvoll die momenterzeugende Funktion, falls sie schon berechnet wurde, in dem Attribut `MGF` zu speichern. Dies findet analog für die kumulantenerzeugende Funktion unter `CGF` und die im folgenden Kapitel vorgestellt charakteristische Funktion unter `CF` statt.

Für diese Zufallsvariable macht es scheinbar keinen großen Unterschied, mit welcher Methode man die Momente berechnet. Es ist allerdings sehr interessant, dass die Berechnung der Momente mittels Substitution, falls zuvor das n-te Moment berechnet quasi instantan stattfindet. Man könnte sich an dieser Stelle wundern, wieso es keine Methode gibt, die ein Moment mittels Substitution berechnet. Die Berechnung des n-ten Moments ist für die meisten Verteilungen höchst aufwendig und für viele Verteilungen kann SymPy auch keinen geschlossenen Ausdruck angeben. Des Weiteren haben wir, wie bei dem Beispiel zur nicht-integrierbaren Familie gesehen, auch ein Problem, falls das n-te Moment eine stückweise Funktion ist. Es fällt auf, dass die Berechnung mithilfe der momenterzeugenden Funktion um circa 0.008 Sekunden schneller ist, als die Berechnung mittels Integration. Daher scheint es momentan eher unsinnvoll zu sein die Integration als Standardmethode gewählt zu haben. Weiter fällt auf, dass alle Methoden nahezu konstanten Aufwand haben. Die Steigung ist jeweils im Bereich von 10^{-6}, was vermutlich auf Messungenauigkeiten zurückzuführen ist.

Beispiel 3.49 (Laufzeit Normalverteilung)**:** Sei $X \sim \mathrm{N}(\mu, \sigma)$ mit $\mu, \sigma > 0$ normalverteilt. Da die Berechnungen hier deutlich mehr Zeit in Anspruch nehmen, lassen wir die ersten hundert Momente nur zehnmal berechnen und verzichten auf die unoptimierten Varianten der Berechnungen.

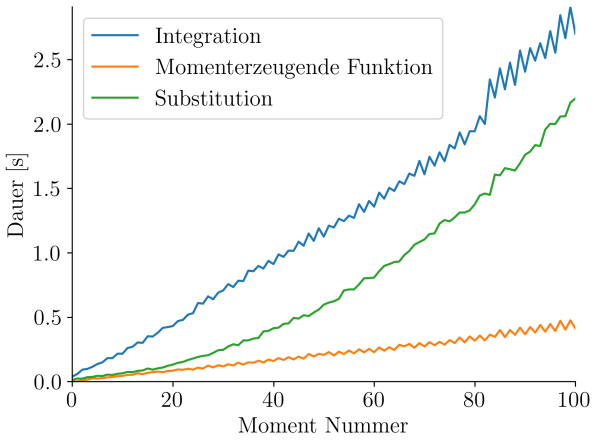

Abb. 19: Dauer der Berechnung der ersten hundert Momente einer $\mathrm{N}(\mu, \sigma)$-Verteilung ohne Vereinfachung

Dieser Plot spiegelt aber nicht ganz die Wahrheit wieder. Lassen wir mit SymPy das n-te Moment berechnen, so erhalten wir

$$\mathbb{E}(X^n) = \frac{2^{\frac{n}{2}-1}\sigma^n \left((-1)^n \, G^{2,1}_{1,2}\left(\begin{matrix} \frac{1}{2} - \frac{n}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{\mu^2}{2\sigma^2} \right) + G^{2,1}_{1,2}\left(\begin{matrix} \frac{1}{2} - \frac{n}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{\mu^2 e^{2i\pi}}{2\sigma^2} \right) \right) e^{-\frac{\mu^2}{2\sigma^2}}}{\pi} \, .$$

Setzen wir beispielsweise den Wert $n = 6$ ein, so erhalten wir

$$\mathbb{E}(X^6) = \frac{2\sigma^4 \left(G^{2,1}_{1,2}\left(\begin{matrix} -\frac{3}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{\mu^2}{2\sigma^2} \right) + G^{2,1}_{1,2}\left(\begin{matrix} -\frac{3}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{\mu^2 e^{2i\pi}}{2\sigma^2} \right) \right) e^{-\frac{\mu^2}{2\sigma^2}}}{\pi} \, .$$

Wir wollen eigentlich den Ausdruck

$$\mathbb{E}(X^6) = \mu^6 + 15\mu^4\sigma^2 + 45\mu^2\sigma^4 + 15\sigma^6 \, .$$

Diesen erhalten wir durch die Anwendung von `simplify` auf obigen Ausdruck. Verwenden wir diesen Schritt auch in der obigen Berechnung, so erhalten wir das folgende Bild.

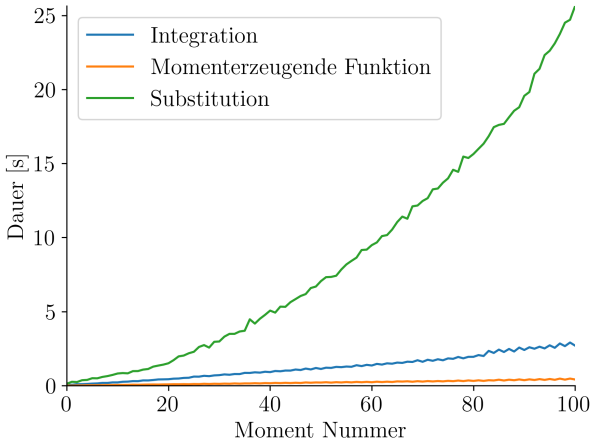

Abb. 20: Dauer der Berechnung der ersten hundert Momente einer $\mathrm{N}(\mu, \sigma)$-Verteilung mit Vereinfachung

Wir erkennen, dass Substitution für eine Normalverteilung aufgrund der nötigen, scheinbar sehr kostspieligen Vereinfachung ein grauenhaft ineffizientes Verfahren ist. Bei der Exponentialverteilung haben wir diese Vereinfachung nicht benötigt, da SymPy selbst große Fakultäten sofort berechnen kann. Wir erkennen weiter, dass der Aufwand zur Berechnung der Moment nicht mehr konstant. Für die Substitution scheint er quadratisch zu sein und für die anderen Methoden linear. Interessant ist auch, dass die Berechnung mittels der momenterzeugenden Funktion deutlich schneller abläuft als die Berechnung mittels Integration. Verwenden wir eine lineare Interpolation, so erhalten wir eine Steigung von 0.004 beziehungsweise 0.240.

Wir werden nun als Beispiel einer der einfachsten stetigen Verteilungen betrachten.

Beispiel 3.50 (Laufzeit stetige Gleichverteilung)**:** Sei X auf $[a, b]$ gleichverteilt mit $a < b$ aus $\mathbb{R}$. Lassen wir an dieser Stelle die ersten hundert Moment fünfzigmal mit den drei Methoden berechnen, so erhalten wir den folgenden Plot.

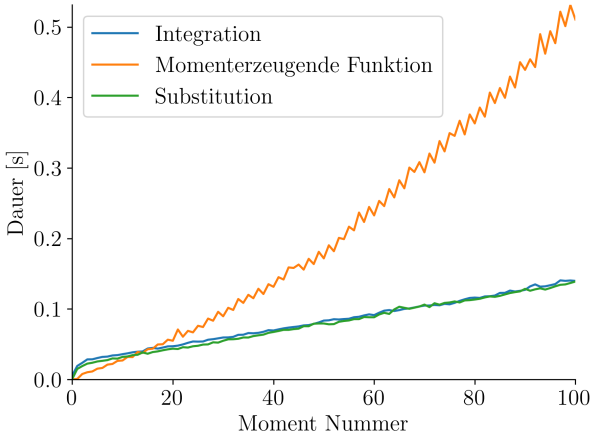

Abb. 21: Dauer der Berechnung der ersten hundert Momente einer Gleichverteilung auf $[a, b]$

Wir erkennen nun etwas scheinbar Überraschendes. Die Berechnung mittels der momenterzeugenden Funktion ist langsamer. Die Berechnung mittels Integration und Substitution scheinen wieder lineares Wachstum zu haben und sind interessanterweise gleich schnell. Die Berechnung mit der momenterzeugenden Funktion scheint quadratischen Aufwand zu haben.

Nun werden wir uns ein finites Beispiel anschauen.

Beispiel 3.51 (Laufzeit Bernoulli-Verteilung)**:** Sei $X \sim \text{Ber}(p)$ mit $p \in (0, 1)$ Bernoulli-verteilt. Lassen wir die ersten hundert Momente tausendmal berechnen, so erhalten wir die folgende Grafik.

