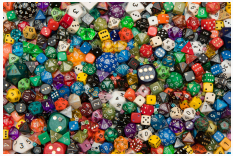
Technische Hochschule Nürnberg

Einführung & Werkzeugkasten

Monte-Carlo-Simulationen mit Python (Fundamente, Tools, erste Demo)

Wintersemester 2025/26,
Benedikt Mangold



Unser Plan

1. Monte Carlo

2. Zufall

3. Reproduzierbarkeit

4. Python-Stack

5. Mini-Demos
   - Schätzung von π
   - PERT-Monte-Carlo in 15 Zeilen
   - Konfidenzintervall des Median
   - Ziegenproblem

6. Ausblick

Literatur

- **NumPy Random – moderne RNG-API (`Generator`, `default_rng`)**. Offizielle Dokumentation. https://numpy.org/doc/2.2/reference/random/generator.html
- **NumPy Random – Überblick**. Modulseite mit Verteilungen und Quick Start. https://numpy.org/doc/2.1/reference/random/index.html
- **Python** `random` – Standardbibliothek (Grundlagen, Seeds, einfache Verteilungen). https://docs.python.org/3/library/random.html
- **Real Python: NumPy Random Number Generator (Einführung)** – praxisnah, inkl. `default_rng`. https://realpython.com/numpy-random-number-generator/
- **Allen B. Downey: Think Stats (kostenloses Buch, Python-first)** – Kapitelweise online/PDF. https://allendowney.github.io/ThinkStats/ und PDF

Lernziele (heute & insgesamt)

Lernziele

- **Verstehen:** Was Monte Carlo *ist* und wann es *hilft*.
  - Von Erwartungswert als Integral bis hin zu Pfaden & Ereignissen.
- **Anwenden:** Simulationen in Python so strukturieren, dass sie wiederholbar und effizient sind.
  - RNG-Design, *saubere* Seeds, modulare Funktionen, Messung von Genauigkeit.
- **Bewerten:** Unsicherheit sichtbar machen (Standardfehler/Konfidenzintervalle) & Entscheidungen mit Band statt Punkt.
- **Entwickeln:** Eigene Simulationsmodelle $\rightarrow$ Hypothesen prüfen, Alternativen vergleichen.

Heute konkret: RNG-Grundlagen, Reproduzierbarkeit, Python-Stack, Mini-Demos.

Monte Carlo

Beispiel für Monte-Carlo

- $X \sim \mathcal{U}(0, 1)$
- $g(x) = \exp(\sin 40x) \cos x^5$
- Berechne $\mathbb{E}[g(X)]$

- Der Ausdruck

$$\mathbb{E}[g(X)] = \int g(x)\, f(x)\, dx$$

  ist hier (und in vielen anderen Beispielen) nicht analytisch lösbar.
- Für hohe Dimensionen/komplexe Domänen ist direktes Integrieren ebenfalls schwierig.

Integrabilität

Wann kann das Integral nicht gelöst werden

Prüfen

Wichtiger Zwischenschritt: immer erst prüfen: ist g integrabel? endliche Varianz?

Beispiel für nicht integrable Funktion

Der Erwartungswert $\mathbb{E}[g(X)] = \int_0^1 g(x)f(x)\,dx$ ist nicht endlich; genauer

$$\mathbb{E}[g(X)] = \int_0^1 \frac{1}{x}\,dx = \infty.$$

Integrabilität

Wann kann das Integral nicht gelöst werden

Beweis:

Was ist Monte Carlo?

Idee hinter dem Buzzword

- **Monte Carlo ersetzt** das Integral durch einen *Durchschnitt*: Unbekannte Größen durch *Zufallsstichproben* schätzen:

$$\hat{\mu}_N = \frac{1}{N} \sum_{i=1}^{N} g(X_i) \text{ mit } X_i \sim f(x)$$

- Funktioniert, wegen des **Gesetzes der großen Zahlen:**

$$\hat{\mu}_N \to \mu_{g(X)} = \mathbb{E}[g(X)] \text{ fast sicher, für } N \to \infty.$$

- **Preis der Einfachheit:** Stichprobenrauschen (Varianz)
  $\Rightarrow$ Dieses Rauschen müssen wir quantifizieren, wir verwenden dazu brauchen Standardfehler/Konfidenzintervalle und Effizienztricks.

Erwartungswerte & Integrale

Wie gut / wie schlecht ist die Monte-Carlo Schätzung?

Zentraler Grenzwertsatz-Intuition: Fehler verhält sich wie $1/\sqrt{N} \Rightarrow 4\times$ mehr Samples halbiert den Standardfehler.

Übung

Eine Monte-Carlo Simulation mit $N = 100$ Wiederholungen hat einen Standardfehler von 22. Wie viel höher muss N gewählt werden, damit der Standardfehler geringer als 2 ist?

Typische Outputs & Unsicherheit

Berichten von Monte-Carlo Ergebnissen

Ergebnisse einer Monte-Carlo-Simulation sind *nicht nur ein einzelner Wert*, sondern immer ein Schätzer mit Unsicherheitsmaß. Wichtige Elemente beim Reporting:

1. **Punkt-Schätzer:** z.B. der Mittelwert $\hat{\mu}$. Er ist leicht zu kommunizieren, aber alleine oft irreführend.
2. **Streuung:** Standardfehler (SE) zeigt, wie stark $\hat{\mu}$ um den wahren Wert schwankt, wenn wir die Simulation wiederholen würden.
3. **Konfidenzintervalle:** z.B. 95%-CI geben eine Spanne, die bei wiederholten Simulationen in 95% der Fälle den wahren Wert enthält (über Replikate oder Bootstrap[1])
4. **Quantile/Schwellenwerte:** in Risikoanwendungen oft wichtiger als der Mittelwert (z.B. 5%-Quantil der Verluste).
5. **Sensitivitäten:** Welche Inputparameter treiben die Unsicherheit? Hier verknüpfen wir zur Sensitivitätsanalyse[2].

