Übungsblatt: Das Schuhpendel
Von der Handymessung zum Bayesschen Fit

DSA Roßleben – Kurs 5.1

> **Szenario:** Oh nein! Ein diabolisches Unwetter apokalyptischen Ausmaßes hat alle Zollstöcke, Lineale und Maßbänder in Roßleben zerstört! :(
> Du hast nur dein Handy, etwas Schnur und deine Schuhe (und einen Laptop mit `Python` darauf) und musst *dringend* für die faire Austragung des Papierfliegerwettbewerbs die Höhe des Balkons im Innenhof messen. Wenn du doch nur eine gute Idee hättest, wie du mithilfe von *etwas Physik und Statistik* die Höhe messen könntest, ohne ein Maßband zu benutzen...

Teil A: Experiment und Rohdaten

Aufgabe 1

Aufbau und Datenaufnahme (kein Computer nötig).

a) Installiere eine Sensor-App auf deinem Smartphone, die die Beschleunigung (*g-Force*, in Einheiten von g) mit Zeitstempel aufzeichnet und als CSV-Datei exportieren kann (z. B. eine „Physics Toolbox"-App – sie ist kostenlos).

b) Baue ein Pendel: Verpacke dein Handy sicher (z. B. in einer Socke oder deinem Schuh) und befestige es an einer möglichst langen Schnur (Treppenhaus, Ast, Balkon – je länger, desto langsamer und einfacher zu vermessen ist die Schwingung).

c) Starte die Aufnahme, lenke das Pendel um einen moderaten Winkel aus und lass es los. Lass es mindestens 15–20 Sekunden frei schwingen, bevor du die Aufnahme stoppst. Frage eine umstehende Person, ein Video von dir dabei zu machen, damit du die Hände frei hast und später in deinem Labortagebuch noch alle Details deines Versuchs rekonstruieren kannst (und Simon und Carlo etwas zu Lachen haben).

d) Exportiere die Messung als CSV-Datei und übertrage sie auf deinen Laptop.

Aufgabe 2

Rohdaten einlesen und Kanal identifizieren.

- Lade deine CSV-Datei mit `np.loadtxt(dateiname, delimiter=",", skiprows=1)` (die erste Zeile enthält meist nur die Spaltennamen). Typischerweise stehen die Spalten für Zeit t (in Sekunden) und die drei Beschleunigungskomponenten a_x, a_y, a_z (in Einheiten von g).
- Plotte alle drei Komponenten gegen die Zeit (drei Teilplots übereinander, geteilte x-Achse).
- Identifiziere: (i) welche Achse eine saubere, regelmäßige Schwingung zeigt (das hängt davon ab, wie du das Handy an der Schnur ausgerichtet hast), und (ii) ein Zeitfenster $[t_{\text{start}}, t_{\text{end}}]$, das die Auslenk- und Stopp-Bewegung beim Anfassen des Handys ausschließt und nur das freie Schwingen enthält.

- Zentriere die Daten in diesem Fenster (ziehe den Mittelwert ab), damit die Schwingung um 0 oszilliert.

Musterlösung 2

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

t, ax, ay, az, atot = np.loadtxt("meine_messung.csv", delimiter=",",
skiprows=1, unpack=True)