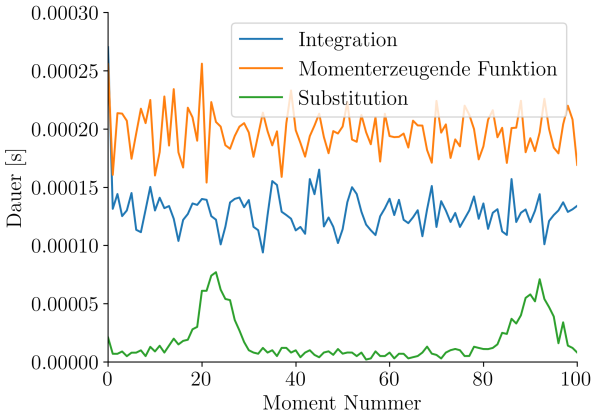

Abb. 22: Dauer der Berechnung der ersten hundert Momente einer Ber(p)-Verteilung

Dieses Beispiel ist eher zum Spaß gedacht. Wir sehen, dass die Dauer der Berechnung im Zehntel Millisekundenbereich liegt. Diese unglaubliche Geschwindigkeit lässt sich darauf zurückführen, dass hier nur simpelste Berechnungen durchgeführt werden müssen. Es muss null oder eins potenziert werden und anschließend mit $1 - p$ respektive p multipliziert werden. Aufgrund dieser Geschwindigkeit sind die Daten auch so rauschend, da die gesamte Berechnung nur wenige Sekunden gedauert hat und somit kleinste Nebentätigkeiten des Betriebssystem große Schwankungen hervorrufen. Vermutlich ist die Berechnung der Momente aller finiten Verteilungen derart „einfach“, da hier wie gesagt nur die Grundoperationen benötigt werden und diese sehr gut optimiert sind.

Als letztes Beispiel werden wir eine diskrete Zufallsvariable betrachten.

Beispiel 3.52 (Laufzeit Poisson-Verteilung)**:** Sei $X \sim \text{Poiss}(\lambda)$ mit $\lambda > 0$ Poisson-verteilt. Hier lassen wir nur die ersten zwanzig Moment zehnmal berechnen. Wir erhalten den folgenden Plot.

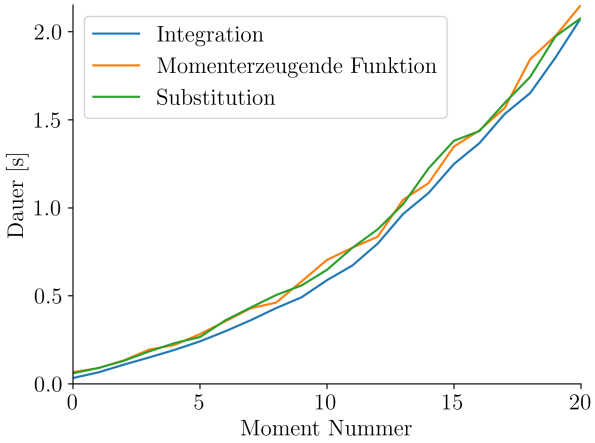

Abb. 23: Dauer der Berechnung der ersten zwanzig Momente einer Poiss(λ)-Verteilung

Interessanterweise sind hier alle drei Arten der Berechnung ziemlich genau gleich schnell und zeigen mindestens quadratisches Wachstum. Vermutlich sind alle Berechnungen ungefähr gleich schnell, da SymPy für Summen beziehungsweise Reihen keine so starken Werkzeuge hat, wie für Integrale. Dies ist vermutlich auch der Grund, wieso schon das zwanzigste Moment über 2 Sekunden Rechenzeit benötigt.

Man stellt sich jetzt natürlich die Frage, was wir aus all diesen Beispielen gelernt haben. Es ist leider nicht klar, welche der Methoden „die Beste“ ist. Je nach Verteilung ist eine andere Methode am schnellsten. Außerdem haben wir gelernt, dass finite Verteilungen blitzschnell berechenbar sind und diskrete Verteilungen scheinbar größere Probleme machen, wobei dies noch etwas näher beleuchtet werden sollte. Bei stetigen Verteilungen ist von der Rechendauer alles dabei.

3.5 Charakteristische Funktionen

In diesem Abschnitt wollen wir uns mit einer weiteren Funktion beschäftigen, die sehr eng mit der momenterzeugenden Funktion verwandt ist.

Definition 3.53 (Charakteristische Funktion)**:** *Sei $(\Omega, \mathscr{A}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und X eine reelle Zufallsvariable. Die charakteristische Funktion von X (characteristic function) ist definiert durch*

$$C_X(t) := \mathbb{E}\left(\exp(itX)\right) \ .$$

Insbesondere ist $C_X : \mathbb{D} \to \mathbb{C} : t \mapsto C_X(t)$ für $\mathbb{D} := \{t \in \mathbb{R} \mid C_X(t) < \infty\}$ eine komplexwertige Abbildung.

Mit dem folgenden Satz werden wir einen einfachen Zusammenhang zwischen momenterzeugender und charakteristischer Funktion erkennen.

Satz 3.54 (Zusammenhang charakteristische und momenterzeugende Funktion)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit momenterzeugender Funktion* M_X*. Es gilt für* $t \in \mathbb{D}$

$$M_X(it) = C_X(t) \ .$$

Beweis :
Betrachte zu $t \in \mathbb{D}$

$$\begin{aligned} M_X(it) &= \mathbb{E}\left(\exp(itX)\right) \\ &= C_X(t) \ . \end{aligned}$$

Wir können an dieser Stelle wieder einen „Beweis“ mit SymPy führen. Wir formen die zu beweisende Identität um und erhalten

$$0 = M_X(it) - C_X(t) \ .$$

Wir betrachten nun.

```
x = sym.Symbol('x', real=True)
t = sym.Symbol('t', real=True)
f = sym.Function('f')(x)
rv = RandomVariableContinuous(f, x, force_density=True)
moment_generating_function = rv.moment_generating_function()
characteristic_function = rv.characteristic_function()
solution = moment_generating_function.subs(t, sym.I * t) - characteristic_function
```

Mit dem Ergebnis von `0`, was zu zeigen war. □

Wir können nun auch die charakteristische Funktion verwenden, um Momente zu bestimmen.

Satz 3.55 (Momente mit charakteristischer Funktion)**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable. Existiert das* n*-te Momente, so gilt*

$$\mathbb{E}(X^n) = \left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}\frac{C_X(t)}{i^n}\right]_{t=0} \ .$$

Beweis :
Betrachte mit vorigem Satz

$$\left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}\frac{C_X(t)}{i^n}\right]_{t=0} = \left[\frac{\mathrm{d}^n}{\mathrm{d}t^n}\frac{M_X(it)}{i^n}\right]_{t=0} \ .$$

Nach der Kettenregel erhalten wir durch das Ableiten einen Faktor von i^n. Da wir $t = 0$ einsetzen, ist die Multiplikation mit i in der momenterzeugenden Funktion irrelevant und wir erhalten aus dem Korollar über die Berechnung der Momente mithilfe der momenterzeugenden Funtion

$$\begin{aligned} &= \frac{i^n}{i^n}\mathbb{E}(X^n) \\ &= \mathbb{E}(X^n) \end{aligned}$$

mittels Kürzen. □

Diese Methode wollen wir allerdings nicht weiter verwenden, da sie extrem ähnlich zur Berechnung mithilfe der momenterzeugenden Funktion ist, aber keinerlei Vorteile hat.

4 Simulationen

In diesem Kapitel werden wir uns mit der Simulation von Zufallsvariablen beschäftigen. Hierzu benötigen wir zunächst die folgende

Definition 4.56 (Inverse Verteilungsfunktion) [NP21]**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit Verteilungsfunktion* F_X*. Zu* $p \in [0,1]$ *ist das* p*-Quantil definiert durch*

$$F_X^{-1}(p) := \inf\{x \in \mathbb{R} \mid F_X(x) \geq p\} \ .$$

Damit erhält man die inverse Verteilungsfunktion

$$F_X^{-1} : [0,1] \to \mathbb{R} : p \mapsto F_X^{-1}(p) \ .$$

Mit dieser Definition können wir nun den Satz formulieren und beweisen, der die Grundlage unserer Simulationsmethode bilden wird.

Satz 4.57 (Inversionsmethode) [NP21]**:** *Sei* $(\Omega, \mathscr{A}, \mathbb{P})$ *ein Wahrscheinlichkeitsraum und* X *eine reelle Zufallsvariable mit Verteilungsfunktion* F_X*. Sei weiter* U *auf* $[0,1]$ *stetig gleichverteilt. Dann ist* $F_X^{-1}(U)$ *wieder eine reelle Zufallsvariable mit der gleichen Verteilung wie* X*.*

Beweis [NP21]*:*
Nach Definition der inversen Verteilungsfunktion gilt insbesondere

$$F_X^{-1}(U) \leq x$$

genau dann, wenn

$$U \leq F_X(x) \ .$$

Betrachte zu $x \in \mathbb{R}$

$$\mathbb{P}(F_X^{-1}(U) \leq x) = \mathbb{P}(U \leq F_X(x)) \ .$$

Da U auf $[0,1]$ gleichverteilt ist, gilt nach dem Beispiel zur Berechnung von Verteilungsfunktionen

$$= F_X(x) \ ,$$

denn $F_X(x) \in [0,1]$ nach den Eigenschaften der Verteilungsfunktion. □

Diese Methode wollen wir gleich programmieren.

