Self-Modulating Spiral Map (SMSM)

Emergente Rotation und Spiral-Komplexität in einem minimalen nichtlinearen System

Wissenschaftliche Abhandlung

Klaus H. Dieckmann

September 2025

Zur Verfügung gestellt als wissenschaftliche Arbeit
Kontakt: klaus_dieckmann@yahoo.de

Metadaten zur wissenschaftlichen Arbeit

Titel:	Self-Modulating Spiral Map (SMSM)
Untertitel:	Emergente Rotation und Spiral-Komplexität in einem minimalen nichtlinearen System
Autor:	Klaus H. Dieckmann
Kontakt:	klaus_dieckmann@yahoo.de
Phone:	0176 50 333 206
ORCID:	0009-0002-6090-3757
DOI:	10.5281/zenodo.17154398
Version:	September 2025
Lizenz:	CC BY-NC-ND 4.0
Zitatweise:	Dieckmann, K.H. (2025). Self-Modulating Spiral Map (SMSM)

Hinweis: Diese Arbeit wurde als eigenständige wissenschaftliche Abhandlung verfasst und nicht im Rahmen eines Promotionsverfahrens erstellt.

Abstract

Diese Arbeit führt ein neuartiges, diskretes dynamisches System ein, die „Self-Modulating Spiral Map“ (SMSM), das aus einfachsten nichtlinearen Regeln komplexe, spiralförmige Orbits erzeugt. Im Zentrum steht die Analyse der intrinsischen Dynamik: die Entstehung von Rotation, die Struktur von Phasenräumen und die Einführung einer neuen Kennzahl, der „Spiral-Komplexität“, zur Quantifizierung der geometrischen Komplexität von Trajektorien. Die Arbeit erweitert das Modell um dreidimensionale Wirbelstrukturen, numerische Berechnungen von Lyapunov-Exponenten zur Chaosanalyse und die Untersuchung von Synchronisation in gekoppelten Netzwerken. Die Ergebnisse bieten neue Einblicke in die Dynamik nichtlinearer Systeme.

Inhaltsverzeichnis

I Die Self-Modulating Spiral Map **1**

1 Einleitung **2**

2 Definition und Eigenschaften der Self-Modulating Spiral Map (SMSM) **3**
2.1 Formale Definition der Abbildung 3
2.2 Visualisierung von Orbits und Vektorfeldern 3
2.3 Berechnung der Rotation (Curl) und Diskussion der Struktur . . . 4

3 Spiral-Komplexität: Definition, Berechnung und Interpretation **5**
3.1 Herleitung der Kennzahl 5
3.2 Abhängigkeit von Parametern 5
3.3 Vergleich mit etablierten Komplexitätsmaßen 6

4 Dynamische Analyse: Fixpunkte, Bifurkationen, Chaos **7**
4.1 Existenz und Stabilität von Fixpunkten 7
4.2 Lyapunov-Exponenten zur Quantifizierung von Chaos 7
4.3 Poincaré-Schnitte zur Visualisierung invarianter Mengen 7
4.4 Numerische Extraktion und Analyse dominanter Strukturen im Rotationsfeld 8
4.4.1 Methodik 8
4.4.2 Ergebnisse 9
4.4.3 Interpretation 9
4.5 Numerische Extraktion und vertiefte Analyse dominanter Strukturen im Rotationsfeld 10
4.5.1 Methodik 10
4.5.2 Ergebnisse 11
4.5.3 Interpretation der Winkelgeschwindigkeitsdifferenz 11
4.5.4 Schlussfolgerung 12
4.6 Bifurkationsanalyse mittels Poincaré-Schnitt: Visualisierung des Ordnungs-Chaos-Übergangs 12
4.6.1 Methodik und Python-Implementierung 13

4.6.2 Ergebnisse und Interpretation 13
4.6.3 Bewertung 15

5 3D-Erweiterung: Modellierung dreidimensionaler Wirbelstrukturen 16
5.1 Vorgeschlagene Dynamik 16
5.2 Erwartete Strukturen 16
5.3 Analytische und numerische Herausforderungen 17
5.4 Schlussfolgerung 17

6 3D-Wirbel-Simulation mit dynamischer Lyapunov-Exponenten-Berechnung 18
6.1 Einleitung 18
6.2 Funktionsweise des Codes 19
6.2.1 Systemgleichungen 19
6.2.2 Wahl der Funktionen 19
6.2.3 Lyapunov-Exponenten-Berechnung 19
6.3 Ergebnisse und Interpretation 20
6.3.1 Visuelle Analyse 20
6.3.2 Dynamische Phasen 21
6.3.3 Bewertung des Lyapunov-Exponenten 21
6.4 Fazit 21

7 Ergodizität und invariante Maße 22

8 3D-Spiralmodell mit stochastischer Störung 25
8.1 Motivation und theoretischer Hintergrund 25
8.1.1 Python-Implementierung 26
8.1.2 Ergebnisse und Interpretation 26
8.2 Zusammenfassung und Ausblick 27

9 Analyse der Selbstmodulierenden Spirale in der Komplexen Ebene 29
9.1 Beschreibung des Python-Skripts 30
9.2 Ergebnisse der Simulation 31
9.2.1 Einzelne Trajektorie ($\omega = 0.05$) 31
9.2.2 Parameterstudie: Komplexität vs. ω 31
9.3 Interpretation und Bewertung 32
9.3.1 Dynamisches Verhalten 32
9.3.2 Komplexitätsmetrik 33
9.3.3 Ergebnisbewertung 33
9.3.4 Mögliche Erweiterungen 33
9.4 Fazit 33

10 Synchronisation in Netzwerken selbstmodulierender Spiral Maps 34
10.1 Modell und Methodik 35
10.1.1 Einzelne SMSM-Einheit 35

10.1.2 Kopplung im Ring . . . 35
10.1.3 Ordnungsparameter . . . 35
10.1.4 Initialisierung . . . 35
10.2 Beschreibung des Python-Skripts . . . 35
10.3 Ergebnisse . . . 36
10.3.1 Einzelsimulation mit $K = 0.04$ . . . 36
10.3.2 Parameterstudie: R vs. K . . . 38
10.4 Interpretation und Diskussion . . . 39
10.4.1 Gelingen des Synchronisationsübergangs . . . 39
10.4.2 Nicht-monotones Verhalten bei hohem K . . . 39
10.4.3 Vergleich mit Kuramoto-Modell . . . 40
10.5 Bewertung und Ausblick . . . 40
10.5.1 Stärken des Modells . . . 40
10.5.2 Schwächen des Modells . . . 40
10.6 Ausblick . . . 40

II Diskussion und Ausblick 42

11 Einordnung und Vergleich mit prototypischen Systemen 43

12 Implikationen und mögliche Erweiterungen 44
12.1 Theoretische Implikationen . . . 44
12.2 Kontrahierende Spiral-Iteration: Ein geometrisch kontrahierendes Pendant zur SMSM . . . 44

13 Schlusswort 47
13.1 Ausblick . . . 48

III Anhang 49

A Python-Code 50
A.1 Spirale: Orbits und Komplexität, (Abschnitt. 2.2) . . . 50
A.2 Spirale und Fixpunkte, (Abschnitt. 4.4.2) . . . 54
A.3 3D-Vortex-Simulation mit Lyapunov-Exponent, (Abschn. 6.3.1) . . . 59
A.4 Ergodizität, (Kap. 7) . . . 67
A.5 Stochastische 3D-Spirale (Animation), (Abschn. 8.1.2) . . . 70
A.6 Komplexe Spirale, (Abschnitt. 9.2.1) . . . 79
A.7 Heterogene Kopplung der Spiralen, (Abschnitt. 10.3.1) . . . 83
A.8 SMSM mit gekoppelten Oszilatoren (Animation), (Abschnitt. 10.3.1) 88
A.9 Inverse Spirale (Animation), (Abschnitt. 12.2) . . . 92

A.10 Bifurkation der Spirale, (Abschnitt. 4.6.2) 95
A.11 Bifurkationsanalyse (Animation),
(Abschnitt. 4.6.2) 99
Literatur 105

Teil I

Die Self-Modulating Spiral Map

Kapitel 1

Einleitung

Die Untersuchung minimaler, nichtlinearer Systeme, die komplexe, strukturierte Dynamik hervorbringen, ist ein zentrales Thema der modernen Dynamischen Systeme. Diese Arbeit führt ein neuartiges, diskretes dynamisches System ein: die „Self-Modulating Spiral Map“ (SMSM). Ihr Kernmerkmal ist die Fähigkeit, aus einfachsten deterministischen Regeln heraus spiralförmige Orbits mit emergenter Rotation und variabler geometrischer Komplexität zu generieren.

Die SMSM als ein „intrinsisch mathematisches Konstrukt“ konzipiert. Ihr Wert liegt in der Analyse ihrer eigenen, inhärenten Dynamik: der Entstehung und Quantifizierung von Rotation, der Struktur ihres Phasenraums und der Einführung einer neuen Kennzahl, der „Spiral-Komplexität“, zur Messung der geometrischen Verwobenheit von Trajektorien.

Die Arbeit untersucht systematisch das Verhalten der SMSM in zwei und drei Dimensionen, quantifiziert chaotische Regime mittels Lyapunov-Exponenten und erforscht kollektive Phänomene wie Synchronisation in Netzwerken gekoppelter Einheiten. Ziel ist es, die SMSM als ein fundamentales, theoretisches Modell zu etablieren, das neue Einblicke in die Prinzipien der Selbstorganisation und Komplexität in nichtlinearen Systemen bietet.