[1] späteres Kapitel
[2] späteres Kapitel

Typische Outputs & Unsicherheit

Berichten von Monte-Carlo Ergebnissen

Ideal: Immer $\hat{\mu} \pm SE$ oder CI[3] berichten und angeben, mit wie vielen Samples (N) und welchem RNG gearbeitet wurde.

Minimalbeispiel eines Ergebnis-Satzes

"Der erwartete Gewinn beträgt $\hat{\mu} = 1.27$ mit SE 0.06 (95%-CI: $[1.16, 1.39]$) bei $N = 200\,000$ Samples, RNG=PCG64, Seed=42. Varianzreduktion: antithetisch."

[3] Confidence Interval, auch KI

Grenzen & Missverständnisse

Wann ist von Monte-Carlo abzuraten

Häufige Missverständnisse

Monte Carlo ist mächtig, aber nicht immer die beste Wahl.

- **Analytische Lösungen ignorieren:** Wenn ein Integral oder Erwartungswert geschlossene Formeln hat, ist MC *unnötig teuer*.
- **"Mehr Samples = bessere Wahrheit":** Mehr Samples verringern nur das Stichprobenrauschen, nicht Modellfehler. Falsche Annahmen bleiben falsch.
- **Kostenfalle:** Die Konvergenzrate ist langsam: um den Fehler zu halbieren, braucht man etwa die vierfache Sample-Zahl. Das kann bei komplexen Modellen sehr teuer werden.
- **Validierung wird vergessen:** Ergebnisse müssen gegen einfache Testfälle oder bekannte Spezialfälle geprüft werden, um Vertrauen aufzubauen.

Monte Carlo liefert *nützliche Approximationen*, aber nur so gut wie das Modell und die Sorgfalt der Anwender.

Zufall

Pseudo-Zufallszahlen (PRNG)

Prinzip & Konsequenzen

Computer können keine echten Zufallszahlen erzeugen, sondern verwenden deterministische Algorithmen. Dennoch sind PRNGs praktikabel:

- Ein PRNG startet mit einem *Seed* und erzeugt daraus eine Sequenz, die *zufällig aussieht*.
- Wichtige Kriterien:
  - Lange Periode (wiederholt sich erst nach vielen Milliarden Zahlen)
  - Gleichmäßige Verteilung
  - geringe Autokorrelation (zeitliche/sequentielle Abhängigkeit der Werte)
- Moderne Algorithmen wie PCG64 oder Philox sind schnell und liefern statistisch sehr robuste Sequenzen.
- Vorteil: Ergebnisse sind exakt reproduzierbar, solange Seed und Algorithmus dokumentiert sind.

NumPy `Generator`

Saubere Praxis statt globalem State

In Python für diese Vorlesung: Verwendung der neuen NumPy-API nutzen:

- **Initialisierung:** `rng = np.random.default_rng(seed)` erzeugt einen Generator.
- **Nutzung:** Zufallszahlen zieht man über Methoden wie `rng.normal`, `rng.uniform`, oder `rng.binomial`.
- **Best Practice:** Generator-Objekt an Funktionen übergeben, statt den globalen Zustand von `np.random` zu verwenden. Das erhöht Transparenz und Reproduzierbarkeit.
- **Nebenläufigkeit:** Mit getrennten Generatoren für verschiedene Threads/Prozesse vermeidet man Überschneidungen.

NumPy Generator

Beispiel Practice Beispiel

```
1  seed = 42
2  rng = np.random.default_rng(seed)
3
4  def simulate(n, rng):
5      x = rng.normal(size=n)
6      return x.mean()
7
8  simulate(100, rng)
```

Echter Zufall (TRNG)

Nice to know, selten nötig

- Manche Anwendungen benötigen echte Zufallsquellen: z.B. Kryptographie (Vorhersagbarkeit wäre fatal).
- Quellen sind physikalisches Rauschen oder OS-Entropy-Pools[4].
- Für Monte-Carlo-Simulationen ist TRNG *nicht nötig*, weil wir meist Reproduzierbarkeit wollen. Mit PRNGs können wir exakt dieselben Ergebnisse rekonstruieren.

$\Rightarrow$PRNGs sind Standard in Wissenschaft und Technik, TRNGs sind Spezialfälle.

[4] z.B. /dev/random

RNG-Qualität prüfen

Leichtgewichtig starten

Bevor man komplexe Test-Batterien[5] bemüht, reicht oft ein einfacher Check:

- **Histogramm:** Sind die gezogenen Werte aus $U(0, 1)$ wirklich gleichverteilt?
- **Scatterplot:** Paare (U_t, U_{t+1}) sollten gleichmäßig im Quadrat verteilt sein, nicht auf Mustern liegen.
- **Momente:** Berechneter Mittelwert und Standardabweichung sollten nahe an den Theoretischen liegen (z.B. Mittelwert von $\mathcal{U}(0, 1) = 0.5$).
- **Stresstests:** Simulationen mit extremen Parametern (z.B. sehr kleine Wahrscheinlichkeiten) decken Schwächen auf.