Code 4.58 (`simulate`)**:** Aufgrund der unterschiedlichen Arten von Verteilungsfunktionen müssen wir diese Methode wieder für jeden Typen getrennt implementieren.

(i) Für finite Zufallsvariablen gilt

```
def simulate(self, number):
    simulate = []
    uni = np.random.uniform(0, 1, number)
    distribution_function = self.distribution_function()
    for num in uni:
        for key in distribution_function.keys():
            if num <= float(distribution_function[key]):
                simulate.append(float(key))
                break
    return simulate
```

Der erste Teil läuft für alle Typen gleich ab. Wir erzeugen mit NumPy ein Array von auf $[0, 1]$ gleichverteilten Zufallszahlen. Die Anzahl muss als Argument übergeben werden. Wir lassen uns anschließend die Verteilungsfunktion berechnen. Nun iterieren wir über alle gleichverteilten Werte und anschließend über alle kumulierten Wahrscheinlichkeiten, die unsere Verteilungsfunktion annimmt. Sobald wir das erste Mal mit dem gleichverteilten Wert kleiner sind, als die kumulierten Wahrscheinlichkeit, fügen wir den zugehörigen x-Wert der Verteilungsfunktion dem Simulationsergebnis hinzu und brechen die innere Schleife ab. Diese Methode funktioniert, da die x-Werte der Verteilungsfunktion im Dictionary sortiert sind und wir so genau das Infimum finden.

(ii) Für diskrete Zufallsvariablen gilt

```
def simulate(self, number, n_max=100):
    simulate = []
    n_list = [n for n in range(n_max) if n >= self.supp[0] and n <= self.supp[1]]
    uni = np.random.uniform(0, 1, number)
    for num in uni:
        cumulative_probability = sym.Integer(0)
        for n in n_list:
            cumulative_probability += self.density.subs(self.variable, n)
            if num <= cumulative_probability:
                simulate.append(n)
                break
    return simulate
```

Die Simulation für diskrete Zufallsvariablen läuft ganz ähnlich ab. Als zusätzliches Argument haben wir `n_max`. Diese Zahl bestimmt, bis zu welchem Wert die kumulierte Wahrscheinlichkeit berechnet wird. Wir bilden zuerst eine Liste der ersten `n_max` Zahlen, die in unserem Träger liegen. Anschließend iterieren wir wieder über jede der gleichverteilten Zahlen. Als nächstes durchlaufen wir die Liste der ersten `n_max` Zahlen und bilden jeweils die kumulierte Wahrscheinlichkeit. Falls die kumulierte Wahrscheinlichkeit das erste Mal größer ist, als der gleichverteilte Wert, fügen wir das `n` der Simulationsliste hinzu und brechen die innere Schleife ab. Da wir uns von unten an den gleichverteilten Wert herantasten, finden wir auf jeden Fall das Infimum. Sollte der gleichverteilte Wert sehr nahe an eins liegen, sorgt das `n_max` dafür, dass die Schleife auf jeden Fall terminiert.

(iii) Für stetige Zufallsvariablen gilt

```
def simulate(self, number):
    simulate = []
    uni = np.random.uniform(0, 1, number)
    t = sym.Symbol('t', real=True)
    distribution_function = self.distribution_function()
    mean = self.mean()
    for num in uni:
        eq = sym.Eq(distribution_function, num)
        sim = sym.nsolve(eq, t, mean)
        simulate.append(float(sim))
    return simulate
```

Diese Simulation läuft etwas anders ab. Wir lassen uns neben der Verteilungsfunktion noch den Erwartungswert berechnen. Somit muss der Erwartungswert endlich sein. Trotz dieser deutlichen Einschränkung ist diese Herangehensweise sehr sinnvoll, da alle Werte der Zufallsvariable um den Erwartungswert verteilt sind. Würden wir einen bestimmten Wert, wie zum Beispiel Null wählen, so wäre eine Normalverteilung mit $\mu = 10^6$ sehr schwierig zu simulieren, da der Abstand zwischen Startwert und gesuchtem Wert zum einen sehr groß ist und zum anderen die Steigung sehr klein wäre.

Im nächsten Schritt der Simulation lösen wir für jeden Wert u der Gleichverteilung die Gleichung

$$F(x) = u$$

nach x auf. Symbolisch lässt sich diese Gleichung meistens leider nicht lösen. Als Startwert für die numerische Lösung verwenden wir den Erwartungswert. Dieser Algorithmus kann im stetigen Falle theoretisch für Verteilungsfunktionen, die nicht streng monoton wachsend sind, schiefgehen. Dies liegt daran, dass die numerische Approximation möglicherweise nicht den richtigen Wert „findet“. Hierzu werden wir gleich ein Beispiel betrachten.

An dieser Stelle sei erwähnt, dass die oben implementierte Simulationsmethode keineswegs effizient implementiert ist. Wie an den meisten Stellen ist es hier leider nicht möglich große Laufzeitvorteile durch Vektorisierung zu erhalten, da es für viele Verteilungen nicht die Möglichkeit gibt die SymPy-Funktionen in beispielsweise NumPy-Funktionen zu übersetzen, um dort effizienter zu arbeiten.

Nun betrachten wir eine Verteilungsfunktion, die vermutlich Probleme machen würde.

Beispiel 4.59 (Nicht-simulierbare Zufallsvariable)**:** Sei X eine reelle Zufallsvariable mit der folgenden Verteilungsfunktion

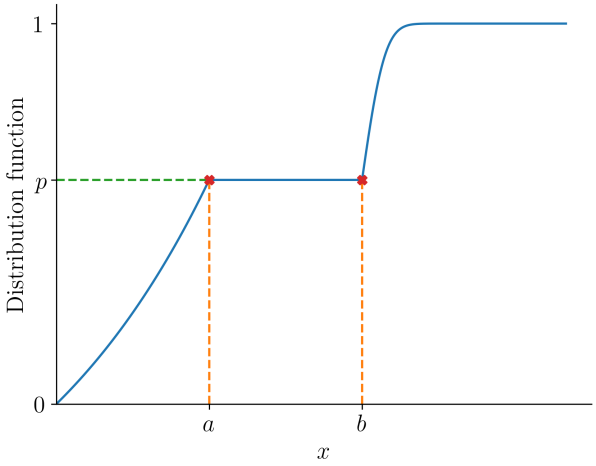

Abb. 24: Verteilungsfunktion einer vermutlich schlecht simulierbaren Verteilung

Die Verteilungsfunktion ist auf $[a, b]$ nur monoton. Ist $U = p$, so ist

$$\begin{aligned} F^{-1}(p) &= \inf\{x \in \mathbb{R} \mid F(x) \geq p\} \\ &= \inf\{x \in [a, \infty)\} \\ &= a \ . \end{aligned}$$

Die numerische Methode hingegen versucht ein $x \in \mathbb{R}$ zu finden mit $F(x) = p$. Ist $\mathbb{E}(X) > b$, so findet die numerische Methode vermutlich zuerst den Wert b, womit wir dann ein falsches Ergebnis erhalten. In den allermeisten Fällen sollte dies jedoch gut gehen.

Nun können wir einige Beispiele untersuchen, für die die Methode wunderbar funktioniert.

Beispiel 4.60 (Klassische Zufallsvariablen)**:** Um die Richtigkeit unserer Ergebnisse zu „überprüfen“, können wir unseren Code mit den entsprechenden Funktionen von NumPy vergleichen. Wir verwenden eine Bernoulli-Verteilung als finites, eine Poisson-Verteilung als diskretes, sowie eine Normal- und Exponentialverteilung als stetige Beispiele.