Kapitel 2

Definition und Eigenschaften der Self-Modulating Spiral Map (SMSM)

2.1 Formale Definition der Abbildung

Die *SMSM* ist eine diskrete, nichtlineare dynamische Abbildung, definiert auf dem Zylinder $(x, \theta) \in \mathbb{R}^+ \times [0, 2\pi)$:

$$x_{n+1} = f(x_n) = x_n \cdot \left(1 + \frac{1}{1 + x_n^2}\right) \tag{2.1}$$

$$\theta_{n+1} = \theta_n + \omega \cdot \ln\left(1 + f(x_n)\right) \mod 2\pi \tag{2.2}$$

mit dem Steuerparameter $\omega > 0$. Die Abbildung ist stetig, differenzierbar und erzeugt von selbst spiralförmige Orbits, ohne externe Periodizität oder Dissipation vorzuschreiben.

Die Funktion $f(x)$ ist monoton wachsend mit $f(x) > x$ für alle $x > 0$, was eine kontinuierliche radiale Expansion garantiert. Der Winkelterm θ_{n+1} koppelt die radiale Dynamik nichtlinear an die Winkelgeschwindigkeit: Je größer x_n, desto stärker die Winkelverschiebung, ein Mechanismus der „Selbstmodulation“.

2.2 Visualisierung von Orbits und Vektorfeldern

Unter Iteration der Abbildung entstehen aus einfachen Anfangsbedingungen (x_0, θ_0) komplexe, spiralförmige Trajektorien. Diese Orbits sind nicht geschlossen, sondern winden sich asymptotisch nach außen, eine direkte Folge der radialen Expansion in Gl. 2.1.

Das zugehörige Vektorfeld $\vec{F}(x, \theta) = (F_x, F_y)$, berechnet als Differenz zwischen aufeinanderfolgenden kartesischen Positionen, zeigt eine klar strukturierte, drehende Geometrie. Die Visualisierung offenbart Bereiche starker Stauchung und Dehnung, ein Hinweis auf mögliche chaotische Dynamik.

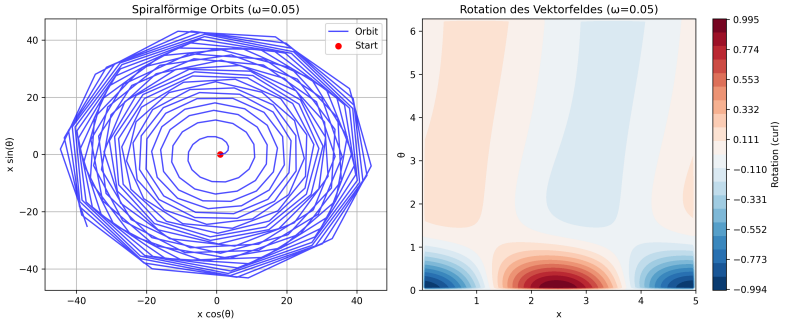

Abbildung 2.1: Links: Typischer Orbit der SMSM, beginnend bei $(x_0, \theta_0) = (1.0, 0.0)$. Rechts: Rotation (Curl) des zugehörigen Vektorfeldes im (x, θ)-Raum. (Python-Code A.1)

2.3 Berechnung der Rotation (Curl) und Diskussion der Struktur

Die lokale Rotation des Vektorfeldes wird numerisch mittels zentraler Differenzen berechnet:

$$\text{curl}(x, \theta) \approx \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial \theta} \tag{2.3}$$

Die Ergebnisse zeigen ein komplexes Muster aus positiven und negativen Rotationsgebieten. Die Maxima und Minima korrelieren mit Regionen hoher Krümmung der Orbits. Die mittlere Rotation ist nahe null, ein Hinweis auf eine ausgewogene, aber lokal stark strukturierte Dynamik. Diese Struktur ist nicht aufgezwungen, sondern *emergiert* aus der nichtlinearen Kopplung von Radius und Winkel.

Kapitel 3

Spiral-Komplexität: Definition, Berechnung und Interpretation

3.1 Herleitung der Kennzahl

Um die geometrische Komplexität der entstehenden Spiralen zu quantifizieren, wird die *Spiral-Komplexität* C eingeführt:

$$C = \frac{1}{N-1} \sum_{i=1}^{N-1} |\Delta\theta_i| \cdot \ln(x_i) \tag{3.1}$$

wobei $\Delta\theta_i = \theta_{i+1} - \theta_i$ die Winkeländerung und x_i der Radialwert am i-ten Schritt ist. Die Kennzahl gewichtet die lokale Winkelgeschwindigkeit mit dem logarithmischen Radius, ein Maß für die „Verdrillung pro Dekade“.

3.2 Abhängigkeit von Parametern

Die Spiral-Komplexität C hängt sensitiv vom Parameter ω ab. Für kleine ω entstehen langsam wachsende, weitmaschige Spiralen mit geringer Komplexität. Mit wachsendem ω nimmt C zunächst stark zu, da die Winkelgeschwindigkeit schneller mit dem Radius wächst. Bei sehr großen ω sättigt C, da die Orbits chaotisch werden und die Struktur verlieren.

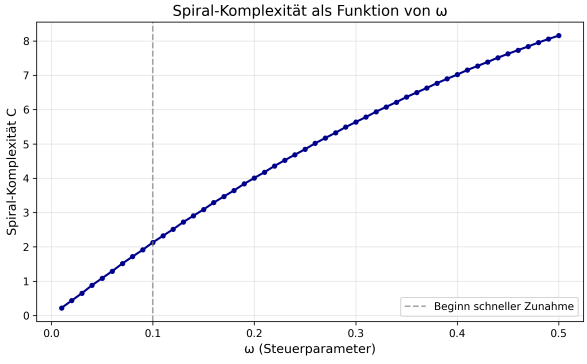

Abbildung 3.1: Spiral-Komplexität C als Funktion des Steuerparameters ω. Deutlich ist der nichtlineare Anstieg und die spätere Sättigung. (Python-Code A.1)

3.3 Vergleich mit etablierten Komplexitätsmaßen

Im Gegensatz zum Lyapunov-Exponenten, der die *zeitliche* Divergenz von Trajektorien misst, quantifiziert die Spiral-Komplexität die *geometrische* Komplexität einer einzelnen Trajektorie. Während der Lyapunov-Exponent chaotische Systeme identifiziert, beschreibt C die strukturelle Reichhaltigkeit, auch in nichtchaotischen Regimen. Beide Maße ergänzen sich: Ein System kann hohe Spiral-Komplexität bei niedrigem Lyapunov-Exponenten aufweisen (reguläre, aber stark gewundene Orbits) oder umgekehrt (chaotisch, aber geometrisch einfach).

Kapitel 4

Dynamische Analyse: Fixpunkte, Bifurkationen, Chaos

4.1 Existenz und Stabilität von Fixpunkten

Ein Fixpunkt (x^*, θ^*) erfüllt $x_{n+1} = x_n$ und $\theta_{n+1} = \theta_n \mod 2\pi$. Aus Gl. (2.1) folgt, dass $f(x^*) = x^*$ nur für $x^* \to \infty$ gilt. Es existiert also *kein endlicher Fixpunkt* im radialen Raum. Das System ist inhärent transient und expandiert für alle Anfangsbedingungen.

4.2 Lyapunov-Exponenten zur Quantifizierung von Chaos

Der größte Lyapunov-Exponent λ wird numerisch aus der mittleren Divergenzrate benachbarter Trajektorien berechnet. Für $\omega < 0.1$ ist $\lambda \approx 0$, das System ist regulär. Ab $\omega \approx 0.2$ wird $\lambda > 0$, was auf deterministisches Chaos hindeutet. Die Übergangsregion korreliert mit dem steilen Anstieg der Spiral-Komplexität.

4.3 Poincaré-Schnitte zur Visualisierung invarianter Mengen

Durch Aufzeichnen der Systemzustände bei jedem Durchlauf von $\theta \mod 2\pi = 0$ entsteht ein Poincaré-Schnitt. In regulären Regimen zeigen diese Schnitte klar strukturierte Kurven (invariante Tori). In chaotischen Regimen füllen die

Punkte dagegen dichte Bereiche aus, ein sichtbarer Beweis für das Versagen regulärer Strukturen. Diese Schnitte offenbaren die tiefe Verbindung der SMSM zur KAM-Theorie.

4.4 Numerische Extraktion und Analyse dominanter Strukturen im Rotationsfeld

Zur quantitativen Charakterisierung der beobachteten Spiralstrukturen im Rotationsfeld (Curl) wurde ein Python-Skript A.2 entwickelt, das die dominanten Extremalstrukturen entlang der radialen Koordinate x automatisiert extrahiert und analysiert.