[5] Für tiefere Analysen gibt es Pakete wie `TestU01` – das ist aber eher Forschungsthema als Inhalt der Vorlesung

RNG-Qualität prüfen

Sanity Check - ein Beispiel

Geben Sie je ein Beispiel für den Test mit **Histogramm** und **Scatterplot**, welche auf hochwertige bzw. schlechte RNGs schließen lassen

Variablentransformation

Inverse Transformationsmethode für die Generierung von Zufallszahlen aus beliebigen Verteilungen

Ziel. Aus einer Gleichverteilung $U \sim \mathcal{U}(0,1)$ eine Zufallsvariable mit gegebener Verteilungsfunktion F erzeugen.

Inverse Transformationsmethode

Sei $U \sim \mathcal{U}(0,1)$ und setze $X = F^{-1}(U)$. Dann besitzt X die Verteilungsfunktion F, also $X \sim F$.

Beweisidee. Für jedes $x \in \mathbb{R}$ gilt mit obiger Definition

$$\mathbb{P}(X \leq x) = \mathbb{P}\big(F^{-1}(U) \leq x\big) = \mathbb{P}\big(U \leq F(x)\big) = F(x),$$

da U auf $[0,1]$ gleichverteilt ist. □

Algorithmus.

1. Ziehe $U \sim \mathcal{U}(0,1)$.
2. Berechne $X = F^{-1}(U)$ (Quantilsfunktion).
3. Dann gilt $X \sim F$.

Beispiele.

- Exponentialverteilung: $F^{-1}(u) = -\frac{1}{\lambda} \ln(1-u)$.
- Normalverteilung: $X = \Phi^{-1}(U)$
- Bernoulli(p) (diskret): $F^{-1}(u) = \mathbf{1}\{u \leq p\}$.

Reproduzierbarkeit

Seeds & deterministische Läufe

„Gleiche Zahlen, gleiche Antworten“

Warum sind Seeds so wichtig?

- Mit einem festen Seed lässt sich ein Experiment exakt reproduzieren, auch Monate später[6].
- Bei paralleler Verarbeitung sollte jeder Prozess seinen eigenen Seed haben, abgeleitet aus einer Haupt-SeedSequence.
- Manchmal jedoch[7], sollte man identische Zufallsfolgen verwenden (Common Random Numbers), um die Differenz stabiler zu schätzen.

Dokumentation: Notieren Sie Seed, Generator-Typ und Versionsnummern der Bibliotheken.

[6] Einzige Voraussetzung: Gleiche Paketversionen (nächstes Unterkapitel)
[7] z.B. für Szenarienvergleiche, späteres Kapitel

Environments & Abhängigkeiten

- **Isolieren:** `venv` oder `conda`; klare `requirements.txt` / `environment.yml`
- **Versionierbar:** `pyproject.toml` (PEP 621) für Projekte
- **Builds:** CI-Checks (z. B. `pytest`), Linting, minimaler Smoke-Test
- **Daten:** unveränderliche Rohdaten, abgeleitete Ergebnisse separat

Environments & Abhängigkeiten

Von “läuft bei mir” zu “läuft bei allen, auch in einem halben Jahr“

Ziel: Eine Umgebung definieren, die heute und in sechs Monaten identisch reproduzierbar ist — auf Ihrem Rechner, einem CI-Server und bei Kommilitoninnen[8].

Bausteine und konkrete Empfehlungen:

- **Version-Pinning statt „latest“:** Bibliotheken mit festen Versionen; große Sprünge nur bewusst.
- **Eine Quelle der Wahrheit:** Entweder `requirements.txt` *oder* `conda-environment.yml`; nicht beides konkurrierend pflegen.
- **CI-Sanity-Checks:** minimaler Lauf (Importe, 1–2 Smoke-Tests) in GitHub Actions/GitLab CI; bricht bei Inkompatibilitäten.
- **Plattformunterschiede:** Achten Sie auf OS-spezifische Wheels (macOS/Windows/Linux). Für volle Kontrolle: Container (Docker/Podman) als Option.

[8] oder beim korrigierenden Dozenten

Environments & Abhängigkeiten

Ist ein Paket ohne Versionsnummer aufgelistet, wird die neuste Version installiert (*latest*)

Beispiel `requirements.txt`**:**

```
numpy==1.26.4
scipy==1.13.1
pandas==2.2.2
matplotlib==3.8.4
```

Beispiel `environment.yml` **(conda):**

```
name: simtools
channels: [conda-forge]
dependencies:
- python=3.11
- numpy=1.26
- scipy=1.13
- pandas=2.2
- matplotlib=3.8
```

Ordnerstruktur & Artefakte

“Finde ich die Dinge in 3 Monaten noch?”

Klare Struktur erleichtert Zusammenarbeit, Review und Reproduzierbarkeit

- **configs/**: Parametrisierung $\Rightarrow$ klare Trennung von Code und Einstellungen.
- **data/**: Rohdaten bleiben unverändert; Zwischenschritte sind nachvollziehbar.
- **src/** & **notebooks/**: Notebook ruft Funktionen aus `src` — Logik ist testbar.
- **reports/**: Tabellen als CSV/Parquet + kurze Erläuterung; Referenzierbar im Pitch.

Extra Mile: Jede erzeugte Datei enthält Metadaten (Seed, Datum, Git-Commit) z.B. als Header-Zeile.