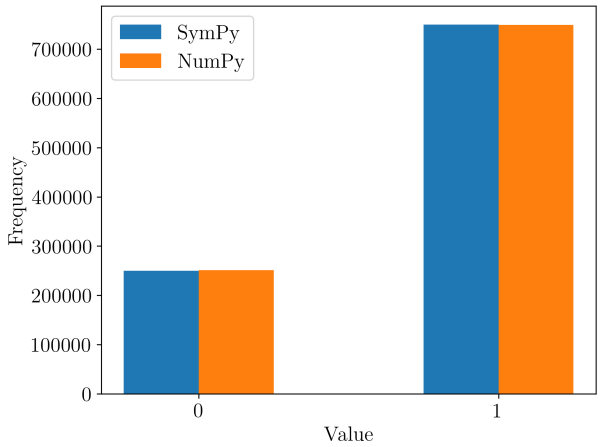

Abb. 25: Simulation einer Ber(3/4)-Verteilung

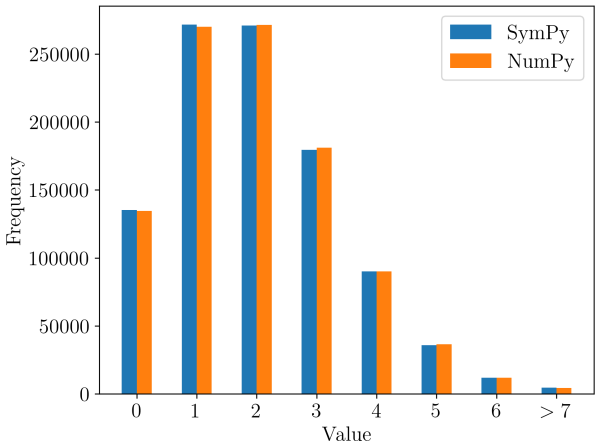

Abb. 26: Simulation einer Poiss(2)-Verteilung

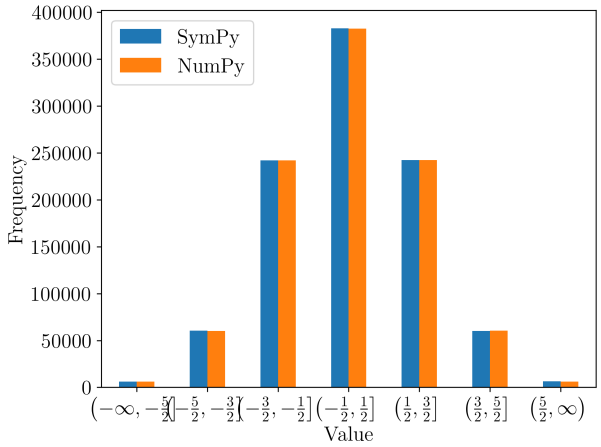

Abb. 27: Simulation einer $\mathrm{N}(0, 1)$-Verteilung

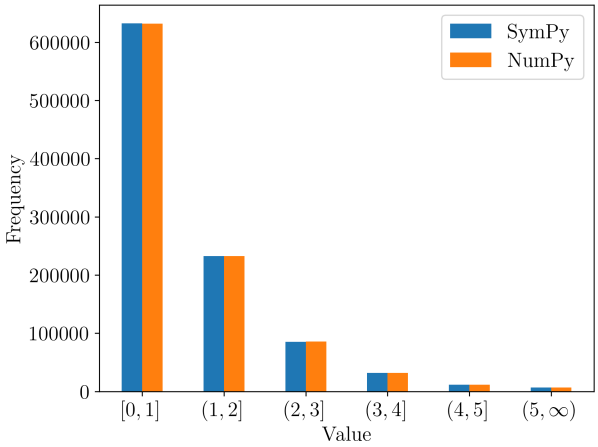

Abb. 28: Simulation einer Exp(3)-Verteilung

Vergleicht man jeweils die Höhe benachbarter Balken miteinander, so scheint es, als würde der von uns programmierte Code genau das tun, was er soll.

Als großen Vorteil gegenüber NumPy können wir die folgenden Beispiele betrachten.

Beispiel 4.61 (Nicht-klassische Zufallsvariablen)**:** Wir werden nun Zufallsvariablen simulieren, die nicht in NumPy implementiert sind und zu denen wir nur die Dichte haben.

(i) Wir können an dieser Stelle das Münzwurfbeispiel simulieren.

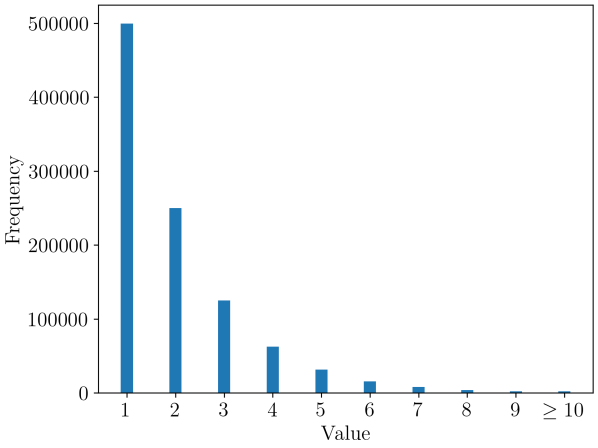

Abb. 29: Simulation einer Verteilung mit Dichte $\sim 2^{-n}$

Der höchste Wert der Zufallsvariable lag bei 27. Dieses Ereignis hat eine Wahrscheinlichkeit von

$$\begin{aligned}\mathbb{P}(X = 27) &= 2^{-27} \\ &= 7.4506 \cdot 10^{-9} \ .\end{aligned}$$

(ii) Gegeben sei einer Verteilung mit der Dichtefunktion

$$\varphi(x) = \left(-\frac{3}{4}x^2 + \frac{3}{4}\right) \mathbb{1}_{[-1,1]}(x) \ .$$

Wir können auf einfache Weise die Normiertheit nachrechnen mittels

$$\begin{aligned}\mathbb{P}_X(\mathbb{R}) &= \int_{\mathbb{R}} \left(-\frac{3}{4}x^2 + \frac{3}{4}\right) \mathbb{1}_{[-1,1]}(x)\,\mathrm{d}x \\ &= \int_{-1}^{1} -\frac{3}{4}x^2 + \frac{3}{4}\,\mathrm{d}x \\ &= \left[-\frac{1}{4}x^3 + \frac{3}{4}x\right]_{-1}^{1} \\ &= -\frac{1}{4}\cdot 1^3 + \frac{3}{4}\cdot 1 + \frac{1}{4}\cdot(-1)^3 - \frac{3}{4}\cdot(-1) \\ &= -\frac{1}{4} + \frac{3}{4} - \frac{1}{4} + \frac{3}{4} \\ &= 1 \ .\end{aligned}$$

Die Nullstellen von φ sind genau an den Grenzen des Trägers, womit die Dichte überall nicht-negativ ist. Wir können nun diese Zufallsvariable simulieren und würden etwas ähnliches, wie bei der Normalverteilung erwarten, nur dass die Werte auf $[-1, 1]$ beschränkt sind.

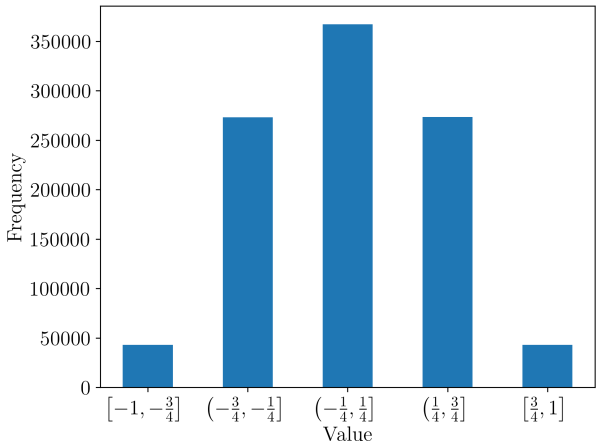

Abb. 30: Simulation einer Verteilung mit Dichte $\sim -x^2$

(iii) Als letztes Beispiel können wir die mit der Normalverteilung verwandte, platykurtische Verteilung simulieren. Wir würden wieder etwas Ähnliches zur Normalverteilung erwarten, nur dass die Enden schneller abfallen und mehr Ereignisse um die Null zentriert sind.

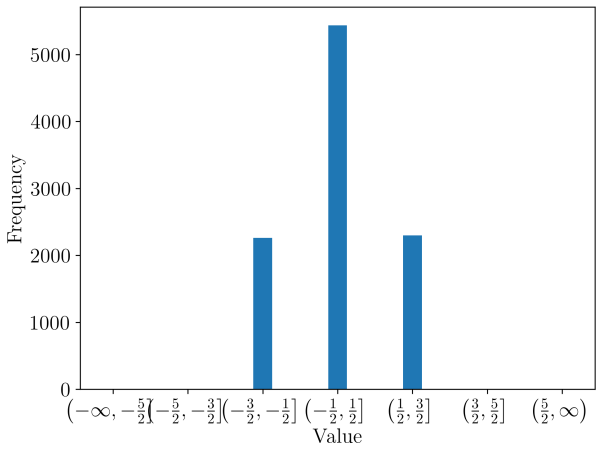

Abb. 31: Simulation einer Verteilung mit Dichte $\sim \exp(-x^4)$

Da die gleiche Aufteilung auf der x-Achse gewählt wurde, erkennen wir genau das Vorhergesagte.

Wir wollen es bei diesen Beispielen belassen. Insbesondere in diesem Kapitel sind der Phantasie keine Grenzen gesetzt und ich möchte Sie dazu anregen, das Programm selbst auszuprobieren.

5 Mögliche Erweiterungen

In diesem letzten Kapitel werden wir uns mit einigen möglichen Erweiterungen beschäftigen.

Eine wichtige Verallgemeinerung wäre es auf jeden Fall diese Bibliothek auf d-dimensionale Zufallsvektoren zu erweitern. Dies war anfangs auch das Ziel, doch leider stoß da SymPy auf einige Probleme. Ein großer Teil aller besprochenen Funktionen hat ohne Probleme auch für mehrdimensionale Zufallsvariablen funktioniert. Entsprechend wird beispielsweise der Erwartungswert zum Erwartungsvektor. Als großes Problem stellt sich dabei heraus, dass die Berechnung nur für Zufallsvektoren mit unabhängigen Einträgen funktionierte. Als erstes wurde eine zweidimensionale Multinormalverteilung versucht und SymPy hat ohne Ergebnis oder Fehler eine gute Stunde versucht den Erwartungswert zu berechnen. Ersetzen von μ_1 und μ_2 durch konkrete Zahlen erlaubte es SymPy nach langer Rechnung zumindest numerisch den Erwartungsvektor ungefähr zu berechnen. Für das nächste Beispiel wurde $X \sim \mathrm{Exp}(\lambda)$ und $Y \sim \mathrm{N}(\mu, \sigma)$ gewählt und der Zufallsvektor $(X, X+Y)^\top$ betrachtet. Auch hier rechnete SymPy ergebnislos für sehr lange Zeit. Da nicht einmal die Berechnung des Erwartungswertes für diese doch ziemlich einfachen Beispiele funktionierte, wurde dieser Teil nicht weiter bearbeitet.