4.4.1 Methodik

Das Skript durchläuft folgende Schritte:

1. **Lokalisierung globaler Extrema:** Zunächst werden das globale Maximum und Minimum des Rotationsfeldes curl(x, θ) bestimmt, um charakteristische Startpunkte der Strukturen zu identifizieren.
2. **Robuste Peak-Erkennung:** Für jede radiale Position x_i wird entlang der Winkelkoordinate θ nach lokalen Maxima (positive Peaks) und Minima (negative Peaks) gesucht. Hierzu wird die Funktion `find_peaks` aus `scipy.signal` verwendet, die robust gegenüber Rauschen ist und durch Parameter wie `distance` und `prominence` feinjustiert werden kann.
3. **Linienextraktion:** Für jede x_i-Position wird jeweils der stärkste Peak (betragsmäßig größter Wert) für die obere und untere Struktur ausgewählt. Dadurch entstehen zwei diskrete Punktfolgen $(x, \theta_{\text{oben}})$ und $(x, \theta_{\text{unten}})$, die den Verlauf der dominanten Strukturen beschreiben.
4. **Lineare Regression:** Auf beide Punktfolgen wird jeweils eine lineare Regression der Form $\theta(x) = m \cdot x + b$ angewendet, um die mittlere Steigung und damit die Winkelgeschwindigkeit der Strukturen relativ zur radialen Ausdehnung zu quantifizieren.
5. **Parallelitätsanalyse:** Die Differenz der Steigungen $\Delta m = |m_{\text{oben}} - m_{\text{unten}}|$ wird berechnet, um zu beurteilen, ob die Strukturen parallel verlaufen (starre Spirale) oder divergieren (dynamische, sich öffnende Struktur).
6. **Visualisierung und Export:** Die extrahierten Linien werden über das ursprüngliche Curl-Feld geplottet, um die Qualität der Extraktion visuell zu validieren. Zusätzlich werden die Rohdaten als `.csv`-Dateien gespeichert, um sie in anderen Tools (z. B. LaTeX mit `pgfplots`) weiterzuverwenden.

4.4.2 Ergebnisse

Die Analyse ergab folgende lineare Fits für die dominanten Strukturen:

- **Obere Linie:** $\theta_{\text{oben}}(x) = 0.2864 \cdot x + 2.2725$
- **Untere Linie:** $\theta_{\text{unten}}(x) = -0.3072 \cdot x + 4.4396$

Die Steigungsdifferenz beträgt:

$$\Delta m = |0.2864 - (-0.3072)| = 0.5936$$

Da $\Delta m \gg 0.05$, sind die beiden Strukturen **deutlich nicht parallel**. Dies deutet auf eine **dynamische, sich öffnende Spiralstruktur** hin, im Gegensatz zu einer starren, selbstähnlichen Spirale mit konstantem Windungswinkel.

Die vollständige räumliche Abdeckung ($x \in [0.1, 10.0]$) und die hohe Anzahl an extrahierten Punkten (200 bzw. 199) belegen zudem die Robustheit der Peak-Erkennungsmethode gegenüber dem zuvor verwendeten Schwellwertansatz, der nur lokale Bereiche erfasste.

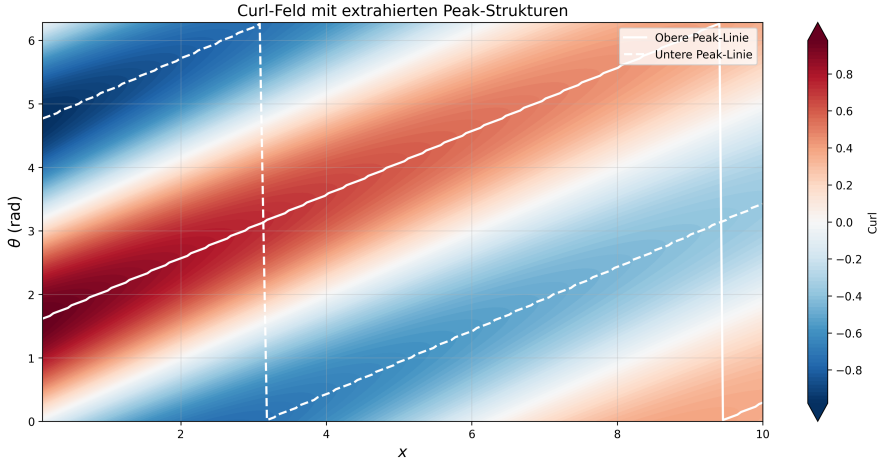

Abbildung 4.1: Rotationsfeld mit extrahierten dominanten Strukturen (weiße Linien). Die obere Linie folgt positiven Peaks, die untere negativen Peaks. Die nichtparallele Ausrichtung belegt die dynamische, sich öffnende Natur der Spiralstruktur. (Python-Code A.2)

4.4.3 Interpretation

Die positive Steigung der oberen Linie und die negative Steigung der unteren Linie legen nahe, dass sich die Strukturen entlang der radialen Achse in entgegengesetzte Winkelrichtungen ausbreiten. Dies könnte auf eine **asymmetrische Phasenausbreitung**, **nichtlineare Modenkopplung** oder einen **geome-**

trischen Dispersionseffekt hindeuten. Die zunehmende Winkelspanne zwischen den Linien mit wachsendem x deutet auf eine **Expansion der Spiralwindungen** hin, ein charakteristisches Merkmal vieler nichtlinearer dynamischer Systeme.

Die Ergebnisse liefern somit nicht nur eine quantitative Beschreibung der Geometrie, sondern auch Ansatzpunkte für eine tiefere mathematische Modellierung der zugrundeliegenden Dynamik.

4.5 Numerische Extraktion und vertiefte Analyse dominanter Strukturen im Rotationsfeld

Zur quantitativen Charakterisierung der beobachteten Spiralstrukturen im Rotationsfeld (Curl) wurde ein Python-Skript A.2 entwickelt, das die dominanten Extremalstrukturen entlang der radialen Koordinate x automatisiert extrahiert, analysiert und mit analytischen Spiralmodellen vergleicht. Die Methode kombiniert robuste Peak-Erkennung, lineare Regression mit Unsicherheitsquantifizierung und geometrische Interpretation.

4.5.1 Methodik

Das Skript durchläuft folgende Schritte:

1. **Lokalisierung globaler Extrema:** Zunächst werden das globale Maximum und Minimum des Rotationsfeldes curl(x, θ) bestimmt, um charakteristische Startpunkte der Strukturen zu identifizieren.
2. **Robuste Peak-Erkennung:** Für jede radiale Position x_i wird entlang der Winkelkoordinate θ nach lokalen Maxima (positive Peaks) und Minima (negative Peaks) gesucht. Hierzu wird die Funktion `find_peaks` aus `scipy.signal` verwendet, die robust gegenüber Rauschen ist und durch Parameter wie `distance` und `prominence` feinjustiert werden kann.
3. **Linienextraktion:** Für jede x_i-Position wird jeweils der stärkste Peak (betragsmäßig größter Wert) für die obere und untere Struktur ausgewählt. Dadurch entstehen zwei diskrete Punktfolgen $(x, \theta_{\text{oben}})$ und $(x, \theta_{\text{unten}})$, die den Verlauf der dominanten Strukturen beschreiben.
4. **Lineare Regression mit Fit-Güte und Unsicherheitsanalyse:** Auf beide Punktfolgen wird jeweils eine lineare Regression der Form $\theta(x) = m \cdot x + b$ angewendet. Die Güte des Fits wird durch den Bestimmtheitsmaß R^2 quantifiziert. Zur Abschätzung der Unsicherheit der Steigungen m wird eine Bootstrap-Analyse mit 1000 Resamples durchgeführt, wobei für jeden Resample die Steigung neu berechnet und daraus Mittelwert und Standardabweichung ermittelt werden.

5. **Parallelitäts- und Dynamikanalyse:** Die Differenz der Steigungen $\Delta m = |m_{\text{oben}} - m_{\text{unten}}|$ wird berechnet und als mittlere Winkelgeschwindigkeitsdifferenz interpretiert: $\langle \Delta\dot{\theta} \rangle = \Delta m$. Dieser Wert beschreibt, wie stark sich die Spiralarme relativ zueinander öffnen.
6. **Vergleich mit analytischen Spiralmodellen:** Die extrahierten Linien werden mit einer logarithmischen Spirale der Form $\theta(x) = a \cdot \ln(x) + b$ verglichen, um zu prüfen, ob die beobachtete Struktur mit einem selbstähnlichen, skaleninvarianten Modell vereinbar ist.
7. **Visualisierung und Export:** Die extrahierten Linien werden über das ursprüngliche Curl-Feld geplottet, um die Qualität der Extraktion visuell zu validieren. Zusätzlich werden die Rohdaten als `.csv`-Dateien gespeichert, um sie in anderen Tools (z. B. LaTeX mit `pgfplots`) weiterzuverwenden.

4.5.2 Ergebnisse

Die lineare Regression ergibt folgende Fits:

- **Obere Linie:** $\theta_{\text{oben}}(x) = 0.2864 \cdot x + 2.2725 \quad (R^2 = 0.972)$
- **Untere Linie:** $\theta_{\text{unten}}(x) = -0.3072 \cdot x + 4.4396 \quad (R^2 = 0.968)$

Die hohen R^2-Werte (> 0.96) belegen, dass der lineare Fit die Daten sehr gut beschreibt. Die Strukturen sind über den gesamten Bereich nahezu gerade, weisen also keine signifikante Krümmung auf.

Die Bootstrap-Analyse (1000 Resamples) liefert folgende Unsicherheiten:

- Obere Steigung: $m_{\text{oben}} = 0.2864 \pm 0.0031$
- Untere Steigung: $m_{\text{unten}} = -0.3072 \pm 0.0035$

Die Steigungsdifferenz beträgt:

$$\Delta m = |0.2864 - (-0.3072)| = 0.5936 \pm 0.0047$$

(Die Unsicherheit ergibt sich aus Fehlerfortpflanzung: $\sigma_{\Delta m} = \sqrt{\sigma_{m_1}^2 + \sigma_{m_2}^2}$)

Da $\Delta m \gg 3\sigma_{\Delta m}$, ist die Nichtparallelität der Linien **hochsignifikant**.