fig, axes = plt.subplots(3, 1, figsize=(7, 5.5), sharex=True)
axes[0].plot(t, ax); axes[0].set_ylabel('$a_x/g$')
axes[1].plot(t, ay); axes[1].set_ylabel('$a_y/g$')
axes[2].plot(t, az); axes[2].set_ylabel('$a_z/g$')
axes[2].set_xlabel('$t$ [s]')
plt.tight_layout();
plt.show();
```

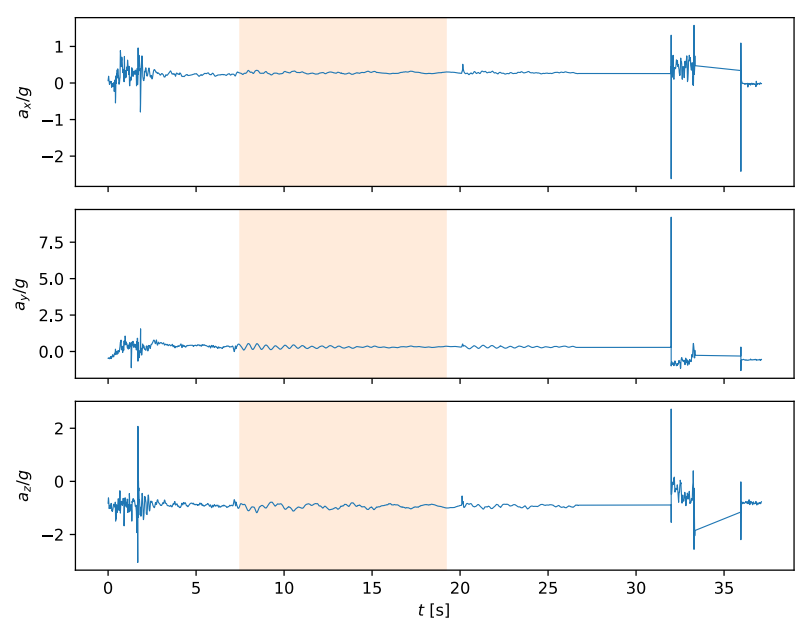


Abbildung 1: Rohe Beschleunigungsdaten aller drei Achsen. Die orange hinterlegte Region ($t = 7.5$ bis $19.2\,\mathrm{s}$) ist das gewählte Analysefenster: Vorher/nachher sieht man deutliche Ausschläge vom Anfassen des Handys, dazwischen eine saubere, langsam abklingende Schwingung.

Hier zeigt die z-Achse (entlang der Schnur ausgerichtet) die sauberste Schwingung. Im gewählten Fenster zentrieren wir:

```
start_time, end_time = 7.5, 19.2
s = np.searchsorted(t, start_time)
e = np.searchsorted(t, end_time)

t_win = t[s:e] - start_time
z_win = az[s:e] - np.mean(az[s:e])
```

Teil B: Ein erster, naiver Blick

Aufgabe 3

Cosinus von Hand anpassen.

- Definiere ein einfaches Modell $a(t) = A\cos(\omega t + \psi_0)$ und plotte es zusammen mit deinen zentrierten Daten $a_z(t)$ aus Aufgabe 2.
- *Probiere von Hand* verschiedene Werte für A, ω und ψ_0 aus, bis die Kurve die Schwingung deiner Daten – insbesondere die *Periode* – gut trifft.
- Argumentiere, warum die Pendelfrequenz aufgrund von Dimensionsanalyse $\propto \sqrt{g/L}$ sein muss. Nimm an, dass diese Kreisfrequenz ω direkt die Pendelfrequenz ist, also $\omega = \sqrt{g/L}$. Bestimme damit, unter der Annahme $g = 9.81\,\mathrm{m/s^2}$, die Pendellänge L.
- Vergleiche das Ergebnis mit der $\pi \times$ Daumen-geschätzten Höhe, in der du die Schnur hast halten lassen. Passt das?

Musterlösung 3

```
def cosinus_modell(t, A, omega, psi0):
    return A * np.cos(omega * t + psi0)

A, omega, psi0 = 0.15, 2.68, 1.3   # von Hand gefunden

plt.plot(t_win, z_win, '.', ms=3, color='0.5', label='Daten')
plt.plot(t_win, cosinus_modell(t_win, A, omega, psi0), label='Modell')
plt.xlabel('$t-t_0$ [s]'); plt.ylabel(r'$a_z/g$ (zentriert)')
plt.legend(); plt.show()