Eine weitere nützliche Erweiterung wäre es zur Definition eines `RandomVariable`-Objekts nicht mehr eine Dichte zu fordern. Man könnte also versuchen dieses Objekt mit der Verteilungs- oder charakteristischen Funktion zu definieren. Man müsste dann entsprechend die Methoden anpassen, dass diese in der Lage sind entsprechende Charakteristika auf anderen Wegen zu berechnen. Also beispielsweise

$$\mathbb{E}(X) = \int_0^\infty 1 - F(x)\,\mathrm{d}x - \int_{-\infty}^0 F(x)\,\mathrm{d}x$$

mit Verteilungsfunktion oder

$$\mathbb{E}(X) = \left[\frac{\mathrm{d}}{\mathrm{d}t}\frac{C_X(t)}{i}\right]_{t=0}$$

mit charakteristischer Funktion. Außerdem könnte man damit eine deutlich größere Klasse an Zufallsvariablen simulieren.

Eine weiter schöne Erweiterung wäre es, die Python-Operatoren für Addition, Subtraktion, Multiplikation und Division zu implementieren. Damit könnte man dann zwei `RandomVariable`-Objekte addieren, um daraus ein Neues zu erhalten. Da die Dichte der Summe von Zufallsvariablen deren Faltung ist, müsste man eine entsprechende Faltungsmethode definieren. Damit könnte man dann vielleicht entsprechende Sätze über die Summe von unabhängigen (identisch verteilten) Zufallsvariablen „zeigen“ oder eine Brücke zu stochastischen Prozessen schlagen.

Umgekehrt gäbe es die Möglichkeit einer Spezialisierung. Man könnte für häufig vorkommende Verteilungen, wie zum Beispiel der Normalverteilung eine Unterklasse erstellen, die dann manche Berechnungen nicht mehr durchführen muss, um Rechenaufwand zu sparen. Beispielsweise verschwinden die ungeraden Momente einer Standardnormalverteilung aus Symmetriegründen. SymPy sieht dies leider nicht immer und berechnet deshalb das entsprechende Integral ohne abzukürzen. Durch die Unterklasse könnte man sich diese Berechnung ersparen. Die Implementierung von diesen Unterklassen hätte dann auch den Vorteil, dass Eingaben einfacher werden würden. Momentan muss man die ganzen Dichten und Symbole mit SymPy manuell definieren. Für diese Unterklassen wäre das dann alles schon programmiert und für ein Normalverteilungsobjekt müsste man nur ein μ und ein σ angeben.

Sehr interessant wäre es zudem einen vierten Typen der Zufallsvariablen zu implementieren, der deutlich abstrakter ist. Man müsste zuerst eine Klasse auf die `sym.Function`-Klasse aufbauen, die Dichtefunktionen beschreibt. Für diese Dichtefunktionen sollte man einen allgemeinen Integralbegriff implementieren, der Eigenschaften wie Linearität besitzt und für den man

$$\int \varphi \, \mathrm{d}\mathbb{P} = 1$$

definiert. Mit Dichtefunktions-Klasse könnte man dann eine `RandomVariableAbstract`-Klasse definieren, mit der man vielleicht mehr Sätze, wie in der Bachelorarbeit angedeutet, mit SymPy „beweisen“ kann.

Ein wichtiger Teil, der in meinem Programm recht wenig Beachtung gefunden hat, ist das Exception Handling. Damit der Code nicht allzu umfangreich wird und da dieses Programm eher als proof-of-concept gedacht ist, wurde auf derartige Überprüfungen verzichtet. Man kann beispielsweise in die `moment`-Methode ohne Probleme eine negative Zahl oder Brüche reinstecken und wird dann vermutlich einen schwierig interpretierbaren Fehler von SymPy erhalten. Um dies zu verbessern, müssten also in jede Methode entsprechende `try`- und `except`-Blöcke eingebaut werden, die diese Fehler herausfiltern.

Abschließend denke ich, dass mit dieser Arbeit ein gut funktionierender Grundstein für vielfältige Erweiterungen gelegt wurde.

6 Anhang

Hier der Code von `RandomVariableFinite.py`

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

from .RandomVariable import RandomVariable


class RandomVariableFinite(RandomVariable):
    def __init__(self, density, variables, force_density=False):
        """
        A subclass of RandomVariable for finite random variables.

        Parameters:
            density (dict): The density function of the random variable as a dictionary mapping
    values to probabilities.
            variables (sympy.Symbol): The symbol used in the density function as a sympy object.
            force_density (bool): An Argument for skipping the check if the density function is
    normalized. Default is False.
        """
        self.type = 'f'
        density = RandomVariableFinite._make_density(density)
        # Initiates Random Variable
        super(RandomVariableFinite, self).__init__(density, variables, force_density=
    force_density)

    @staticmethod
    def _make_density(old_density):
        """
        Converts the density function of the random variable to a sympy expression if necessary.

        Parameters:
            old_density (dict): The density function of the random variable as a dictionary
    mapping values to probabilities.

        Returns:
            dict: The density function of the random variable as a dictionary mapping sympy
    expressions to sympy expressions.
        """
        density = {}
        for key in old_density.keys():
            density.update({sym.sympify(key): sym.sympify(old_density[key])})
        return density

    def integrate_random_variable(self, expr, lower=-sym.oo, upper=sym.oo):
        """
        Returns the sum of an expression involving the random variable.

        Parameters:
            expr (sympy.Expr): The expression to be summed as a symbolic expression.
            lower (sympy.Expr): The lower bound of the summation. Default is -sym.oo.
            upper (sympy.Expr): The upper bound of the summation. Default is sym.oo.

        Returns:
            sympy.Expr: The sum as a symbolic expression.
        """
        integral = sym.Integer(0)
        for key in self.density.keys():
            # Only values inside intervall
            if key >= lower and key <= upper:
                integral += expr.subs(self.variable, key) * self.density[key]
```

```
    integral = sym.simplify(integral)
    return integral

def distribution_function(self):
    """
    Returns the distribution function of the random variable.

    Returns:
        dict: The distribution function of the random variable as a dictionary mapping
values to cumulative probabilities.
    """
    sortable = True
    keys = list(self.density.keys())
    for key in keys:
        if isinstance(key, sym.Symbol):
            print('WARNING: Can\'t sort values.')
            sortable = False
            break
    if sortable:
        keys = sorted(keys)
    cumulative_probability = self.density[keys[0]]
    distribution_function = {keys[0]: cumulative_probability}
    keys.pop(0)
    for key in keys:
        cumulative_probability += self.density[key]
        distribution_function.update({key: cumulative_probability})
    return distribution_function

def entropy(self):
    """
    Returns the entropy of the random variable.

    Returns:
        sympy.Expr: The entropy as a symbolic expression.
    """
    entropy = sym.Integer(0)
    for probability in self.density.values():
        entropy += probability * sym.log(probability)
    entropy = - entropy
    entropy = sym.simplify(entropy)
    return entropy

def plot_density(self, show=True, use_latex=True):
    """
    Plots the density function of the random variable.

    Parameters:
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is True.

    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    x_values = list(self.density.keys())
    y_values = list(self.density.values())
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
```

```
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.scatter(x_values, y_values, marker='.')
    if show:
        plt.show()
    else:
        return fig, ax

def plot_distribution_function(self, show=True, use_latex=True):
    """
    Plots the distribution function of the random variable.

    Parameters:
        lower (float or None): The lower bound of the plot domain. Default is None.
        upper (float or None): The upper bound of the plot domain. Default is None.
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is False.

    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    distribution_function = self.distribution_function()
    keys = list(distribution_function.keys())
    min_value = min(keys)
    max_value = max(keys)
    distance = max_value - min_value
    # Broader Plot
    lower = min_value - 0.1 * distance
    upper = max_value + 0.1 * distance
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Distribution function')
    # 0 from - infinity to first value
    ax.hlines(y=0, xmin=lower, xmax=min_value, color='tab:blue', linewidth=2)
    # 1 from last value to infinity
    ax.hlines(y=1, xmin=max_value, xmax=upper, color='tab:blue', linewidth=2)
    for num, key in enumerate(keys):
        # Skips last line
        if num < len(keys) - 1:
            ax.hlines(y=distribution_function[key], xmin=keys[num], xmax=keys[num+1], color=
'tab:blue', linewidth=2)
    if show:
        plt.show()
    else:
        return fig, ax

def simulate(self, number):
    """
    Simulates the random variable using inverse transform sampling.