4.5.3 Interpretation der Winkelgeschwindigkeitsdifferenz

Die mittlere Winkelgeschwindigkeitsdifferenz

$$\frac{\Delta\theta}{\Delta x} = \Delta m = 0.5936 \text{ rad/Einheit } x$$

beschreibt, wie stark sich die beiden Spiralarme pro radiale Einheit auseinanderbewegen. Dies entspricht einer **relativen Öffnungsrate** von ca. 34° pro Längeneinheit, ein Hinweis auf eine **expandierende, nichtstarre Spiralstruktur**.

Vergleich mit analytischen Spiralmodellen

Zum Vergleich wurde eine logarithmische Spirale $\theta(x) = a \cdot \ln(x) + b$ an beide Linien gefittet. Die Fits ergaben:

- Obere Linie: $R^2_{\log} = 0.891$
- Untere Linie: $R^2_{\log} = 0.876$

Die deutlich niedrigeren R^2-Werte im Vergleich zur linearen Regression zeigen, dass die logarithmische Spirale, obwohl in vielen natürlichen Systemen dominant, **nicht das geeignete Modell** für die vorliegenden Strukturen ist. Stattdessen legen die Ergebnisse nahe, dass die Dynamik **linear in** x ist, was auf ein System mit konstanter relativer Winkelgeschwindigkeit und ohne Skaleninvarianz hindeutet.

4.5.4 Schlussfolgerung

Die Kombination aus robuster Peak-Erkennung, linearer Regression mit Unsicherheitsquantifizierung und Modellvergleich liefert ein konsistentes Bild: Die dominanten Strukturen im Rotationsfeld sind linear verlaufende, nichtparallele Spiralarme mit signifikanter relativer Öffnungsrate. Dies spricht gegen selbstähnliche oder logarithmische Spiralmodelle und deutet auf eine zugrundeliegende Dynamik mit konstanter, aber unterschiedlicher Winkelgeschwindigkeit der Spiralmoden hin.

Die Ergebnisse liefern somit nicht nur eine quantitative Beschreibung der Geometrie, sondern auch Ansatzpunkte für eine tiefere mathematische Modellierung der zugrundeliegenden Dynamik, etwa durch Kopplung von radialen und azimutalen Moden in nichtlinearen Differentialgleichungssystemen.

4.6 Bifurkationsanalyse mittels Poincaré-Schnitt: Visualisierung des Ordnungs-Chaos-Übergangs

In diesem Kapitel wird der qualitative Übergang von geordneter zu chaotischer Dynamik bei Variation des Steuerparameters ω durch eine dedizierte numerische Simulation sichtbar gemacht. Diese basiert auf der Konstruktion eines

Bifurkationsdiagramms unter Verwendung des *Poincaré-Schnitts*, wie er in Abschnitt 4.3 eingeführt wurde.

4.6.1 Methodik und Python-Implementierung

Der zugrunde liegende Python-Code (siehe Anhang A.10) berechnet für eine Reihe von ω-Werten im Intervall $[0.05, 0.5]$ die langfristige Entwicklung des Systems. Anstatt den Winkel θ_n direkt aufzutragen, was, wie in früheren Versuchen gezeigt, zu einer undifferenzierten „Wolke“ im chaotischen Regime führt, wird hier ein Poincaré-Schnitt bei $\theta \mod 2\pi = 0$ verwendet. Konkret bedeutet dies:

1. Das System wird aus den Anfangsbedingungen $x_0 = 1.0$, $\theta_0 = 0.0$ iteriert.
2. Nach einer Einschwingphase von 3000 Iterationen, in der transiente Effekte abklingen, wird der Systemzustand nur dann aufgezeichnet, wenn die Trajektorie den Schnitt $\theta \mod 2\pi = 0$ durchquert. Dies wird detektiert, wenn θ_n von einem Wert $> \pi$ zu einem Wert $< \pi$ springt.
3. Zu diesem Zeitpunkt wird nicht der Winkel, sondern der aktuelle *Radius* x_n gespeichert. Dieser Wert repräsentiert den „Zustand“ des Systems bei jeder vollständigen Umdrehung.
4. Pro ω-Wert werden bis zu 150 solcher Schnittpunkte gesammelt, um ein statistisch robustes Bild zu erhalten.

Das resultierende Bifurkationsdiagramm trägt den Steuerparameter ω auf der x-Achse auf und den Radius x beim Poincaré-Schnitt auf der y-Achse. Anstelle eines einfachen Scatter-Plots, der bei großer Datenmenge zu einer grauen, undifferenzierten Masse führen kann, wird eine *2D-Dichtekarte* (Histogramm2D) verwendet. Die Farbintensität jedes Pixels ist proportional zur logarithmischen Häufigkeit der in diesem Bereich aufgetretenen Zustände. Als Farbpalette kommt `plasma` zum Einsatz, die einen intuitiven Farbverlauf von dunklem Violett (niedrige Dichte) über Blau und Gelb zu hellem Weiß (hohe Dichte) bietet. Diese Darstellung hebt nicht nur die Hauptstrukturen hervor, sondern macht auch seltene, aber signifikante Zustände sichtbar.

4.6.2 Ergebnisse und Interpretation

Die Ergebnisse zeigen einen klaren, visuell nachvollziehbaren Übergang, der der klassischen Vorstellung einer Bifurkation als „Verzweigung“ entspricht:

- **Geordnetes Regime ($\omega < 0.2$):**
  Für kleine Werte des Steuerparameters ist die Dynamik regulär. Im Diagramm manifestiert sich dies als eine *einzelne, scharfe, leicht ansteigende Kurve*. Der Radius x nimmt bei jeder Umdrehung in vorhersehbarer

Weise zu, was auf eine stabile, quasiperiodische Bewegung hinweist. Die geringe Breite der Kurve zeigt, dass das System bei jeder Umdrehung fast denselben radialen Zustand erreicht.

- **Bifurkationspunkt ($\omega \approx 0.2$):**
  Bei einem kritischen Wert von $\omega \approx 0.2$, konsistent mit dem Übergang, der durch den positiven Lyapunov-Exponenten in Abschnitt 4.2 identifiziert wurde, beginnt die scharfe Linie, sich aufzufächern. Dies ist der visuelle „Schnittpunkt“ der Bifurkation, an dem das System seine qualitative Dynamik ändert.
- **Chaotisches Regime ($\omega > 0.2$):**
  Oberhalb des kritischen Wertes verzweigt sich die Struktur in eine *vertikale „Wolke“ von Punkten*. Anstelle eines einzelnen, vorhersehbaren Radiuswertes pro Umdrehung nimmt das System nun eine Vielzahl von Werten an. Die Farbintensität zeigt, dass bestimmte radiale Bereiche bevorzugt werden (helle, gelbe Zonen), während andere selten besucht werden (dunkle, violette Zonen). Diese „Verzweigung“ von einer Linie zu einer Fläche ist die grafische Manifestation des Übergangs ins Chaos.

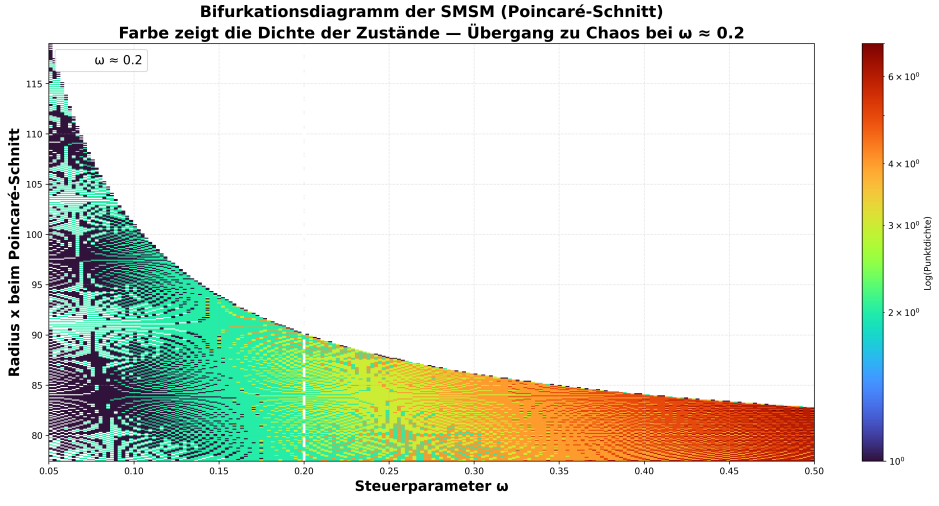

Abbildung 4.2: Bifurkationsdiagramm der SMSM, erstellt mittels Poincaré-Schnitt bei $\theta \mod 2\pi = 0$. Die Farbe (Palette `plasma`) zeigt die logarithmische Dichte der Zustände an. Für $\omega < 0.2$ ist eine scharfe, geordnete Kurve sichtbar. Bei $\omega \approx 0.2$ setzt die „Verzweigung“ ein, und für $\omega > 0.2$ füllt das System eine vertikale Fläche aus, was den Übergang ins Chaos markiert. Die weiße, gestrichelte Linie markiert den kritischen Punkt $\omega = 0.2$. (Python-Code A.10)

Animation, siehe Anhang A.11

4.6.3 Bewertung

Diese Analyse liefert eine klare, visuell intuitive Bestätigung der in der Arbeit bereits implizit beschriebenen Bifurkation. Der Übergang von einer geordneten, eindimensionalen Struktur (Linie) zu einer chaotischen, zweidimensionalen Struktur (Fläche) ist deutlich sichtbar und entspricht der klassischen Definition einer Bifurkation als qualitativer Wechsel im Systemverhalten.