Ordnerstruktur & Artefakte

```
project/
    configs/          # yaml/json für Seeds, Parameter
    data/
        raw/          # unverändert, read-only
        interim/      # bereinigt/transformiert
        processed/    # final für Analysen
    figures/          # png/pdf-Plots
    notebooks/        # Exploration, Demos
    reports/          # Tabellen + Kurztexte
    src/              # Funktionen/Modelle
    tests/            # Z.B. Unit-Tests
    requirements.txt # oder environment.yml
    README.md         # Informationen über das Projekt im Browser gut lesbar
```

Cookiecutter

- Es kann lästig sein, alle diese Ordner anzulegen.
- Daher gibt es python-Pakete, welche die Ordnerstruktur und Dummy-Dateien automatisiert anlegen:

  https://cookiecutter-data-science.drivendata.org/
- Selbstverständlich kann man Verzeichnisse, die leer bleiben würden, oder Dateien, die nicht benötigt werden, wieder löschen.

Beispiel-Repository
https://github.com/davidfrivas/eeg_random_forest

Notebooks vs. Skripte

Klare Rollenverteilung

- **Notebooks sind Präsentations- und Explorationsmedien**: kurze Versuche, Visualisierung, Erklär-Text. Keine Business-Logik, keine *silent side effects*[9].
- **Skripte/Module sind Produktionscode**: Funktionen sind testbar, wiederverwendbar, dokumentiert. Notebooks verwenden nur öffentliche Funktionen.
- **Workflow**: Idee im Notebook $\rightarrow$ robuste Funktion in `src/` $\rightarrow$ Notebook ruft Funktion auf $\rightarrow$ Plot/Erklärung.

[9] unerwartete oder nicht dokumentierte Änderungen im Zustand eines Systems

Notebooks vs. Skripte

Trennung der Zuständigkeiten[10], Kernfunktionen in eigenen Skripten verfassen:

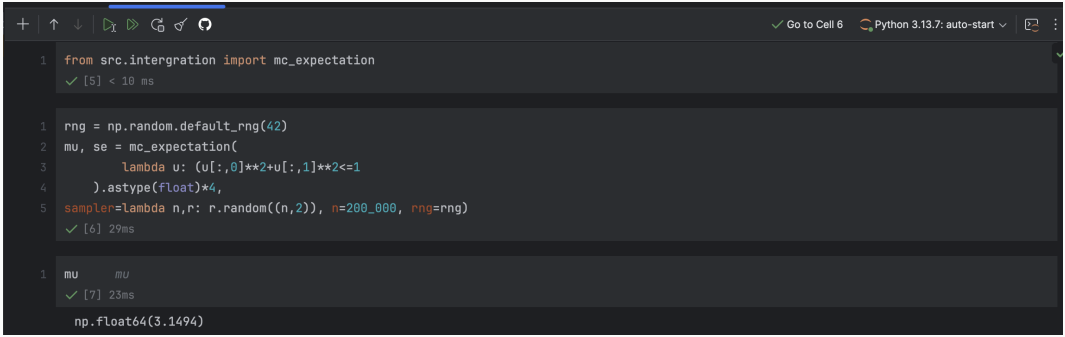
```python
# src/integration.py
import numpy as np

def mc_expectation(g, sampler, n, rng):
    x = sampler(n, rng)
    gx = g(x)
    mu = gx.mean()
    se = gx.std(ddof=1)/np.sqrt(n)
    return mu, se
```

[10] *Separation of concerns*

Notebooks vs. Skripte

Versuche, Tests, Präsentationen der Ergebnisse in eigenem Notebook (z.B. unter `notebooks/presentation.ipynb`)



Tipp: Bevor das Notebook gespeichert (gepusht) wird, **Restart** → **Run All**, damit die Zellen in der gleichen Reihenfolge ausgeführt werden (Reproduzierbarkeit)

ohm

Python-Stack

Wichtige Pakete

Python Pakete, welche wir vorerst verwenden werden:

Paket	Wofür
NumPy	Vektorisierung, Broadcasting, schnelle Arrays **RNG:** `np.random.default_rng` → `rng.normal`, `rng.binomial`, …
SciPy Stats	PDF/CDF/PPF, MLE-Fitting, Stichproben, `scipy.stats.bootstrap`

NumPy & SciPy Stats — Fundament mit konkreten Mustern

- **Pattern**[11] Sampling $\rightarrow$ Transform $\rightarrow$ Aggregation ($\hat{\mu}$, Var, CI)
  1. Stichproben ziehen
  2. Transformieren
  3. Aggregieren
  4. Unsicherheit angeben
- **Fallstricke:** Numerische Stabilität (log-Dichten statt Dichten), Underflow bei sehr kleinen Wahrscheinlichkeiten, **immer** den RNG explizit übergeben.