    Parameters:
        number (int): The number of samples to generate.

    Returns:
        list: A list of simulated values of the random variable.
    """
```

```
        simulate = []
        uni = np.random.uniform(0, 1, number)
        distribution_function = self.distribution_function()
        for num in uni:
            for key in distribution_function.keys():
                if num <= float(distribution_function[key]):
                    simulate.append(float(key))
                    break
        return simulate
```

Hier der Code von `RandomVariableDiscrete.py`

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

from .RandomVariable import RandomVariable


class RandomVariableDiscrete(RandomVariable):
    def __init__(self, density, variables, supp=[], force_density=False):
        """
        A subclass of RandomVariable for discrete random variables.

        Parameters:
            density (sympy.Expr): The density function of the random variable as a symbolic
    expression.
            variables (sympy.Symbol): The symbol used in the density function as a sympy object.
            supp (list): The support of the density function as a list of two elements. Default
    is [sym.Integer(0), sym.oo].
            force_density (bool): An Argument for skipping the check if the density function is
    normalized. Default is False.
        """
        self.type = 'd'
        if supp == []:
            self.supp = [sym.Integer(0), sym.oo]
        else:
            self.supp = supp
        # Initiates Random Variable
        super(RandomVariableDiscrete, self).__init__(density, variables, force_density=
    force_density)

    def integrate_random_variable(self, expr, lower=sym.Integer(0), upper=sym.oo):
        """
        Returns the sum of an expression involving the random variable.

        Parameters:
            expr (sympy.Expr): The expression to be summed as a symbolic expression.
            lower (sympy.Expr): The lower bound of the summation. Default is sym.Integer(0).
            upper (sympy.Expr): The upper bound of the summation. Default is sym.oo.

        Returns:
            sympy.Expr: The sum as a symbolic expression.
        """
        lower = sym.Max(lower, self.supp[0])
        upper = sym.Min(upper, self.supp[1])
        integral = sym.summation(expr * self.density, (self.variable, lower, upper)).doit()
        integral = RandomVariable.clean_piecewise(integral)
        integral = sym.simplify(integral)
        return integral

    def distribution_function(self, value=None):
        """
        Returns the distribution function of the random variable.
```

```
    Parameters:
        value (float or None): The value at which to evaluate the distribution function.
Default is None.

    Returns:
        sympy.Expr or float: The distribution function as a symbolic expression if value is
None, or as a numerical value if value is given.
    """
    # Purely symbolic calculation
    if value == None:
        t = sym.Symbol('t', real=True)
        upper = sym.Min(self.supp[1], sym.floor(t))
        distribution_function = self.integrate_random_variable(sym.Integer(1), upper=upper)
        return distribution_function
    # Numeric calculation
    else:
        value = sym.sympify(value)
        upper = sym.Min(self.supp[1], sym.floor(value))
        distribution_function = self.integrate_random_variable(sym.Integer(1), upper=upper)
        distribution_function = float(distribution_function.evalf())
        return distribution_function

def plot_density(self, lower=0, upper=10, show=True, use_latex=True):
    """
    Plots the density function of the random variable.

    Parameters:
        lower (int): The lower bound of the plot domain. Default is 0.
        upper (int): The upper bound of the plot domain. Default is 10.
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is True.

    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    x_values = np.arange(lower, upper + 1, step=1, dtype=int)
    y_values = []
    for x_value in x_values:
        # Value inside support
        if x_value >= self.supp[0] and x_value <= self.supp[1]:
            y_value = float(self.density.subs(self.variable, x_value).evalf())
        # Value inside support
        else:
            y_value = 0
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.scatter(x_values, y_values, marker='.')
    if show:
        plt.show()
    else:
        return fig, ax

def plot_distribution_function(self, lower=0, upper=10, show=True, use_latex=True):
    """
```

```
        Plots the distribution function of the random variable.

        Parameters:
            lower (int): The lower bound of the plot domain. Default is 0.
            upper (int): The upper bound of the plot domain. Default is 10.
            show (bool): A flag to indicate whether to show the plot or return the figure and
    axis objects. Default is True.
            use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
    labels. Default is False.

        Returns:
            matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
    plot, if show is False.
        """
        if self._test_for_symbols():
            return
        lower = int(np.floor(lower))
        upper = int(np.ceil(upper))
        if use_latex:
            plt.rc('text', usetex=True)
        fig, ax = plt.subplots()
        ax.set_xlabel(f'${sym.latex(self.variable)}$')
        ax.set_ylabel('Distribution function')
        for num in range(lower, upper + 1):
            propability = self.distribution_function(value=num)
            ax.hlines(y=propability, xmin=num - 1, xmax=num, color='tab:blue', linewidth=2)
        if show:
            plt.show()
        else:
            return fig, ax

    def simulate(self, number, n_max=100):
        """
        Simulates the random variable using inverse transform sampling.

        Parameters:
            number (int): The number of samples to generate.
            n_max (int): The maximum value of the random variable to consider. Default is 100.

        Returns:
            list: A list of simulated values of the random variable.
        """
        simulate = []
        n_list = [n for n in range(n_max) if n >= self.supp[0] and n <= self.supp[1]]
        uni = np.random.uniform(0, 1, number)
        for num in uni:
            cumulative_probability = sym.Integer(0)
            for n in n_list:
                cumulative_probability += self.density.subs(self.variable, n)
                if num <= cumulative_probability:
                    simulate.append(n)
                    break
        return simulate
```

Hier der Code von `RandomVariableContinuous.py`

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

from .RandomVariable import RandomVariable


class RandomVariableContinuous(RandomVariable):
    def __init__(self, density, variable, supp=[], force_density=False):
```

```
    """
    A subclass of RandomVariable for continuous random variables.

    Parameters:
        density (sympy.Expr): The density function of the random variable as a symbolic
expression.
        variable (sympy.Symbol): The symbol used in the density function as a sympy object.
        supp (list): The support of the density function as a list of two elements. Default
is [-sym.oo, sym.oo].
        force_density (bool): An Argument for skipping the check if the density function is
normalized. Default is False.
    """
    self.type = 'c'
    if supp == []:
        self.supp = [-sym.oo, sym.oo]
    else:
        self.supp = supp
    # Initiates Random Variable
    super(RandomVariableContinuous, self).__init__(density, variable, force_density=
force_density)

def integrate_random_variable(self, expr, lower=-sym.oo, upper=sym.oo):
    """
    Returns the integral of an expression involving the random variable.

    Parameters:
        expr (sympy.Expr): The expression to be integrated as a symbolic expression.
        lower (sympy.Expr): The lower bound of the integration. Default is -sym.oo.
        upper (sympy.Expr): The upper bound of the integration. Default is sym.oo.

    Returns:
        sympy.Expr: The integral as a symbolic expression.
    """
    lower = sym.Max(lower, self.supp[0])
    upper = sym.Min(upper, self.supp[1])
    integral = sym.integrate(expr * self.density, (self.variable, lower, upper)).doit()
    integral = RandomVariable.clean_piecewise(integral)
    integral = sym.simplify(integral)
    return integral

def distribution_function(self):
    """
    Returns the distribution function of the random variable.

    Returns:
        sympy.Expr: The distribution function as a symbolic expression.
    """
    t = sym.Symbol('t', real=True)
    distribution_function = self.integrate_random_variable(sym.Integer(1), upper=t)
    return distribution_function

def plot_density(self, lower=-5, upper=5, numpoints=100, show=True, use_latex=True):
    """
    Plots the density function of the random variable.

    Parameters:
        lower (float): The lower bound of the plot domain. Default is -5.
        upper (float): The upper bound of the plot domain. Default is 5.
        numpoints (int): The number of points to plot. Default is 100.
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is True.
```

```
    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    x_values = np.linspace(lower, upper, num=numpoints)
    y_values = []
    for x_value in x_values:
        # Value inside support
        if x_value > self.supp[0] and x_value < self.supp[1]:
            y_value = float(self.density.subs(self.variable, x_value).evalf())
        # Outside support zero
        else:
            y_value = 0
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(self.variable)}$')
    ax.set_ylabel('Density function')
    ax.plot(x_values, y_values)
    if show:
        plt.show()
    else:
        return fig, ax

def plot_distribution_function(self, lower=-5, upper=5, numpoints=100, show=True, use_latex=
True):
    """
    Plots the distribution function of the random variable.

    Parameters:
        lower (float): The lower bound of the plot domain. Default is -5.
        upper (float): The upper bound of the plot domain. Default is 5.
        numpoints (int): The number of points to plot. Default is 100.
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is True.

    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    distribution_function = self.distribution_function()
    t = sym.Symbol('t', real=True)
    x_values = np.linspace(lower, upper, num=numpoints)
    y_values = []
    for x_value in x_values:
        if x_value < self.supp[0]:
            y_value = 0
        elif x_value > self.supp[1]:
            y_value = 1
        else:
            y_value = float(distribution_function.subs(t, x_value).evalf())
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
```

```
        ax.set_xlabel(f'${sym.latex(self.variable)}$')
        ax.set_ylabel('Distribution function')
        ax.plot(x_values, y_values)
        if show:
            plt.show()
        else:
            return fig, ax

    def simulate(self, number):
        """
        Simulates the random variable using inverse transform sampling.