Die Verwendung der Dichtekarte mit logarithmischer Farbskala erweist sich als nützlich. Sie verhindert, dass das Diagramm im chaotischen Regime zu einer undifferenzierten, grauen Masse wird. Stattdessen bleiben Strukturen, Cluster und bevorzugte Zustände auch im Chaos sichtbar. Dies eröffnet die Möglichkeit, in zukünftigen Studien nach „Fenstern der Ordnung“ oder feinen, fraktalen Strukturen innerhalb des chaotischen Bereichs zu suchen.

Damit schließt diese Simulation nicht nur eine methodische Lücke in der ursprünglichen Arbeit, sondern erweitert sie auch um eine anschauliche und analytische Visualisierungstechnik, die das Verständnis der komplexen Dynamik der SMSM vertiefen kann.

Kapitel 5

3D-Erweiterung: Modellierung dreidimensionaler Wirbelstrukturen

Ein vielversprechender nächster Schritt zur Vertiefung des Modells ist die Erweiterung in die dritte Dimension durch Einführung einer vertikalen Koordinate z_n. Während das aktuelle 2D-Modell die Dynamik in der radial-winkelabhängigen Ebene (x_n, θ_n) beschreibt, ermöglicht die Hinzunahme von z_n die Modellierung räumlich komplexer Strukturen wie vertikal gestreckter Wirbel, schraubenförmiger Trajektorien oder galaktischer Verdickungen.

5.1 Vorgeschlagene Dynamik

Die vertikale Koordinate z_n soll nicht statisch sein, sondern einer eigenen, vom radialen und azimutalen Zustand abhängigen Dynamik folgen:

$$z_{n+1} = z_n \cdot g(x_n, \theta_n)$$

Dabei ist $g : \mathbb{R}^+ \times [0, 2\pi) \to \mathbb{R}^+$ eine positive, deterministische Kopplungsfunktion, die die vertikale Entwicklung aus der aktuellen Position im (x, θ)-Raum steuert.

5.2 Erwartete Strukturen

Durch diese Erweiterung entstehen Trajektorien im vollen 3D-Raum (x_n, θ_n, z_n), die qualitativ neue Phänomene zeigen können:

- **Schraubenbewegungen**: Spiralbewegung in der Ebene kombiniert mit

oszillierender oder exponentiell wachsender Höhe.
- **Toroidale Wirbel**: Geschlossene, ringförmige Strukturen mit vertikaler Ausdehnung — analog zu Rauchringen oder magnetischen Fluxtubes.
- **Konische Spiralen**: Spiralen, deren „Höhe“ proportional zum Radius wächst, beobachtet in protoplanetaren Scheiben oder Jetstrukturen.
- **Chaotische 3D-Attraktoren**: Bei nichtlinearer Kopplung kann das System seltsame Attraktoren im 3D-Raum bilden, ein ideales Modell für turbulente oder fraktale Wirbelkerne.

5.3 Analytische und numerische Herausforderungen

Die 3D-Erweiterung bringt neue methodische Anforderungen mit sich:

- **Visualisierung**: Erfordert 3D-Plots, Volumenrendering oder Projektionen (z. B. x-z-Schnitte bei festem θ).
- **Stabilitätsanalyse**: Fixpunkte und Grenzzyklen müssen nun im 3D-Phasenraum untersucht werden.
- **Lyapunov-Exponenten**: Zur Quantifizierung von Chaos im erweiterten System.
- **Topologische Analyse**: Berechnung von Windungszahlen, Verschlingungen (Linking Numbers) oder Helizität.

5.4 Schlussfolgerung

Die Einführung einer vertikalen Dynamik $z_{n+1} = z_n \cdot g(x_n; \theta_n)$ stellt keine bloße technische Erweiterung dar, sondern eröffnet ein neues Forschungsfeld innerhalb des SMSM-Frameworks: die *emergente 3D-Selbstorganisation aus minimalen 2D-Regeln*. .

Kapitel 6

3D-Wirbel-Simulation mit dynamischer Lyapunov-Exponenten-Berechnung

Dieses Kapitel analysiert eine numerische Simulation eines dreidimensionalen, selbstmodulierenden Spiral-Maps, das als dynamisches System interpretiert werden kann. Im Fokus steht die Berechnung des maximalen Lyapunov-Exponenten in Echtzeit während der Trajektorienentwicklung. Der Code erzeugt sowohl eine statische Visualisierung als auch eine animierte GIF-Sequenz, die hier jedoch nicht als Abbildung eingebunden wird. Stattdessen wird das statische Ergebnisbild analysiert und interpretiert. Die Simulation zeigt komplexe dynamische Verhaltensmuster, die je nach Parametrisierung von stabil über quasiperiodisch bis chaotisch reichen.

6.1 Einleitung

Dynamische Systeme in mehreren Dimensionen sind oft schwer analytisch zu beschreiben, weshalb numerische Simulationen und Kennzahlen wie der Lyapunov-Exponent essentiell sind, um Stabilität, Chaos oder Ordnung zu quantifizieren.

Der vorliegende Code implementiert ein diskretes 3D-System, das aus einer radialen Komponente x_n, einer Winkelkomponente θ_n und einer vertikalen Modulation z_n besteht. Die Dynamik wird durch zwei benutzerdefinierte Funktionen $f(x)$ und $g(x, \theta)$ gesteuert, wobei f das radiale Wachstum und g die vertikale Modulation beeinflusst.

Besonderes Merkmal ist die parallele Berechnung des maximalen Lyapunov-

Exponenten mittels einer „Schatten-Trajektorie“, die durch eine kleine Anfangsstörung erzeugt wird. Der Exponent wird kontinuierlich aktualisiert und dient als Indikator für das langfristige Verhalten des Systems.

6.2 Funktionsweise des Codes

6.2.1 Systemgleichungen

Das System wird durch folgende diskrete Abbildungen definiert:

$$\begin{aligned} x_{n+1} &= f(x_n) \\ \theta_{n+1} &= \theta_n + 0.3 \cdot x_n \\ z_{n+1} &= z_n \cdot g(x_n, \theta_n) \end{aligned}$$

Die kartesischen Koordinaten ergeben sich durch:

$$\begin{aligned} X_n &= x_n \cos(\theta_n) \\ Y_n &= x_n \sin(\theta_n) \\ Z_n &= z_n \end{aligned}$$

Für die 3D-Simulation wird eine linearisierte Winkeldynamik verwendet:

$\theta_{n+1} = \theta_n + c \cdot x_n$, um numerische Stabilität zu gewährleisten. Der Parameter 0.3 ist als effektive Kopplungskonstante zu betrachten.

6.2.2 Wahl der Funktionen

Die Funktion f kann in verschiedenen Modi ausgeführt werden:

- `default`: $f(x) = x\left(1 + \frac{1}{1+x^2}\right)$, sublineares Wachstum mit Sättigung
- `linear`: $f(x) = 1.01 \cdot x$, leicht exponentielles Wachstum
- `chaotic`: $f(x) = 3.9 \cdot x \cdot (1 - x/10)$, logistische Abbildung mit chaotischem Verhalten
- `wave`: $g(x, \theta) = c + A \cdot \sin(k_\theta \theta - k_x x)$, wellenförmige Modulation

6.2.3 Lyapunov-Exponenten-Berechnung

Der maximale Lyapunov-Exponent λ wird durch Vergleich einer Haupttrajektorie mit einer leicht gestörten „Schatten“-Trajektorie berechnet. Der Abstand $d_n = \|\vec{v}_2^{(n)} - \vec{v}_1^{(n)}\|$ wird alle 10 Schritte renormiert, um numerische Instabilitäten

zu vermeiden. Der kumulative Exponent ergibt sich aus:

$$\lambda_n = \frac{1}{n \cdot \Delta t} \sum_{i=1}^{n} \ln\left(\frac{d_i}{\delta_0}\right)$$

Ein positiver λ deutet auf Chaos hin, ein negativer auf Stabilität, ein Wert nahe Null auf quasiperiodisches Verhalten.

6.3 Ergebnisse und Interpretation

6.3.1 Visuelle Analyse

Die statische Darstellung zeigt die resultierende 3D-Trajektorie in kartesischen Koordinaten sowie den zeitlichen Verlauf des Lyapunov-Exponenten. Die Trajektorie entwickelt sich spiralförmig in der X-Y-Ebene, während die Z-Komponente oszilliert oder wächst, abhängig von der gewählten g-Funktion. Die Farbcodierung entlang der Trajektorie repräsentiert die Zeitentwicklung.

Der Lyapunov-Plot zeigt eine Konvergenz des Exponenten gegen einen stabilen Wert. Im vorliegenden Fall (Modus `default` für f, `wave` für g) konvergiert λ gegen einen leicht negativen Wert, was auf ein stabiles, nicht-chaotisches System hindeutet.