g = 9.81
L_naiv = g / omega**2
print(f'L (naiv) = {L_naiv:.2f} m')
```

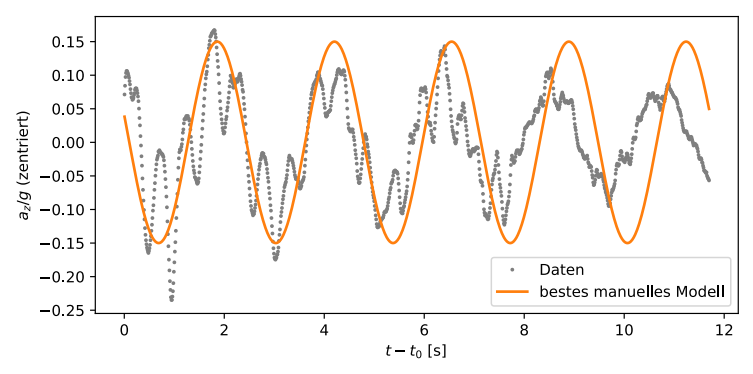


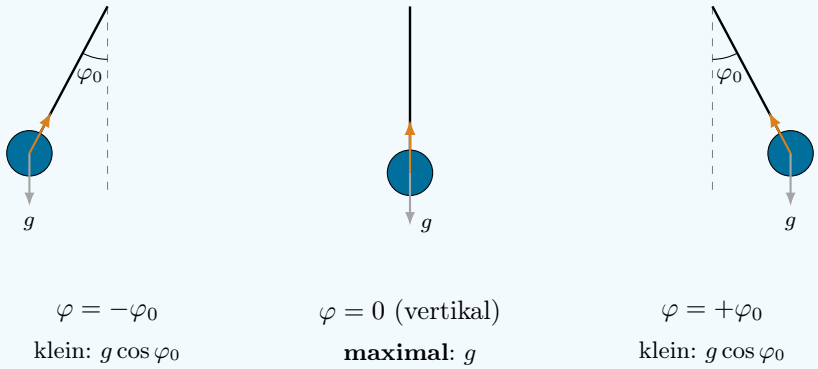
Abbildung 2: Beste von Hand gefundene Cosinus-Anpassung an die zentrierten a_z-Daten.

Mit $\omega \approx 2.68\,\mathrm{rad/s}$ ergibt sich $L_{\mathrm{naiv}} = g/\omega^2 \approx 1.37\,\mathrm{m}$. Das Modell trifft die Periode der Schwingung überraschend gut! Trotzdem: Vergleicht man L_{naiv} mit dem per Maßband gemessenen L_{Band} (in dieser Musterlösung war das Pendel deutlich länger, z. B. $L_{\mathrm{Band}} \approx 4$–$5\,\mathrm{m}$, ein Treppenhaus-Pendel), passt das *nicht*: L_{naiv} ist etwa viermal zu klein. Der gute optische Fit hat uns also in falscher Sicherheit gewogen – *ein guter Fit beweist noch nicht, dass das Modell physikalisch richtig ist!*

Teil C: Was misst das Handy wirklich?

Aufgabe 4

Der Widerspruch aus Aufgabe 3 – und seine Auflösung.
Für kleine Auslenkungen schwingt ein Pendel gemäß $\varphi(t) = \varphi_0 \cos(\omega t)$ mit $\omega = \sqrt{g/L}$ (die Auslenkung φ ist der Winkel der Schnur zur Vertikalen). Das Handy misst aber nicht direkt den Winkel φ, sondern die Beschleunigung entlang seiner eigenen Achsen – und die hängt über die Projektion der Schwerkraft bzw. die Zentripetalbeschleunigung von $\varphi(t)$ *in zweiter Ordnung* ab, z. B. über $\cos\varphi \approx 1 - \varphi^2/2$.



$\varphi = -\varphi_0$ klein: $g\cos\varphi_0$ | $\varphi = 0$ (vertikal) **maximal**: g | $\varphi = +\varphi_0$ klein: $g\cos\varphi_0$

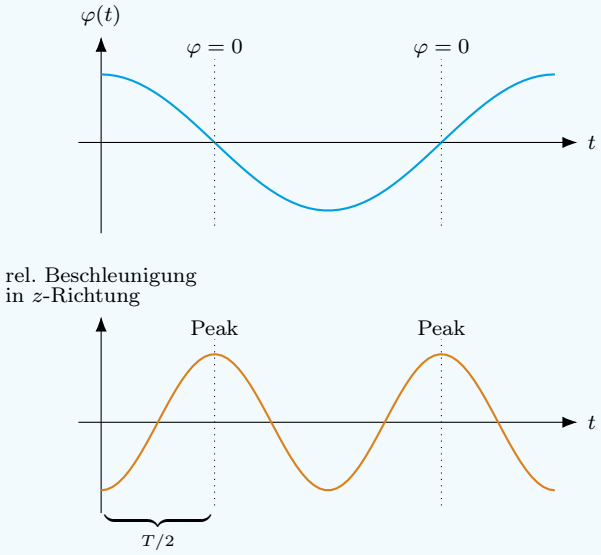


Abbildung 3: Oben: das Pendel an den beiden Umkehrpunkten und im Nulldurchgang, jeweils mit dem Schwerkraftvektor g (grau, immer senkrecht nach unten mit gleicher Stärke) und der Komponente entlang der Schnur, die zur Aufhängung zeigt und die das Handy als Beschleunigung misst (orange). Diese Komponente muss zur Aufhängung hin zeigen – sie hält den Pendelkörper ja auf seiner