        Parameters:
            number (int): The number of samples to generate.

        Returns:
            list: A list of simulated values of the random variable.
        """
        simulate = []
        uni = np.random.uniform(0, 1, number)
        t = sym.Symbol('t', real=True)
        distribution_function = self.distribution_function()
        mean = self.mean()
        for num in uni:
            eq = sym.Eq(distribution_function, num)
            sim = sym.nsolve(eq, t, mean)
            simulate.append(float(sim))
        return simulate
```

Hier der Code von `RandomVariable.py`

```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt


class RandomVariable:
    CLEAN_PIECEWISE = True

    def __init__(self, density, variable, force_density=False):
        """
        Initializes a random variable object with a given density function and symbol used in
    the density function.

        Parameters:
            density (sympy.Expr or dict): The density function of the random variable as a
    symbolic expression or dictionary.
            variable (sympy.Symbol): The symbol used in the density function as a sympy object.
            force_density (bool): An Argument for skipping the check if the density function is
    normalized. Default is False.
        """
        self.density = density
        self.variable = variable
        self.force_density = force_density
        if force_density == False:
            self._is_density()

    def integrate_random_variable(self):
        pass

    @staticmethod
    def clean_piecewise(expr):
        """
        Cleans up a piecewise expression by removing unnecessary branches and arguments.
```

```
        Parameters:
            expr (sympy.Expr): The expression to be cleaned up.

        Returns:
            sympy.Expr: The cleaned up expression.
        """
        if not RandomVariable.CLEAN_PIECEWISE:
            return expr
        if isinstance(expr, sym.Piecewise):
            print('WARNING: Chopped up piecewise-function')
            expr = expr.args[0][0]
        elif expr.is_Atom:
            expr = expr
        # Repeats previous steps for all parts
        else:
            args = [RandomVariable.clean_piecewise(arg) for arg in expr.args]
            expr = expr.func(*args)
        return expr

    @staticmethod
    def no_chopping():
        """
        Disables the clean_piecewise method.
        """
        RandomVariable.CLEAN_PIECEWISE = False

    def _test_for_symbols(self):
        """
        Tests if the density function of the random variable has multiple symbols.

        Returns:
            bool: True if the density function has multiple symbols, False otherwise.
        """
        if self.type == 'f':
            # Tests key for symbols
            for key in self.density.keys():
                if isinstance(key, sym.Symbol):
                    print('ERROR: Multiple symbols in density not supported.')
                    return True
            # Tests values for symbols
            for value in self.density.values():
                if isinstance(value, sym.Symbol):
                    print('ERROR: Multiple symbols in density not supported.')
                    return True
            return False
        else:
            if set(self.density.free_symbols) != set([self.variable]):
                print('ERROR: Multiple symbols in density not supported.')
                return True
            else:
                return False

    def _is_density(self):
        """
        Checks if the density function of the random variable is normalized.

        Returns:
            sympy.Expr: The integral of the density function over the whole domain.
        """
        total = self.integrate_random_variable(sym.Integer(1))
        if not total.equals(sym.Integer(1)):
            print('WARNING: Density not standardized!')
        return total
```

```
# Methods for raw moments
def moment_generating_function(self):
    """
    Returns the moment generating function of the random variable.

    Returns:
        sympy.Expr: The moment generating function as a symbolic expression.
    """
    # Tests for moment generating function attribute
    if hasattr(self, 'MGF'):
        moment_generating_function = self.MGF
    else:
        t = sym.Symbol('t', real=True)
        moment_generating_function = self.integrate_random_variable(sym.exp(t * self.
variable))
        # Sets moment generating function attribute
        self.MGF = moment_generating_function
    return moment_generating_function

def _moment_integration(self, n):
    """
    Returns the nth raw moment of the random variable using integration.

    Parameters:
        n (int): The order of the moment.

    Returns:
        sympy.Expr: The nth raw moment as a symbolic expression.
    """
    moment = self.integrate_random_variable(self.variable**n)
    return moment

def _moment_generating(self, n):
    """
    Returns the nth raw moment of the random variable using differentiation of the moment
generating function.

    Parameters:
        n (int): The order of the moment.

    Returns:
        sympy.Expr: The nth raw moment as a symbolic expression.
    """
    t = sym.Symbol('t', real=True)
    moment_generating_function = self.moment_generating_function()
    moment = sym.diff(moment_generating_function, (t, n))
    moment = moment.subs(t, sym.Integer(0))
    moment = sym.simplify(moment)
    return moment

def moment(self, n, use_integration=True):
    """
    Returns the nth raw moment of the random variable.

    Parameters:
        n (int): The order of the moment.
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the raw moment. Default is True.

    Returns:
        sympy.Expr: The nth raw moment as a symbolic expression.
    """
```

```
    if use_integration == True:
        moment = self._moment_integration(n)
    else:
        moment = self._moment_generating(n)
    return moment

# Methods for central moments
def central_moment_generating_function(self):
    """
    Returns the central moment generating function of the random variable.

    Returns:
        sympy.Expr: The central moment generating function as a symbolic expression.
    """
    # Tests for central moment generating function attribute
    if hasattr(self, 'CMGF'):
        central_moment_generating_function = self.CMGF
    else:
        t = sym.Symbol('t', real=True)
        mu = self.mean()
        moment_generating_function = self.moment_generating_function()
        central_moment_generating_function = sym.exp(- mu * t) * moment_generating_function
        central_moment_generating_function = sym.simplify(central_moment_generating_function
)
        # Sets central moment generating function attribute
        self.CMGF = central_moment_generating_function
    return central_moment_generating_function

def _central_moment_integration(self, n):
    """
    Returns the nth central moment of the random variable by integration.

    Parameters:
        n (int): The order of the central moment.

    Returns:
        sympy.Expr: The nth central moment as a symbolic expression.
    """
    mean = self.mean()
    central_moment = self.integrate_random_variable((self.variable - mean)**n)
    return central_moment

def _central_moment_generating(self, n):
    """
    Returns the nth central moment of the random variable by using differentiation of the
moment generating function.

    Parameters:
        n (int): The order of the central moment.

    Returns:
        sympy.Expr: The nth central moment as a symbolic expression.
    """
    t = sym.Symbol('t', real=True)
    central_moment_generating_function = self.central_moment_generating_function()
    central_moment = sym.diff(central_moment_generating_function, (t, n))
    central_moment = central_moment.subs(t, sym.Integer(0))
    central_moment = sym.simplify(central_moment)
    return central_moment

def central_moment(self, n, use_integration=True):
    """
    Returns the nth central moment of the random variable.
```

```
    Parameters:
        n (int): The order of the central moment.
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the central moment. Default is True.

    Returns:
        sympy.Expr: The nth central moment as a symbolic expression.
    """
    if use_integration == True:
        central_moment = self._central_moment_integration(n)
    else:
        central_moment = self._central_moment_generating(n)
    return central_moment

# Methods for standardized moments
def standardized_moment_generating_function(self):
    """
    Returns the standardized moment generating function of the random variable.

    Returns:
        sympy.Expr: The standardized moment generating function as a symbolic expression.
    """
    # Tests for standardized moment generating function attribute
    if hasattr(self, 'SMGF'):
        standardized_moment_generating_function = self.SMGF
    else:
        t = sym.Symbol('t', real=True)
        mu = self.mean()
        sigma = self.standard_deviation()
        moment_generating_function = self.moment_generating_function().subs(t, t / sigma)
        standardized_moment_generating_function = sym.exp(- mu / sigma * t) *
moment_generating_function
        standardized_moment_generating_function = sym.simplify(
standardized_moment_generating_function)
        # Sets standardized moment generating function attribute
        self.SMGF = standardized_moment_generating_function
    return standardized_moment_generating_function

def _standard_moment_integration(self, n):
    """
    Returns the nth standardized moment of the random variable by integration.

    Parameters:
        n (int): The order of the standardized moment.

    Returns:
        sympy.Expr: The nth standardized moment as a symbolic expression.
    """
    central_moment = self.central_moment(n, use_integration=True)
    standard_deviation = self.standard_deviation(use_integration=True)
    standard_moment = central_moment / standard_deviation**n
    standard_moment = sym.simplify(standard_moment)
    return standard_moment

def _standard_moment_generating(self, n):
    """
    Returns the nth standardized moment of the random variable by using the moment
generating function.

    Parameters:
        n (int): The order of the standardized moment.
```

```
        Returns:
            sympy.Expr: The nth standardized moment as a symbolic expression.
        """
        t = sym.Symbol('t', real=True)
        standardized_moment_generating_function = self.standardized_moment_generating_function()
        standardized_moment = sym.diff(standardized_moment_generating_function, (t, n))
        standardized_moment = standardized_moment.subs(t, sym.Integer(0))
        standardized_moment = sym.simplify(standardized_moment)
        return standardized_moment

    def standard_moment(self, n, use_integration=True):
        """
        Returns the nth standardized moment of the random variable.