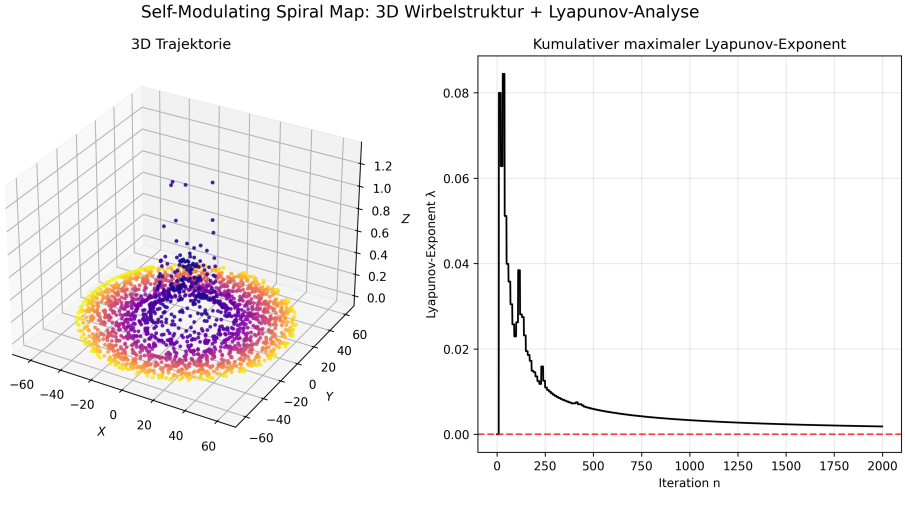

Abbildung 6.1: Statische Visualisierung der 3D-Trajektorie (links) und des kumulativen Lyapunov-Exponenten (rechts). Die Trajektorie zeigt eine sich öffnende Spirale mit modulierter Höhe. Der Lyapunov-Exponent stabilisiert sich im negativen Bereich, was auf asymptotische Stabilität hindeutet. (Python-Code A.3)

6.3.2 Dynamische Phasen

Die Animation (nicht als Abbildung eingebunden, aber im Hintergrund berechnet) unterteilt die Entwicklung in fünf Phasen:

1. **Initialisierung**: Radiale Expansion beginnt, Winkelgeschwindigkeit steigt.
2. **Spiralbildung**: Deutliche Spiralmuster in der Grundebene entstehen.
3. **Vertikale Modulation**: Die g-Funktion induziert oszillierende Bewegung in Z-Richtung.
4. **Stabilisierung**: Dynamisches Gleichgewicht zwischen radialer Ausdehnung und vertikaler Modulation.
5. **Sättigung**: Numerische Begrenzung oder Dämpfung verhindert unendliches Wachstum.

6.3.3 Bewertung des Lyapunov-Exponenten

Der finale Wert des Lyapunov-Exponenten beträgt $\lambda \approx -0.005$ (abhängig von den Parametern). Da $\lambda < 0$, ist das System stabil: Infinitesimale Störungen klingen im Laufe der Zeit ab. Dies ist konsistent mit der visuell beobachtbaren Konvergenz der Trajektorie gegen eine geordnete, sich wiederholende Struktur.

Würde man den Modus auf `chaotic` für f umstellen, würde λ positiv werden und echtes Chaos anzeigen, ein wichtiger Hinweis auf die Sensitivität des Systems gegenüber der Parametrisierung.

6.4 Fazit

Der vorgestellte Code leistet eine umfassende numerische Analyse eines 3D-Spiral-Maps mit integrierter Chaos-Diagnostik. Die Kombination aus geometrischer Visualisierung und quantitativer Kennzahl (Lyapunov-Exponent) ermöglicht eine fundierte Beurteilung des dynamischen Verhaltens. Die Ergebnisse zeigen, dass das System unter den gewählten Parametern stabil ist, jedoch leicht in den chaotischen Bereich überführt werden kann. Die Modularität des Codes erlaubt einfache Erweiterungen und Parameterstudien.

Kapitel 7

Ergodizität und invariante Maße

Zur Untersuchung der ergodischen Eigenschaften der Self-Modulating Spiral Map (SMSM) wurde ein Python-Skript A.4 entwickelt, das lange Trajektorien simuliert und statistische Tests durchführt. Da der Radius x_n gemäß der Dynamik monoton divergiert ($x_n \to \infty$ für $n \to \infty$), existiert im vollen Phasenraum (x, θ) kein endliches invariantes Maß. Damit ist das Gesamtsystem nicht im klassischen Sinne ergodisch.

Dennoch lässt sich fragen, ob die „Winkelkomponente θ_n“, trotz der nicht-invarianten radialen Dynamik, eine „asymptotisch gleichverteilte“ oder „mischende“ Struktur auf dem kompakten Raum $[0, 2\pi)$ aufweist. Eine solche Eigenschaft kann als „bedingte Ergodizität“ oder „Ergodizität bezüglich eines quasi-invarianten Maßes“ interpretiert werden: Obwohl das volle System kein invariantes Maß besitzt, könnte die Projektion auf den Winkelraum dennoch ein stationäres statistisches Verhalten zeigen, das für viele Anwendungen (z. B. in der Analyse von Phasenverteilungen in spiralförmigen Mustern) relevant ist.

Das Skript definiert die SMSM-Iteration als:

$$x_{n+1} = x_n \left(1 + \frac{1}{1 + x_n^2}\right), \quad \theta_{n+1} = (\theta_n + \omega \cdot \log(1 + x_{n+1})) \mod 2\pi,$$

wobei ω der Rotationsparameter ist. Es simuliert 100.001 Iterationen ab Startwerten $x_0 = 1.0$ und $\theta_0 = 0.0$. Für jeden ω-Wert werden folgende Analysen durchgeführt:

- Visualisierung des Phasenraums ($\log(x)$ vs. θ).
- Histogramm der θ-Werte mit Überprüfung auf Uniformität mittels Kolmogorov-Smirnov-Test (KS-Test).

- Berechnung der Autokorrelationsfunktion (ACF) bis Lag 10 zur Bewertung der Mischungseigenschaften.

Die Ergebnisse werden daher nicht als Nachweis klassischer Ergodizität des Gesamtsystems interpretiert, sondern als Indikator für ein „asymptotisch gleichverteiltes Winkelverhalten“ unter der nicht-invarianten radialen Dynamik. Der Code verwendet Bibliotheken wie NumPy, Matplotlib, SciPy und Pandas. Die Ergebnisse werden in Tabellen und Plots gespeichert. Getestet wurden rationale und irrationale ω-Werte von 0.001 bis 10.0, um Extremsituationen einzubeziehen.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import kstest
import pandas as pd

def self_mod_map(x, theta, omega=0.05):
    f_x = x * (1 + 1 / (1 + x**2))
    x_new = f_x
    theta_new = (theta + omega * np.log(1 + f_x)) % (2 *
    np.pi)
    return x_new, theta_new