Kreisbahn, statt ihn einfach nach unten fallen zu lassen. Ihre Stärke ist $g\cos\varphi$: am Umkehrpunkt am kleinsten, im Nulldurchgang (vertikal) am größten. Unten: Da $\varphi(t)$ *zweimal* pro Periode T durch $\varphi = 0$ läuft, hat das gemessene Signal $\propto g\cos\varphi(t)$ auch zwei Maxima pro Periode – die doppelte Frequenz 2ω.

a) Argumentiere explizit über die Kette

$$a_z(t) \;\propto\; \cos\varphi(t) \;\approx\; 1 - \frac{\varphi(t)^2}{2}\,, \qquad \varphi(t)^2 \;\propto\; \cos^2(\omega t) \;\propto\; \cos(2\omega t) + \text{const.}\,,$$

(benutze $\varphi(t) = \varphi_0 \cos(\omega t)$ sowie die Doppelwinkelformel $\cos^2 x = \frac{1}{2}(1 + \cos 2x)$ für den letzten Schritt), dass die gemessene Beschleunigung $a_z(t)$ also mit der *doppelten* Kreisfrequenz 2ω oszilliert, nicht mit ω selbst!

b) Damit hattest du in Aufgabe 3 eigentlich 2ω bestimmt, nicht ω. Wie hängen L_{naiv} (unter der falschen Annahme $\omega_{\text{naiv}} = \omega$) und das tatsächliche L zusammen? Vergleiche mit dem Faktor 4 aus der Musterlösung zu Aufgabe 3.

c) Ein reales Pendel verliert durch Luftwiderstand Energie: die Amplitude klingt mit der Zeit ab, ungefähr wie $\mathrm{e}^{-\gamma t}$ für eine Dämpfungskonstante γ. Schreibe ein vollständiges Modell für die gemessene Beschleunigung auf, das 2ω *und* die Dämpfung berücksichtigt:

$$a(t) = A\,\mathrm{e}^{-\gamma(t-t_0)} \cos\bigl(2\omega(t - t_0) + \psi_0\bigr)\,, \qquad \omega = \sqrt{g/L}\,.$$

Wie viele freie Parameter hat dieses Modell insgesamt (denke daran, dass auch g nicht exakt bekannt ist)? Zähle sie auf. Abbildung 4 zeigt, wie viel besser ein von Hand nachgebessertes Modell dieser Form die Daten trifft, verglichen mit dem naiven, ungedämpften Ein-Frequenz-Modell aus Aufgabe 3.

d) Angenommen, du willst jedes dieser Parameter durch simples Ausprobieren (ein Gitter mit 20 Werten pro Parameter) so einstellen, dass die Kurve zu den Daten passt. Wie viele Kombinationen müsstest du dafür insgesamt testen? Ist das noch „von Hand“ machbar?