        Parameters:
            n (int): The order of the standardized moment.
            use_integration (bool): A flag to indicate whether to use integration or moment
    generating function to calculate the standardized moment. Default is True.

        Returns:
            sympy.Expr: The nth standardized moment as a symbolic expression.
        """
        if use_integration == True:
            standard_moment = self._standard_moment_integration(n)
        else:
            standard_moment = self._standard_moment_generating(n)
        return standard_moment

    def absolute_moment(self, n):
        """
        Returns the nth absolute moment of the random variable.

        Parameters:
            n (int): The order of the absolute moment.

        Returns:
            sympy.Expr: The nth absolute moment as a symbolic expression.
        """
        absolute_moment = self.integrate_random_variable(sym.Abs(self.variable)**n)
        return absolute_moment

    # Methods for cumulants
    def cumulant_generating_function(self):
        """
        Returns the cumulant generating function of the random variable.

        Returns:
            sympy.Expr: The cumulant generating function as a symbolic expression.
        """
        # Tests for cumulant generating function attribute
        if hasattr(self, 'CGF'):
            cumulant_generating_function = self.CGF
        else:
            moment_generating_function = self.moment_generating_function()
            cumulant_generating_function = sym.log(moment_generating_function)
            cumulant_generating_function = sym.simplify(cumulant_generating_function)
            # Sets cumulant generating function attribute
            self.CHF = cumulant_generating_function
        return cumulant_generating_function

    def cumulant(self, n):
        """
        Returns the nth cumulant of the random variable using differentiation of the cumulant
```

```
generating function.

    Parameters:
        n (int): The order of the cumulant.

    Returns:
        sympy.Expr: The nth cumulant as a symbolic expression.
    """
    cumulant_generating_function = self.cumulant_generating_function()
    t = sym.Symbol('t', real=True)
    cumulant = sym.diff(cumulant_generating_function, t, n)
    cumulant = cumulant.subs(t, 0)
    cumulant = sym.simplify(cumulant)
    return cumulant

# Special moments
def mean(self, use_integration=True):
    """
    Returns the mean of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the mean. Default is True.

    Returns:
        sympy.Expr: The mean as a symbolic expression.
    """
    mean = self.moment(1, use_integration=use_integration)
    return mean

def variance(self, use_integration=True):
    """
    Returns the variance of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the variance. Default is True.

    Returns:
        sympy.Expr: The variance as a symbolic expression.
    """
    variance = self.central_moment(2, use_integration=use_integration)
    return variance

def standard_deviation(self, use_integration=True):
    """
    Returns the standard deviation of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the standard deviation. Default is True.

    Returns:
        sympy.Expr: The standard deviation as a symbolic expression.
    """
    variance = self.variance(use_integration=use_integration)
    standard_deviation = sym.sqrt(variance)
    standard_deviation = sym.simplify(standard_deviation)
    return standard_deviation

def skewness(self, use_integration=True):
    """
    Returns the skewness of the random variable.
```

```
    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the skewness. Default is True.

    Returns:
        sympy.Expr: The skewness as a symbolic expression.
    """
    skewness = self.standard_moment(3, use_integration=use_integration)
    return skewness

def kurtosis(self, use_integration=True):
    """
    Returns the kurtosis of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the kurtosis. Default is True.

    Returns:
        sympy.Expr: The kurtosis as a symbolic expression.
    """
    kurtosis = self.standard_moment(4, use_integration=use_integration)
    return kurtosis

def excess_kurtosis(self, use_integration=True):  # Exzess
    """
    Returns the excess kurtosis of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the excess kurtosis. Default is True.

    Returns:
        sympy.Expr: The excess kurtosis as a symbolic expression.
    """
    kurtosis = self.kurtosis(use_integration=use_integration)
    excess_kurtosis = kurtosis - sym.Integer(3)
    excess_kurtosis = sym.simplify(excess_kurtosis)
    return excess_kurtosis

def hyperskewness(self, use_integration=True):
    """
    Returns the hyperskewness of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the hyperskewness. Default is True.

    Returns:
        sympy.Expr: The hyperskewness as a symbolic expression.
    """
    hyperskewness = self.standard_moment(5, use_integration=use_integration)
    return hyperskewness

def hypertailedness(self, use_integration=True):
    """
    Returns the hypertailedness of the random variable.

    Parameters:
        use_integration (bool): A flag to indicate whether to use integration or moment
generating function to calculate the hypertailedness. Default is True.
```

```
        Returns:
            sympy.Expr: The hypertailedness as a symbolic expression.
        """
        hypertailedness = self.standard_moment(6, use_integration=use_integration)
        return hypertailedness

    def interpret_excess_kurtosis(self, use_integration=True):  # Englische Ausgabe
        """
        Prints the interpretation of the excess kurtosis of the random variable.

        Parameters:
            use_integration (bool): A flag to indicate whether to use integration or moment
    generating function to calculate the excess kurtosis. Default is True.
        """
        excess_kurtosis = self.excess_kurtosis(use_integration=use_integration)
        if excess_kurtosis.free_symbols != set():
            print('ERROR: Symbols in excess kurtosis not supported.')
        if excess_kurtosis == 0:
            kind = "mesokurtic"
        elif excess_kurtosis > 0:
            kind = "leptokurtic"
        elif excess_kurtosis < 0:
            kind = "platykurtic"
        print(f"The Distribution is {kind}.")

    def coefficient_of_variation(self, use_integration=True):
        """
        Returns the coefficient of variation of the random variable.

        Parameters:
            use_integration (bool): A flag to indicate whether to use integration or moment
    generating function to calculate the coefficient of variation. Default is True.

        Returns:
            sympy.Expr: The coefficient of variation as a symbolic expression.
        """
        mean = self.mean(use_integration=use_integration)
        standard_deviation = self.standard_deviation(use_integration=use_integration)
        coefficient_of_variation = standard_deviation / mean
        coefficient_of_variation = sym.simplify(coefficient_of_variation)
        return coefficient_of_variation

    def entropy(self):
        """
        Returns the entropy of the random variable.

        Returns:
            sympy.Expr: The entropy as a symbolic expression.
        """
        entropy = self.integrate_random_variable(- sym.log(self.density))
        return entropy

    def mean_absolute_deviation(self):
        """
        Returns the mean absolute deviation of the random variable.

        Returns:
            sympy.Expr: The mean absolute deviation as a symbolic expression.
        """
        absolute_first_moment = self.absolute_moment(1)
        standard_deviation = self.standard_deviation()
        mean_absolute_deviation = absolute_first_moment / standard_deviation
        mean_absolute_deviation = sym.simplify(mean_absolute_deviation)
```

```
    return mean_absolute_deviation

def characteristic_function(self):
    """
    Returns the characteristic function of the random variable.

    Returns:
        sympy.Expr: The characteristic function as a symbolic expression.
    """
    t = sym.Symbol('t', real=True)
    characteristic_function = self.integrate_random_variable(sym.exp(sym.I * t * self.
variable))
    return characteristic_function

def plot_moment_generating_function(self, lower=-1, upper=1, numpoints=100, show=True,
use_latex=True):
    """
    Plots the moment generating function of the random variable.

    Parameters:
        lower (float): The lower bound of the plot domain. Default is -1.
        upper (float): The upper bound of the plot domain. Default is 1.
        numpoints (int): The number of points to plot. Default is 100.
        show (bool): A flag to indicate whether to show the plot or return the figure and
axis objects. Default is True.
        use_latex (bool): A flag to indicate whether to use LaTeX for rendering the plot
labels. Default is True.

    Returns:
        matplotlib.figure.Figure, matplotlib.axes.Axes: The figure and axis objects of the
plot, if show is False.
    """
    if self._test_for_symbols():
        return
    t = sym.Symbol('t', real=True)
    moment_generating_function = self.moment_generating_function()
    x_values = np.linspace(lower, upper, num=numpoints)
    y_values = []
    for x_value in x_values:
        y_value = float(moment_generating_function.subs(t, x_value).evalf())
        y_values.append(y_value)
    if use_latex:
        plt.rc('text', usetex=True)
    fig, ax = plt.subplots()
    ax.set_xlabel(f'${sym.latex(t)}$')
    ax.set_ylabel('Momentgenerating function')
    ax.plot(x_values, y_values)
    if show:
        plt.show()
    else:
        return fig, ax
```

Literatur

[Kle20] Achim Klenke. *Wahrscheinlichkeitstheorie*. Springer Berlin Heidelberg, 2020.

[MS05] David Meintrup and Stefan Schäffler. *Stochastik*. Springer Berlin Heidelberg, 2005.

[NP21] Barry L. Nelson and Linda Pei. *Foundations and Methods of Stochastic Simulation: A First Course*. Springer International Publishing, 2021.

[Soc23] Joram Soch. *The Book of Statistical Proofs*. 2023.

[Tea23] SymPy Development Team. *SymPy Documentation*, 2023.

[Wik24a] Wikipedia contributors. Coefficient of variation — Wikipedia, the free encyclopedia, 2024. [Online; accessed 18-January-2024].

[Wik24b] Wikipedia contributors. Moment (mathematics) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 4-February-2024].