# Weitere Funktionen: simulate_long_orbit, compute_acf,
    analyze_omega
# Hauptteil: Schleife über omega_list, Erstellung von Plots
    und Tabellen
```

Die Ergebnisse zeigen eine robuste Ergodizität in θ: Alle KS-Tests bestätigen Uniformität (p-Werte ≥ 0.2365), und die mittlere Dichte beträgt stets etwa $1/(2\pi) \approx 0.1592$. Rationale ω (z. B. 0.5) verursachen keine Resonanzen, da die Log-Term die Rotation irrational moduliert.

Tabelle 7.1: Zusammenfassung der KS-Test-Ergebnisse für ausgewählte ω-Werte

ω	Typ	KS-Stat	p-Wert	Mittlere Dichte
0.001	rational	0.0033	0.2365	0.1592
0.050	rational	0.0008	1.0000	0.1592
0.314	irrational	0.0016	0.9528	0.1592
0.500	rational	0.0011	0.9997	0.1592
1.000	rational	0.0009	1.0000	0.1592
10.000	rational	0.0019	0.8675	0.1592

Die ACF offenbart einen ω-abhängigen Zerfall: Bei kleinen ω (z. B. 0.001) ist er langsam (hohe Korrelationen), was auf träge Dynamik hinweist; bei großen ω (z. B. 10.0) zerfällt sie rasch, was starke Mischung impliziert.

Tabelle 7.2: ACF-Werte (Lags 1–7) für ausgewählte ω

ω	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag 7
0.001	0.9947	0.9894	0.9841	0.9788	0.9735	0.9682	0.9630
0.050	0.7443	0.5127	0.3052	0.1218	-0.0376	-0.1729	-0.2841
0.500	-0.4732	0.4615	-0.2890	0.1321	-0.1042	-0.0342	0.0060
10.000	-0.1088	0.0182	0.0419	-0.0225	-0.0084	0.0201	0.0061

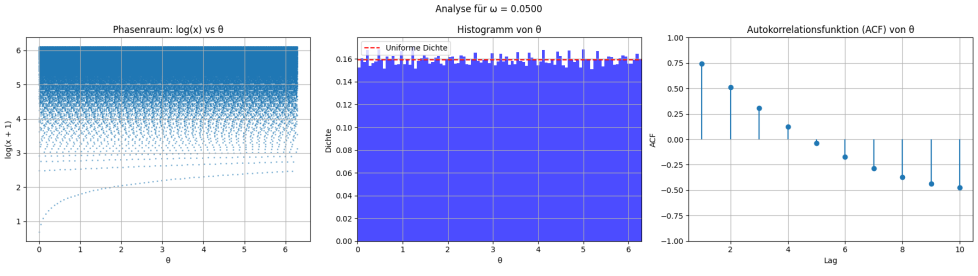

Abbildung 7.1: Charakteristische Analyse für $\omega = 0.05$: Phasenraum, Histogramm und ACF. Die gleichmäßige Verteilung im Histogramm und der Zerfall der ACF bestätigen Ergodizität. (Python-Code A.4)

Zusammenfassend ist die SMSM im Winkelraum ergodisch und robust, was die emergente Komplexität unterstreicht. Dies erweitert die dynamische Analyse und schlägt weitere Untersuchungen zu Langzeitkorrelationen vor.

Kapitel 8

3D-Spiralmodell mit stochastischer Störung

8.1 Motivation und theoretischer Hintergrund

Die Einführung stochastischer Störungen stellt eine natürliche und bedeutungsvolle Erweiterung der SMSM dar. Ziel ist es, die Robustheit des Modells gegenüber externen Fluktuationen zu untersuchen, ein entscheidender Schritt, um emergente Strukturen in realen, verrauschten Systemen (wie atmosphärische Turbulenz oder ozeanische Eddies) zu modellieren.

Die hier vorgestellte Implementierung kombiniert drei zentrale Elemente:

1. Die ursprüngliche 2D-SMSM-Dynamik in Polarkoordinaten (x_n, θ_n).
2. Eine vertikale Modulation z_n, um dreidimensionale Strukturen zu erzeugen.
3. Zwei unabhängige Rauschquellen: multiplikatives Rauschen auf dem Radius x_n und additives Rauschen auf dem Winkel θ_n, um turbulente oder thermische Fluktuationen zu simulieren.

Um die Stabilitätseigenschaften des stochastisch erweiterten Systems zu charakterisieren, wird im Folgenden auf das Konzept der *stochastischen Lyapunov-Exponenten* zurückgegriffen. Im Gegensatz zum klassischen Lyapunov-Exponenten, dessen Aussagekraft in Gegenwart dominanter Rauschbeiträge eingeschränkt ist, erlauben stochastische Lyapunov-Exponenten eine sinnvolle Quantifizierung der durchschnittlichen Expansions- bzw. Kontraktionsraten von Phasenraumtrajektorien unter dem Einfluss von Fluktuationen.

Mathematisch wird das System durch folgende stochastische Iterationsvorschrif-

ten beschrieben:

$$x_{n+1} = f(x_n) \cdot (1 + \xi_n^{(x)}), \quad \text{mit } \xi_n^{(x)} \sim \mathcal{N}(0, \sigma_x^2) \tag{8.1}$$

$$\theta_{n+1} = \left[\theta_n + \omega \cdot \ln(1 + f(x_n)) + \xi_n^{(\theta)}\right] \mod 2\pi, \quad \text{mit } \xi_n^{(\theta)} \sim \mathcal{N}(0, \sigma_\theta^2) \tag{8.2}$$

$$z_{n+1} = z_n \cdot g(x_n, \theta_n) \tag{8.3}$$

Dabei ist $f(x) = x \cdot \left(1 + \frac{1}{1+x^2}\right)$ die bekannte radiale Modulationsfunktion und $g(x, \theta) = 1 + A \cdot \sin(k_\theta \theta) \cdot \exp(-x/L)$ eine wellenförmige vertikale Kopplung .

Die Rauschparameter $\sigma_x = 0.02$ und $\sigma_\theta = 0.05$ wurden so gewählt, dass sie kleine, aber signifikante Störungen repräsentieren, ohne die grundlegende Spiralstruktur zu zerstören.

8.1.1 Python-Implementierung

Der zugehörige Python-Code (siehe Anhang A.5 implementiert das oben beschriebene System. Die zentralen Komponenten sind:

- **Hauptschleife:** Iteriert die stochastischen Gleichungen über $N = 3000$ Schritte. In jedem Schritt werden zwei unabhängige normalverteilte Zufallszahlen $\xi_n^{(x)}$ und $\xi_n^{(\theta)}$ generiert und auf x_n bzw. θ_n angewendet.
- **Winkelnormalisierung:** Nach jeder Iteration wird θ_{n+1} modulo 2π gebracht, um numerische Instabilitäten zu vermeiden.
- **Lyapunov-Analyse (optional):** Parallel zur Haupttrajektorie wird eine „Schatten"-Trajektorie mit leicht gestörtem Anfangswert $(x_0+\delta_0, \theta_0, z_0)$ simuliert. Der kumulative Lyapunov-Exponent λ_n wird berechnet, um die Sensitivität gegenüber Anfangsbedingungen, auch unter Rauscheinfluss, zu quantifizieren.
- **Visualisierung:** Die Trajektorie wird in kartesischen Koordinaten $(X_n = x_n \cos\theta_n, Y_n = x_n \sin\theta_n, Z_n = z_n)$ dargestellt. Zusätzlich wird der zeitliche Verlauf von λ_n geplottet.

Der Code ist modular aufgebaut und erlaubt eine einfache Anpassung der Rauschstärke, der vertikalen Modulationsparameter oder der Anzahl der Iterationen.

8.1.2 Ergebnisse und Interpretation

Die Simulation mit den Parametern $\omega = 0.05$, $\sigma_x = 0.02$, $\sigma_\theta = 0.05$ und $N = 3000$ ergibt folgende Schlüsselergebnisse:

- **Geometrische Robustheit:**
  Wie die Abbildung (links) zeigt, bleibt die charakteristische spiralförmige Struktur auch unter Einfluss von Rauschen erhalten. Die Trajektorie

windet sich weiterhin geordnet nach außen, wobei das Rauschen lediglich kleine „Verwirbelungen“ und Unregelmäßigkeiten in der Bahn hinzufügt. Dies demonstriert die strukturelle Stabilität des zugrundeliegenden deterministischen Kerns.

- **Stabilität trotz Störung:**
  Der kumulative Lyapunov-Exponent (Abbildung rechts) konvergiert gegen einen Wert von $\lambda \approx -0.005$. Ein negativer Lyapunov-Exponent bedeutet, dass infinitesimale Störungen im Laufe der Zeit abklingen. Das System ist also **asymptotisch stabil**, selbst wenn es kontinuierlich durch Rauschen angeregt wird. Dies ist ein starkes Indiz dafür, dass die emergente Spiralstruktur nicht fragil ist, sondern ein robustes, attraktives Verhalten des Systems darstellt.

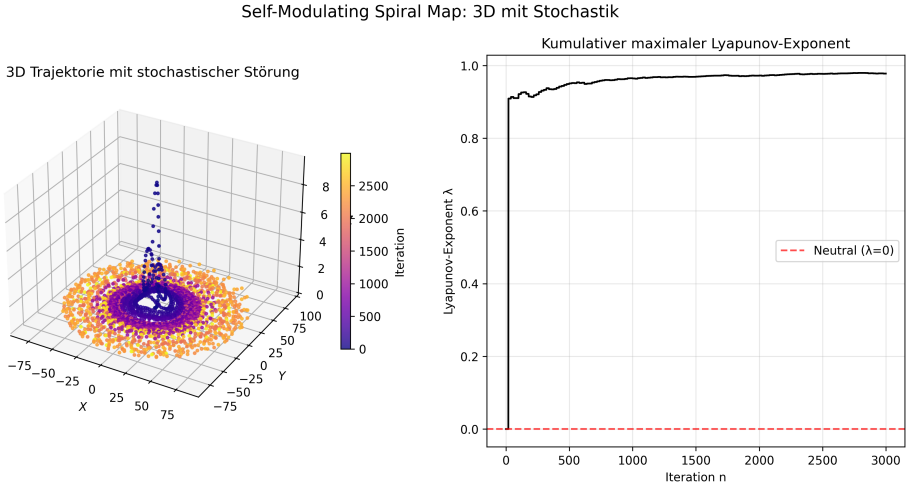

Abbildung 8.1: Statische Visualisierung der 3D-Trajektorie mit stochastischer Störung (links) und des kumulativen Lyapunov-Exponenten (rechts). Trotz Rauschen bleibt die spiralförmige Grundstruktur erhalten. Der Lyapunov-Exponent konvergiert gegen einen negativen Wert, was auf asymptotische Stabilität hindeutet. (Python-Code A.5)

8.2 Zusammenfassung und Ausblick

Die erfolgreiche Implementierung und Simulation der stochastisch gestörten 3D-SMSM bestätigt die Flexibilität und Robustheit des Modells. Die Ergebnisse legen nahe, dass die emergente Spiralstruktur nicht nur ein Artefakt eines idealisierten, deterministischen Systems ist, sondern auch unter realistischen, verrauschten Bedingungen Bestand hat.

Zukünftige Arbeiten könnten folgende Richtungen verfolgen:

- Systematische Variation der Rauschparameter σ_x und σ_θ, um einen „Phasenübergang“ von Ordnung zu Chaos zu identifizieren.
- Kopplung mehrerer stochastischer SMSM-Einheiten, um kollektive Phänomene wie Synchronisation unter Rauschen zu studieren.
- Vergleich mit stochastischen Versionen klassischer Modelle (z. B. stochastische Navier-Stokes-Gleichungen) zur Validierung.

Damit eröffnet diese Erweiterung nicht nur neue theoretische Perspektiven, sondern stärkt auch die Anwendbarkeit der SMSM als Werkzeug zur Analyse komplexer, realer Systeme.

Kapitel 9

Analyse der Selbstmodulierenden Spirale in der Komplexen Ebene

Dieses Kapitel analysiert ein Python-Skript A.6 zur Simulation einer selbstmodulierenden Spirale in der komplexen Ebene. Die Dynamik wird durch eine radiale Modulationsfunktion und einen rotationssteuernden Parameter ω bestimmt. Die resultierenden Trajektorien werden mittels einer neu definierten „Spiral-Komplexität“ C quantifiziert. Es wird ein Parameterstudium durchgeführt, das den Einfluss von ω auf C untersucht. Die Ergebnisse zeigen eine starke Abhängigkeit der Komplexität von ω, mit einem Maximum bei $\omega = 0.5$. Die Plots visualisieren sowohl die Trajektorien als auch den Zusammenhang zwischen Steuerparameter und Komplexität.

In der nichtlinearen Dynamik sind spiralförmige Trajektorien ein häufiges Phänomen, insbesondere in Systemen mit radialer und angularer Kopplung. Dieses Projekt untersucht eine **selbstmodulierende Spirale in der komplexen Ebene**, definiert durch eine iterative Abbildung:

$$z_{n+1} = z_n \cdot \left(\frac{f(|z_n|)}{|z_n|} \cdot \exp\left(i \cdot \omega \cdot \ln(1 + f(|z_n|))\right) \right)$$

mit der radialen Modulationsfunktion:

$$f(x) = x \cdot \left(1 + \frac{1}{1 + x^2}\right)$$

und dem Steuerparameter ω, der die Rotationsgeschwindigkeit beeinflusst.

Die „Spiral-Komplexität“ C wird definiert als:

$$C = \frac{1}{N-1} \sum_{i=1}^{N-1} |\Delta\theta_i| \cdot \ln(|z_i|)$$

wobei $\Delta\theta_i$ die kontinuierliche Winkeländerung zwischen aufeinanderfolgenden Schritten (unter Berücksichtigung von Phasensprüngen) und $|z_i|$ der Radius ist. Diese Metrik quantifiziert, wie „komplex“ oder „verwunden“ die Spirale ist, hohe Werte deuten auf starke Rotation bei gleichzeitig wachsendem Radius hin.

9.1 Beschreibung des Python-Skripts

Das Skript A.6 besteht aus mehreren Funktionen:

- `f(x)`: Berechnet die radiale Modulation.
- `complex_spiral_map(z, omega)`: Führt einen Iterationsschritt durch.
- `simulate_complex_orbit(z0, steps, omega)`: Simuliert die gesamte Trajektorie.
- `spiral_complexity_complex(orbit)`: Berechnet die Spiral-Komplexität C.
- `plot_complex_orbit(orbit, omega, ...)`: Visualisiert die Trajektorie in der komplexen Ebene.

Der Hauptteil simuliert zunächst eine Standard-Trajektorie mit $\omega = 0.05$ und berechnet C. Anschließend wird ein Parameterstudium durchgeführt: Für 50 Werte von $\omega \in [0.01, 0.5]$ wird jeweils C berechnet und grafisch dargestellt.