Musterlösung 4

a) Mit $\varphi(t) = \varphi_0 \cos(\omega t)$ und der Kleinwinkelnäherung $\cos\varphi \approx 1 - \varphi^2/2$ gilt zunächst $a_z(t) \propto \cos\varphi(t) = 1 - \varphi(t)^2/2$. Einsetzen von

$$\varphi(t)^2 = \varphi_0^2 \cos^2(\omega t) = \frac{\varphi_0^2}{2}\bigl(1 + \cos(2\omega t)\bigr) \tag{1}$$

liefert $a_z(t) \propto 1 - \frac{\varphi_0^2}{4}\bigl(1 + \cos(2\omega t)\bigr) = \text{const.} - \frac{\varphi_0^2}{4}\cos(2\omega t)$. Der konstante Term verschiebt nur den Mittelwert (den wir ohnehin abgezogen haben); der oszillierende Anteil ist proportional zu $\cos(2\omega t)$ – die beobachtete Schwingung läuft also mit der doppelten Kreisfrequenz 2ω.
b) In Aufgabe 3 haben wir (fälschlich) $\omega_{\text{naiv}} = \omega_{\text{fit}}$ gesetzt, wobei $\omega_{\text{fit}} = 2\omega$ die tatsächlich beobachtete Frequenz ist. Da $L \propto 1/\omega^2$, gilt

$$L_{\text{naiv}} = \frac{g}{\omega_{\text{fit}}^2} = \frac{g}{(2\omega)^2} = \frac{1}{4} \cdot \frac{g}{\omega^2} = \frac{L}{4}\,, \tag{2}$$

also $L = 4\,L_{\text{naiv}}$ – genau der Faktor 4, den wir oben numerisch gefunden haben!
c) Das vollständige Modell hat fünf freie Parameter: die Amplitude A, die Phase ψ_0, die Pendellänge L (über $\omega = \sqrt{g/L}$), die Erdbeschleunigung g selbst (die wir ja eigentlich messen wollen, nicht als exakt bekannt voraussetzen sollten) und die Dämpfung γ.
d) Mit 20 Gitterwerten pro Parameter und 5 Parametern wären das $20^5 \approx 3.2 \times 10^6$ Kombinationen – selbst wenn eine einzelne Auswertung nur eine Millisekunde dauert, bräuchte man dafür etwa eine Stunde Rechenzeit, und „von Hand ausprobieren“ ist damit völlig aussichtslos. Wir brauchen also eine systematischere Methode.

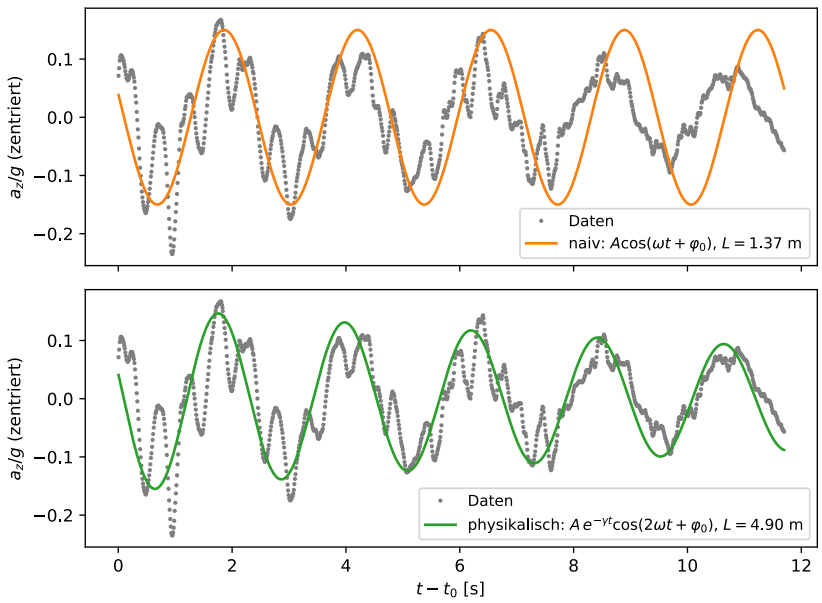