[11] Welches wir wirklich oft tun werden

Experiment

Vektorisieren vs. Loop

Vektorisierung (*x* als Array) ist meist 10–100 × schneller als Python-Schleifen.

```
1  def loop_op(x_list):
2      y = [0.0]*len(x_list)
3      for i in range(len(x_list)):
4          y[i] = a * x_list[i] + b
5      return y
6
7  def listcomp_op(x_list):
8      return [a * xi + b for xi in x_list]
9
10 def numpy_op(x_np):
11     return a * x_np + b
```

Experiment

Auswertung

N N	Loop [s]	List Comprehension [s]	NumPy Vectorized [s]	Speedup vs Loop (NumPy)	Speedup vs ListComp (NumPy)
1 000	0.000	0.000	0.000	31.201	22.276
10 000	0.000	0.000	0.000	97.476	64.027
100 000	0.004	0.002	0.000	132.888	88.165
1 000 000	0.043	0.029	0.001	75.834	52.238
2 000 000	0.084	0.059	0.001	144.853	101.606

NumPy & SciPy Stats — Fundament mit konkreten Mustern

Beispiel: MC-Integration eines Erwartungswerts[12]: $X \sim \beta(2,5), g(x) = \sqrt{x}$

```python
import numpy as np

seed = 42
rng = np.random.default_rng(seed)

def mc_beta_sqrt(n=100_000, rng):
    x = rng.beta(a=2, b=5, size=n)
    gx = np.sqrt(x)
    mu = gx.mean()
    se = gx.std(ddof=1)/np.sqrt(n)
    return mu, se
```

[12] Die β-Verteilung ist uns noch unbekannt, die Wurzel daraus wahrlich keine Standardverteilung, also gibt es keine geschlossene Formel für den Erwartungswert

pandas & matplotlib

Ergebnisse in Tabellen & Plots übersetzen

Ziel: Replikationen und Szenarien sauber zusammenfassen und mit Fehlermaßen darstellen.

- **pandas:** Ergebnisse als `DataFrame`; Gruppierungen für Szenarien/Seeds
- **Matplotlib:** Histogramme, Dichten, Fehlerbalken, Pfadplots
- **Minimal-Dashboard:** Tabelle (Schätzer, SE, CI) + 1–2 Grafiken
- **Export:** `.csv`/`.parquet` + `.png`/`.pdf` (Plots) für Berichte

Beispiel nächste Folie:

- Die Tabelle enthält die Zusammenfassung für verschiedene Stichprobenumfänge N (Szenarien).
- In der Grafik: Mittelwert $\pm$ 1.96 SE_μ als Fehlerbalken

Beispiel für Output

Tabelle (Zunächst ohne Kontext, Aufgabe zu diesem Ergebnis folgt in nächsten Unterkapitel)

- Je nach dem, ob Sie LaTeX oder andere Textverarbeitung verwenden kann ein `pandas`-Dataframe exportiert werden[13]
- Folgender Output wurde mit `df.to_latex()` produziert[14]:

N	μ	SE_μ	95%-CI difference
1 000	3.144600	0.011959	0.023439
10 000	3.138580	0.003075	0.006027
100 000	3.141076	0.001043	0.002045

- Für Word, Powerpoint rät es sich, den DataFrame als .csv oder .xlsx Datei abzuspeichern
- Für die `README.md` in unserem repository eignet sich `df.to_markdown()`

[13] ggf. müssen noch Pakete als depenencies installiert werden
[14] dieser Foliensatz wurde mit LaTeX generiert

Beispiel für Output

Grafik (Zunächst ohne Kontext, Aufgabe zu diesem Ergebnis folgt in nächsten Unterkapitel)

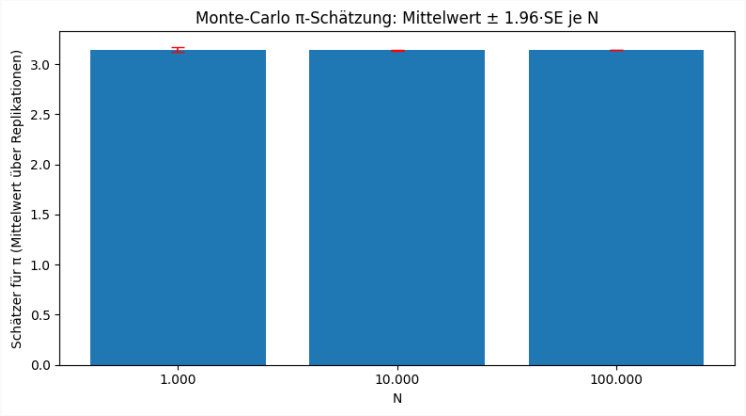


Figure 1: 95%-CI um den Schätzwert μ (basierend auf $R = 20$ Replikationen)

Beispiel für Output (Zoom)

Grafik (Zunächst ohne Kontext, Aufgabe zu diesem Ergebnis folgt in nächsten Unterkapitel)

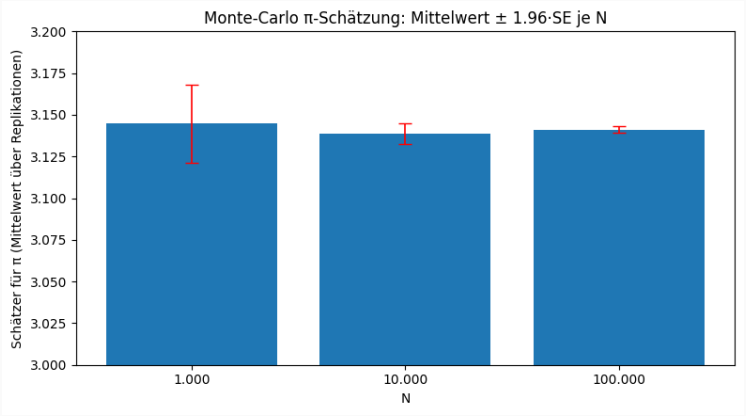


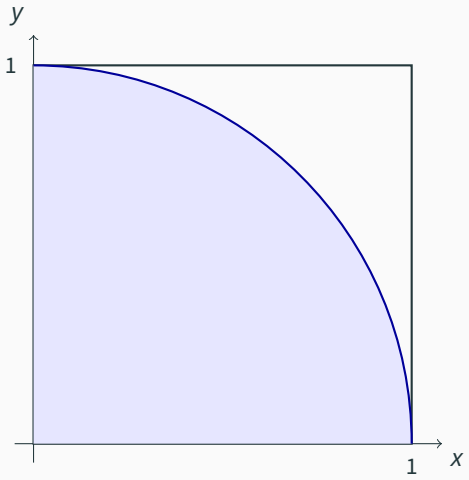
Figure 2: 95%-CI um den Schätzwert μ (Zoom)(basierend auf $R = 20$ Replikationen)