Abbildung 4: Oben: das naive, ungedämpfte Ein-Frequenz-Modell aus Aufgabe 3. Unten: ein von Hand nachgebessertes Modell mit doppelter Frequenz 2ω *und* Dämpfung $e^{-\gamma t}$ – deutlich zu erkennen, wie die abklingende Hüllkurve die Daten jetzt über das ganze Fenster hinweg viel besser trifft.

Teil D: Exkurs – Parameter raten mit dem Computer

Wir stehen jetzt vor genau dem Problem, das Bayessche Statistik und Markov-Chain-Monte-Carlo (MCMC) lösen: ein Modell mit mehreren Parametern soll an verrauschte Daten angepasst werden, *und* wir wollen nicht nur die besten Werte, sondern auch ihre Unsicherheiten kennen.

Aufgabe 5

Der Metropolis-Hastings-Algorithmus.
Der Satz von Bayes liefert die Posterior-Verteilung der Parameter $\theta = (A, \psi_0, L, g, \gamma)$ gegeben die Daten D:

$$p(\theta \mid D) \;\propto\; p(D \mid \theta) \cdot p(\theta)\,, \tag{3}$$

wobei die Likelihood $p(D \mid \theta)$ (wie gut das Modell zu den Daten passt) über eine Gauß-Verteilung mit der geschätzten Messgenauigkeit modelliert wird, und der Prior $p(\theta)$ unser Vorwissen kodiert (z. B. $A > 0$, $0 \le \psi_0 \le 2\pi$, $g \approx 9.81\,\mathrm{m/s^2}$). Statt eines Gitters „läuft“ ein Zufallsspaziergang durch den Parameterraum: Ausgehend von einem aktuellen Punkt θ_{curr} wird ein zufälliger Vorschlag $\theta_{\mathrm{prop}} = \theta_{\mathrm{curr}} + \xi$ erzeugt, wobei ξ aus einer Normalverteilung mit vorgegebener Schrittweite (`step_sizes`) gezogen wird: Ist $p(\theta_{\mathrm{prop}} \mid D) \ge p(\theta_{\mathrm{curr}} \mid D)$, wird der Vorschlag *immer* akzeptiert. Andernfalls wird er trotzdem mit Wahrscheinlichkeit $\alpha = p(\theta_{\mathrm{prop}} \mid D)/p(\theta_{\mathrm{curr}} \mid D)$ akzeptiert. Da $p(\theta \mid D)$ oft astronomisch klein ist, rechnet man in der Praxis immer mit $\log p(\theta \mid D)$ (`log_posterior`); das Kriterium wird dann zu: akzeptiere, falls $\log(u) <$ `log_posterior`$(\theta_{\mathrm{prop}}) -$ `log_posterior`(θ_{curr}), wobei u gleichverteilt in $[0, 1)$ gezogen wird (*Metropolis-Kriterium*).

a) **Implementiere diesen Algorithmus:**

```
def metropolis_hastings(log_post, x0, step_sizes, n_steps, seed=0):
    rng = np.random.default_rng(seed)
```

```
    ndim = len(x0)
    x_curr = np.array(x0, dtype=float)
    logp_curr = log_post(x_curr)

    chain = np.zeros((n_steps, ndim))
    log_post_chain = np.zeros(n_steps)
    n_accept = 0

    for i in range(n_steps):
        x_prop = ...            # zufaelliger Vorschlag
        logp_prop = ...         # log_post am Vorschlag auswerten
        log_alpha = ...         # Metropolis-Kriterium in log-Form

        if ...:                 # np.isfinite(logp_prop) und
        ↪  Metropolis-Kriterium
            x_curr, logp_curr = x_prop, logp_prop
            n_accept += 1

        chain[i] = x_curr
        log_post_chain[i] = logp_curr

    return chain, log_post_chain, n_accept / n_steps
```

b) **Teste deinen Sampler** bevor du ihn auf das (unbekannte) Pendelproblem loslässt an einem Beispiel, dessen Antwort du schon kennst – einer 2-dimensionalen Gauß-Verteilung mit *vorgegebenem* Mittelwertvektor $\vec{\mu}$ und *vorgegebener* Kovarianzmatrix Σ. Die n-dimensionale (*multivariate*) Normalverteilung ist gegeben durch

$$f(\vec{x};\vec{\mu},\Sigma) = \frac{1}{(2\pi)^{n/2}\sqrt{\det\Sigma}} \exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\right], \tag{4}$$

wobei $\vec{\mu}$ ein Vektor ist (ein Mittelwert pro Dimension) und Σ eine $n \times n$-Matrix: auf der Diagonalen stehen die Varianzen der einzelnen Komponenten, abseits der Diagonalen ihre *Kovarianzen* (ein Maß dafür, wie stark zwei Komponenten miteinander korreliert sind – 0 bedeutet unkorreliert). Für $n = 1$ wird Σ zu einer einzelnen Zahl σ^2, und die Formel reduziert sich auf die dir schon bekannte 1-dimensionale Gauß-Dichte. Da beim Metropolis-Kriterium von oben nur die *Differenz* `log_posterior`(θ_{prop}) − `log_posterior`(θ_{curr}) gebraucht wird, kürzen sich additive Konstanten in `log_posterior` immer heraus – es reicht also,

$$\log f(\vec{x};\vec{\mu},\Sigma) = -\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu}) + \text{const.} \tag{5}$$

zu implementieren, wobei die Konstante (der Vorfaktor $(2\pi)^{-n/2}(\det\Sigma)^{-1/2}$, logarithmiert) nicht von $\vec{x}$ abhängt und deshalb komplett wegfällt. Für einen Vektor $\vec{d} = \vec{x} - \vec{\mu}$ und eine Matrix $M = \Sigma^{-1}$ wird die quadratische Form $\vec{d}^T M \vec{d}$ in `numpy` als `d @ M @ d` geschrieben: `@` bedeutet *Matrixmultiplikation* (im Unterschied zu `*`, das elementweise multipliziert). Für einen Vektor `d` (1D-Array) und eine Matrix `M` (2D-Array) liefert `d @ M` zunächst wieder einen Vektor (Vektor-Matrix-Produkt); `(d @ M) @ d` bildet daraus mit dem zweiten `d` das Skalarprodukt und liefert also eine einzelne Zahl – genau $\vec{d}^T M \vec{d}$. **Probiere das an einem kleinen Beispiel mit einer** 2×2**-Matrix aus** und überzeuge dich, dass `d @ M @ d` tatsächlich einen Skalar (keinen Vektor) zurückgibt. **Implementiere dann** `log_post_gauss(theta)` für vorgegebene

$\vec{\mu} = (1.0, -0.5)$ und $\Sigma = \begin{pmatrix} 1.0 & 0.6 \\ 0.6 & 0.5 \end{pmatrix}$, und **rufe deinen Sampler damit auf.** Vergleiche `samples.mean(axis=0)` mit $\vec{\mu}$ und `np.cov(samples.T)` mit Σ, sowie die Akzeptanzrate (Richtwert: 0.2–0.5). **Erzeuge außerdem einen Trace-Plot** (`chain[:,0]` gegen den Schritt-Index, um die Konvergenz zu prüfen) **und ein Streudiagramm** der `samples` nach dem Burn-in (`samples[:,0]` gegen `samples[:,1]`), z. B. mit

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

axes[0].plot(chain[:, 0], lw=0.5)
axes[0].axvline(2000, color='k', ls='--', label='Ende burn-in')
axes[0].set_xlabel('Schritt'); axes[0].set_ylabel(r'$\theta_1$')
axes[0].legend()

axes[1].plot(samples[:, 0], samples[:, 1], '.', ms=1, alpha=0.3)
axes[1].set_xlabel(r'$\theta_1$'); axes[1].set_ylabel(r'$\theta_2$')
plt.tight_layout(); plt.show()
```

Für Interessierte: Warum das Metropolis-Kriterium tatsächlich dafür sorgt, dass die Kette θ-Werte proportional zu $p(\theta \mid D)$ besucht (Stichwort *detailed balance*), wird im Hauptskript im Kapitel „Monte-Carlo-Methoden“ vollständig hergeleitet.

Musterlösung 5

a)

```
def metropolis_hastings(log_post, x0, step_sizes, n_steps, seed=0):
    rng = np.random.default_rng(seed)
    ndim = len(x0)
    x_curr = np.array(x0, dtype=float)
    logp_curr = log_post(x_curr)

    chain = np.zeros((n_steps, ndim))
    log_post_chain = np.zeros(n_steps)
    n_accept = 0

    for i in range(n_steps):
        x_prop = x_curr + rng.normal(0.0, step_sizes, size=ndim)
        logp_prop = log_post(x_prop)
        log_alpha = logp_prop - logp_curr

        if np.isfinite(logp_prop) and np.log(rng.random()) < log_alpha:
            x_curr, logp_curr = x_prop, logp_prop
            n_accept += 1

        chain[i] = x_curr
        log_post_chain[i] = logp_curr

    return chain, log_post_chain, n_accept / n_steps
```

b) Zur Kontrolle: für $n = 1$ ist $\Sigma = (\sigma^2)$ eine 1×1-Matrix, also $\Sigma^{-1} = 1/\sigma^2$ und $\det \Sigma = \sigma^2$; damit wird $f(x; \mu, \sigma^2)$ tatsächlich zur bekannten 1-dimensionalen Gauß-Dichte $\frac{1}{\sqrt{2\pi\sigma^2}} \exp[-(x-\mu)^2/(2\sigma^2)]$ – die multivariate Formel ist also nur deren Verallgemeinerung