Mini-Demos

Idee & Herleitung

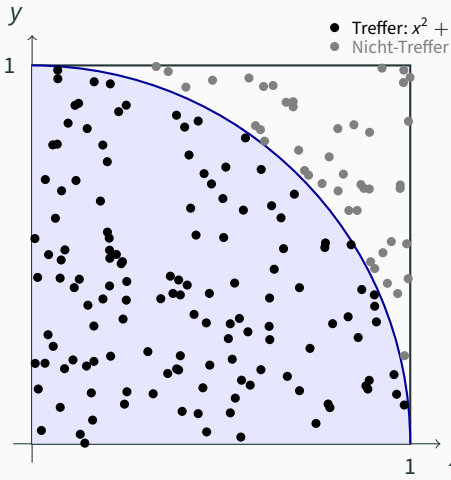
Warum $4\hat{p}$ ein Schätzer für π ist

- Im Einheitsquadrat $[0, 1]^2$ liegt ein Viertelkreis mit Radius 1. Dessen Fläche beträgt $\pi/4$.



Idee & Herleitung

Warum $4\hat{p}$ ein Schätzer für π ist



- Wenn wir N Punkte (X_i, Y_i) gleichverteilt ziehen, ist

$$p = \mathbb{P}(X^2+Y^2 \leq 1) = \frac{\text{Viertelkreis-Fläche}}{\text{Quadrat-Fläche}} = \frac{\pi/4}{1}.$$

- Die Trefferquote $\hat{p}$ ist ein unverzerrter Schätzer von p; daher ist $\widehat{\pi} = 4\hat{p}$ ein Schätzer für π.
- *Rauschen:* $\hat{p}$ ist binomialverteilt[a] mit Varianz $p(1-p)/N$. Daraus folgt $\mathrm{SE}(\widehat{\pi}) = 4\sqrt{\hat{p}(1-\hat{p})/N}$.

[a] überlegen Sie weshalb

Idee & Herleitung

Warum $4\hat{p}$ ein Schätzer für π ist

Zu beachten:

- **Design-Entscheidungen**:
  - Anzahl der Punkte (Ziehungen)*N*
  - Seed
  - RNG-Typ
  - Vektorisierung oder Streaming
    - Vektorisierung: erzeugt alle Zufallszahlen auf einmal und nutzt Array-Operationen → schnell, braucht aber Speicher proportional zu N.
    - Streaming (Chunking): verarbeitet die Simulation in Blöcken → etwas weniger Overhead pro Block, vor allem aber konstanter Speicherbedarf; ideal, wenn N groß ist oder der RAM knapp ist.
- **Validierung**: Erwartung $\pi \approx 3.1416$, ein Konfidenzintervall sollte diesen Wert häufig enthalten.

Python-Snippet (NumPy, vektorisierte Lösung)

```python
import numpy as np

seed = 42
rng = np.random.default_rng(seed)

def estimate_pi(N=200_000, rng=rng):
    x = rng.random(N)
    y = rng.random(N)
    hits = (x*x + y*y) <= 1.0
    p_hat = hits.mean()
    pi_hat = 4.0 * p_hat
    se = 4.0 * np.sqrt(p_hat * (1.0 - p_hat) / N)
    return pi_hat, se

pi_hat, se = estimate_pi()
print(f"pi approx {pi_hat:.5f} plus/minus {1.96*se:.5f}  (95%-CI)")
```

Erste Experimente

Was wir systematisch variieren

1. **Stichprobengröße *N*:** Verdoppeln von *N* sollte den Standardfehler theoretisch um $\approx 1/\sqrt{2}$ senken. *Test:* log–log-Plot von Fehler vs. *N* mit Steigung $-1/2$.
2. **Replikationen *R*:** Mehrere Seeds geben Stabilität. *Test:* Verteilung der $\hat{\pi}$ für festes *N*; Deckungswahrscheinlichkeit des 95%-CI.
3. **RNG-Varianten:** PCG64 vs. Philox — Ergebnisse sollten statistisch ununterscheidbar sein. *Test:* Zwei-Sample-*t*-Test der Schätzungen.
4. **Varianzreduktion:**[15] Antithetische Paare nutzen $(U, 1-U)$. *Erwartung:* geringere Varianz bei gleichem *N*.

Bericht: Tabelle mit *N*, Mittelwert der $\hat{\pi}$, SE/CI, Laufzeit; Histogramm der Schätzer pro *N*.