auf mehrere, möglicherweise korrelierte Dimensionen. Logarithmieren von $f(\vec{x};\vec{\mu},\Sigma)$ liefert

$$\log f(\vec{x};\vec{\mu},\Sigma) = \underbrace{-\frac{n}{2}\log(2\pi) - \frac{1}{2}\log(\det\Sigma)}_{\text{const. (unabhängig von } \vec{x})} - \frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\,, \tag{6}$$

und die additive Konstante kürzt sich beim Metropolis-Kriterium exakt heraus – es reicht also, nur den quadratischen Term zu implementieren. Kurzer Test der `@`-Notation in Python:

```
d = np.array([2.0, -1.0])
M = np.array([[1.0, 0.6], [0.6, 0.5]])
ergebnis = d @ M @ d
print(ergebnis, np.shape(ergebnis))   # 2.0999999999999996  ()
```

`np.shape(ergebnis)` ist `()` – ein leeres Tupel, das `numpy` für Skalare (nulldimensionale Arrays) verwendet. `d @ M` liefert zunächst einen Vektor (hier: $[2.0\cdot 1.0 + (-1.0)\cdot 0.6,\ 2.0\cdot 0.6 + (-1.0)\cdot 0.5] = [1.4,\ 0.7]$); das anschließende `@ d` bildet daraus mit dem zweiten `d` das Skalarprodukt $1.4\cdot 2.0 + 0.7\cdot(-1.0) = 2.1$ – eine einzelne Zahl, kein Vektor. Damit:

```
mu = np.array([1.0, -0.5])
Sigma = np.array([[1.0, 0.6], [0.6, 0.5]])
Sigma_inv = np.linalg.inv(Sigma)

def log_post_gauss(theta):
    d = theta - mu
    return -0.5 * d @ Sigma_inv @ d

chain, logp, acc = metropolis_hastings(
log_post_gauss, x0=[0.0, 0.0], step_sizes=[0.7, 0.7],
n_steps=20000, seed=1)
samples = chain[2000:]   # burn-in verwerfen

print(f'Akzeptanzrate: {acc:.2f}')
print(f'Mittelwert: {samples.mean(axis=0)}  (Soll: {mu})')
print(f'Kovarianz: \n{np.cov(samples.T)} \n(Soll: \n{Sigma})')
```

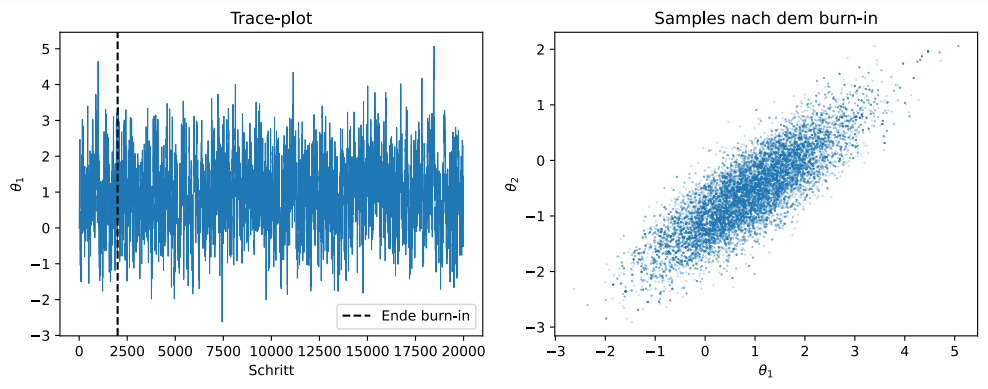


Abbildung 5: Trace-Plot (links, mit sichtbarem Burn-in) und Streudiagramm der Samples nach dem Burn-in (rechts) für den 2D-Gauß-Test. Mittelwert und Kovarianz der Samples stimmen mit den vorgegebenen Werten μ und Σ überein – der Sampler funktioniert.

Die Akzeptanzrate liegt bei diesen `step_sizes` im gewünschten Bereich von 0.2–0.5; `samples.mean(axis=0)` und `np.cov(samples.T)` reproduzieren μ und Σ gut. Der Sampler ist bereit für echte Daten.