[15] Im Kapitel 3

PERT & Dreiecksverteilung

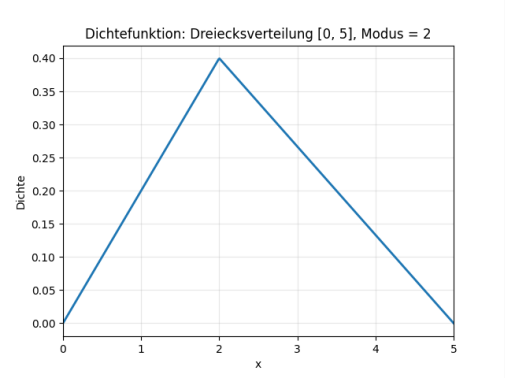
Wann zu benutzen

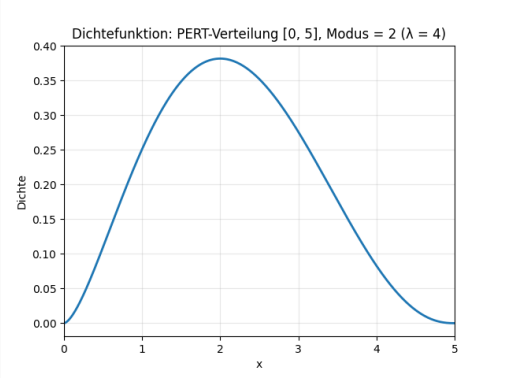
- Ist lediglich der Wertebereich (z.B. $[0, 5]$, allg. $[a, b]$) bekannt, sonst nichts, wählt man nach dem *Prinzip der maximalen Entropie*[16] die Gleichverteilung auf diesem Bereich
- Weiß man Zusätzlich, dass es im Intervall $[a, b]$ eine Stelle m gibt, die am Wahrscheinlichsten ist, führt das textitPrinzip der maximalen Entropie wahlweise zur PERT (Spezialfall der β-Verteilung) oder zur Dreiecksverteilung

[16] nicht Teil der Veranstaltung

PERT & Dreiecksverteilung

Beispiele für Dichtefunktion





PERT–Monte–Carlo

Produktlaunch mit fixem Termin

- **Kontext:** Geplantes Go-Live einer App am *15.01.2026*.
  - Tasks mit unsicheren Dauern (z. B. Backend, Integrationstest, Rollout)
  - Abhängigkeiten zwischen den Tasks (Rollout erst nach Impelentierung möglich)
- **Fragestellung:** Wie groß ist $\mathbb{P}(T_{\text{fertig}} \leq D)$? Was kann man über das Datum sagen, an dem das Produkt mit einer Wahrscheinlichkeit von 90% fertig ist??
- **Daten/Modell:** Drei-Punkt-Schätzungen je Task $(a, m, b \rightarrow$ Triangular/PERT), Abhängigkeitsgraph (Serie/Parallel).
- **Output:** Verteilung von T, On-Time-Wahrscheinlichkeit, kritische Tasks (Sensitivität/„Tornado"-Eindruck)[17].

Nutzen / Entscheidungen anhand des Monte-Carlo Ergebnisses: Reicht der Puffer? Scope kürzen, Ressourcen verschieben oder Termin anpassen?

[17] späteres Kapitel

PERT–Monte–Carlo (Mini)

```python
import numpy as np

# Drei Tasks in Serie, Triangular-Zeiten (min, mode, max)
TASKS = [(2, 3, 6), (1, 2, 4), (3, 4, 9)]
rng = np.random.default_rng(42)

def draw_tri(a, m, b, n, rng):
    u = rng.random(n)
    c = (m - a)/(b - a)
    x = np.where(
        u < c,
        a + np.sqrt(u*(b-a)*(m-a)),
        b - np.sqrt((1-u)*(b-a)*(b-m))
    )
    return x
```

PERT–Monte–Carlo (Mini)

```
1 def sim_T(n=100_000, deadline=10):
2     samples = [draw_tri(a, m, b, n, rng) for (a, m, b) in TASKS]
3     T = np.sum(samples, axis=0)
4     p_on_time = np.mean(T <= deadline)
5     se = np.sqrt(p_on_time * (1 - p_on_time) / n)
6     return p_on_time, 1.96 * se
```

- $\hat{\mu}$ liefert 0.23174, mit Standardfehler 0.003
- Obwohl die Summe der wahrscheinlichsten Dauern der Tasks gleich 9 ist, ist die Summe der Dauern nur mit 23.174 %-iger Wahrscheinlichkeit kleiner als 10.
- Abhängigkeiten zwischen und mögliche Parallelisierungen von Tasks führen zu einer komplizierteren Situation in Zeile 2

Bootstrap-CI (Median)

Lieferzeiten robust ausweisen

- **Kontext:** Onlineshop mit *schiefen* Lieferzeiten (viele pünktlich, wenige starke Ausreißer).
- **Fragestellung:** Können wir für „*Median-Lieferzeit*“ ein 95%-Konfidenz-Intervall angeben?
- **Daten/Modell:**
  - Historische Bestellungen ($n \approx 500$)
  - Median als robuste Kennzahl
  - Analytisches Konfidenzintervall schwierig
- **Lösung:** Bootstrap[18] - nichtparametrisch, ausreißerrobust $\Rightarrow$ direktes Konfidenzintervall für Median/Quantile (und andere Statistiken berechenbar)

Nutzen / Entscheidungen anhand des Bootstrap-CI für den Median: SLA/Marketing-Versprechen („Lieferung in 2–3 Tagen“) halten oder anpassen; Engpassanalyse starten?

[18] Erklärungen in späterem Kapitel

Bootstrap

Grundidee

- Normale Monte-Carlo Situation: Ziehe X aus Verteilung F
- Daten Situation (oder Verteilung stark unregelmäßig) $\Rightarrow$ Verteilung nicht bekannt / gut durch ein F darstellbar
- Bootstrap: Erstelle die empirische Verteilungsfunktion $\hat{F}$, und ziehe X aus Verteilung $\hat{F}$ (**wichtig**: mit Zurücklegen).
- Verwende das Bootstrap-Sample in einer Monte-Carlo Simulation

Bootstrap-CI (Median)

```python
import numpy as np

rng = np.random.default_rng(42)
x = rng.lognormal(mean=1.0, sigma=0.6, size=500)  # z.B. Lieferzeiten

def boot_ci(x, B=5000, alpha=0.05, rng=rng):
    n = len(x)
    # Indices der Beobachtungen für die B Bootstrap samples
    idx = rng.integers(0, n, size=(B, n))
    # Ziehe die B Boostrap samples und berechne pro sample den median
    boots = np.median(x[idx], axis=1)
    # Berechne aus allen B Medianen die Quantile
    lo, hi = np.quantile(boots, [alpha/2, 1-alpha/2])
    return np.median(x), (lo, hi)
```

Bootstrap-CI (Median)

Ergebnis

Code liefert

- Schätzer für den Median: 2.724
- 95%-Konfidenzintervall: [2.5127, 2.9350]

Übung

Verwenden Sie das Ergebnis um eine Aussage über das SLA/Marketing-Versprechen („Lieferung in 2–3 Tagen“) zu treffen.

Monty Hall (Original)[19]

[19] https://wirelesspi.com/the-reason-why-the-monty-hall-problem-continues-to-perplex-everyone/

Monty Hall (Original)

Problemstellung — das Ziegenproblem

- **Setup:** Drei Türen. Hinter einer steht ein Auto, hinter zwei eine Ziege.
- **Ablauf:** Sie wählen *eine* Tür (ohne zu öffnen).
- **Moderator-Regeln (klassisch):**
  - Monty kennt die Inhalte aller Türen.
  - Er öffnet *immer* eine der *anderen* Türen mit einer Ziege.
  - Er öffnet *nie* Ihre gewählte Tür und *nie* die Autotür.
  - Wenn zwei Ziegentüren möglich sind, wählt er (gedacht) zufällig eine davon.
- **Angebot:** Monty bietet Ihnen an, auf die *einzige* verbleibende geschlossene Tür zu wechseln.
- **Frage:** *Sollten* Sie wechseln, um die Gewinnchance zu maximieren?
- **Demo-Ziel:** Gewinnwahrscheinlichkeiten der Strategien „Bleiben“ vs. „Wechseln“ per Simulation schätzen.

Ziegenproblem

Simulation statt Intuition

```python
import numpy as np
rng = np.random.default_rng(42)

def monty(n=100_000, switch=True, rng):
    prize = rng.integers(0, 3, size=n)
    choice = rng.integers(0, 3, size=n)
    # Host öffnet eine Niete != choice
    open_door = np.where(prize==choice,
                         (choice + 1 + (prize == (choice+1))) % 3,
                         3 - prize - choice)
    if switch:
        choice = 3 - choice - open_door
    win = (choice==prize).mean()
    se = np.sqrt(win * (1 - win)/n)
    return win, 1.96 * se

print(monty(True), monty(False))
```

Ziegenproblem

Ergebnis

Übung

1. Überlegen Sie, weshalb Zeile 8-10 das Monty Hall Problem (Ziegenproblem) abbildet

2. Führen Sie obigen Code aus und entscheiden Sie, ob sich ein Wechsel lohnt oder nicht. Geben Sie ein Konfidenzintervall für die Gewinnwahrscheinlichkeit beider Strategien an.

Ausblick

Andere Use-Cases

Sind Regeln klar vorgegeben, kann man diese mit wenig Code implementieren und dann verschiedene Strategien ausprobieren

- Wie viel gewinne ich im Schnitt, wenn ich bei Roulette im Falle eines Verlusts immer verdoppele?
- Mit welcher Strategie gewinne ich bei Mensch Ärgere Dich Nicht?

Abschlussübungen & Bewertungskriterien

Was ist zu tun? Schätzen Sie π!

- Für $N \in \{10^3, 10^4, 10^5\}$ jeweils $R = 20$ Replikationen durchführen (unabhängige Seeds).
- Pro N: Mittelwert von $\hat{\pi}$, Standardfehler (über Replikate) und 95%-CI berichten.
- Einen kurzen Kommentar zu „Genauigkeit vs. Rechenzeit“ verfassen (3–5 Sätze).

Was wird in der Abgabe bewertet?

- *Korrektheit*: saubere RNG-Nutzung, richtige SE/CI-Berechnung, plausibles Verhalten $\propto 1/\sqrt{N}$.
- *Reproduzierbarkeit*: Seed/Versionen dokumentiert, klarer Code-Aufbau (Trennung Notebook/`src`).
- *Kommunikation*: klare Tabelle/Plot, knapper, präziser Text.

Lernziele

Recap

Lernziele

- **Verstehen:** Was Monte Carlo *ist* und wann es *hilft*.
- **Anwenden:** Simulationen in Python so strukturieren, dass sie wiederholbar und effizient sind.
- **Bewerten:** Unsicherheit sichtbar machen (Standardfehler/Konfidenzintervalle)
- **Entwickeln:** Eigene Simulationsmodelle $\rightarrow$ Hypothesen prüfen, Alternativen vergleichen.

Inhalte in diesem Kapitel

- RNG-Grundlagen
- Reproduzierbarkeit
- Python-Stack
- Mini-Demos

Testen Sie Ihr Wissen

Kahoot!