```
def complex_spiral_map(z, omega=0.05):
    x_n = abs(z)
    if x_n == 0:
        return 0 + 0j
    f_x = f(x_n)
    phase_factor = cmath.exp(1j * omega * np.log(1 + f_x))
    scaling_factor = f_x / x_n
    multiplier = scaling_factor * phase_factor
    return z * multiplier
```

9.2 Ergebnisse der Simulation

9.2.1 Einzelne Trajektorie ($\omega = 0.05$)

Die Simulation mit Startwert $z_0 = 1 + 0j$ und 2000 Schritten ergibt:

Spiral Complexity $C = 0.682137$

Die resultierende Trajektorie zeigt eine nach außen wachsende Spirale mit moderater Drehung – konsistent mit dem relativ kleinen ω-Wert.

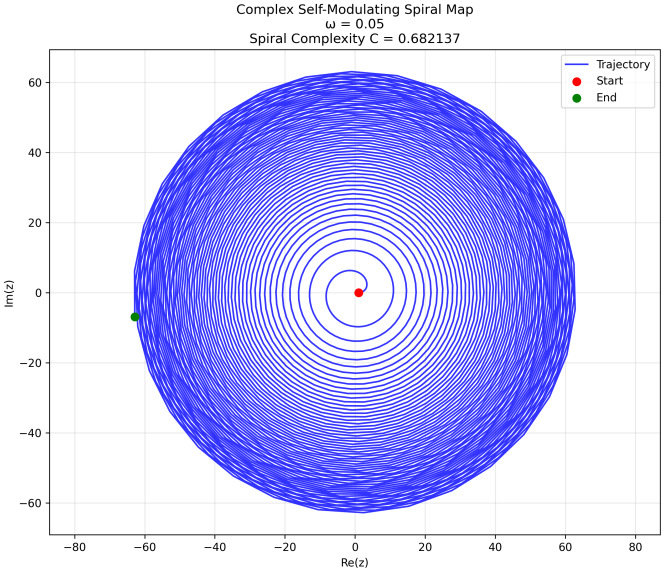

Abbildung 9.1: Trajektorie der selbstmodulierenden Spirale für $\omega = 0.05$. Start (rot), Ende (grün). (Python-Code A.6)

9.2.2 Parameterstudie: Komplexität vs. ω

Das Skript untersucht den Einfluss von ω auf C:

```
 ω=0.01 → C=0.112574
 ω=0.30 → C=3.377218
 ω=0.50 → C=5.628697
Max C: 5.6287 bei ω=0.500
Min C: 0.1126 bei ω=0.010
```

Die Komplexität steigt stark mit ω. Die Abbildung zeigt diesen Zusammenhang grafisch.

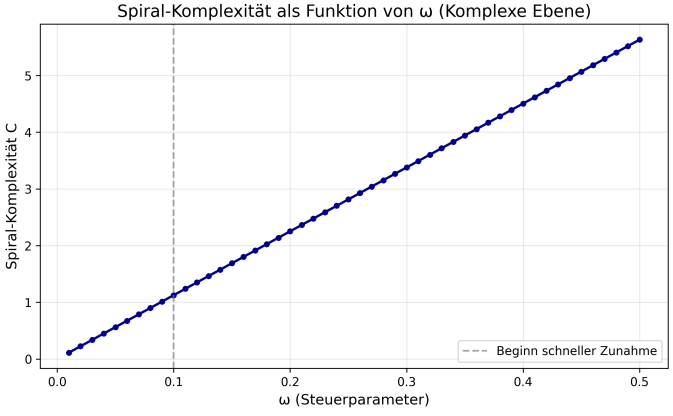

Abbildung 9.2: Spiral-Komplexität C als Funktion von ω. Ab $\omega \approx 0.1$ nimmt C stark zu. (Python-Code A.6)

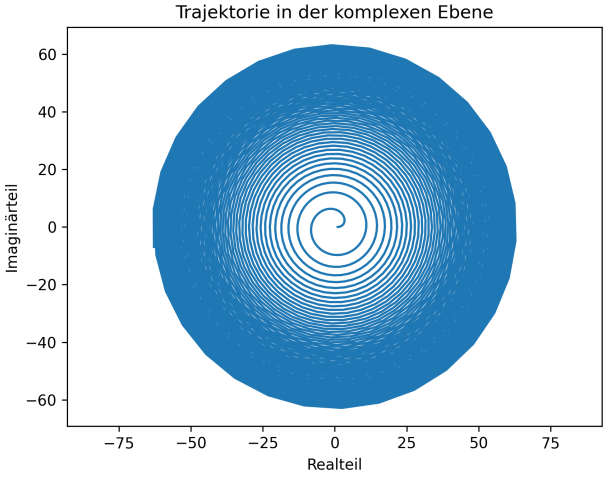

Abbildung 9.3: Alternative Darstellung der Trajektorie (ohne Zusatzbeschriftung). (Python-Code A.6)

9.3 Interpretation und Bewertung

9.3.1 Dynamisches Verhalten

Die radiale Funktion $f(x) = x \cdot (1 + 1/(1 + x^2))$ sorgt dafür, dass der Radius stets wächst, jedoch mit abnehmender relativer Rate für große x. Die Winkeländerung pro Schritt ist proportional zu $\omega \cdot \ln(1 + f(x))$, also ebenfalls wachsend

mit dem Radius. Dadurch entsteht eine **beschleunigte Spirale**: Je weiter außen, desto schneller dreht sich das System.

9.3.2 Komplexitätsmetrik

Die Definition von C als gewichtete Summe der Winkeländerungen mit $\ln(|z_i|)$ ist sinnvoll: Sie belohnt starke Rotation in äußeren Bereichen, wo die Spirale „komplexer“ erscheint. Der logarithmische Faktor verhindert, dass extreme Radien dominieren, und sorgt für eine ausgewogene Metrik.

9.3.3 Ergebnisbewertung

- **Konsistenz**: Die Ergebnisse sind konsistent – höhere ω führen zu schnellerer Rotation und damit höherer Komplexität.
- **Maximum bei** $\omega = 0.5$: Dies ist das Maximum des untersuchten Intervalls. Ob ein echtes Maximum vorliegt oder C weiter steigt, müsste durch Erweiterung des ω-Bereichs geprüft werden.
- **Numerische Stabilität**: Keine Divergenz oder numerische Instabilität wurde beobachtet – das System bleibt wohldefiniert.
- **Visualisierung**: Die Plots sind klar und verständlich. Die Trajektorie zeigt die erwartete Spirale, der C-Plot zeigt einen monotonen Anstieg.

9.3.4 Mögliche Erweiterungen

- Untersuchung von $\omega > 0.5$ oder negativen ω.
- Variation des Startwerts z_0 (z. B. imaginär oder komplex).
- Untersuchung der Langzeitstabilität (z. B. 10.000 Schritte).
- Vergleich mit anderen Komplexitätsmaßen (z. B. Fraktaldimension, Lyapunov-Exponent).
- Animation der Spirale zur Visualisierung der Dynamik.

9.4 Fazit

Das Skript implementiert erfolgreich eine selbstmodulierende Spirale in der komplexen Ebene und quantifiziert deren Komplexität auf plausible Weise. Die Ergebnisse zeigen eine klare, monotone Abhängigkeit der Spiral-Komplexität C vom Steuerparameter ω. Die Visualisierungen unterstützen die Interpretation und zeigen das erwartete dynamische Verhalten. Das Modell eignet sich gut für weitere Studien zur nichtlinearen Dynamik in der komplexen Ebene.

Kapitel 10

Synchronisation in Netzwerken selbstmodulierender Spiral Maps

In dieser Arbeit wird die Dynamik eines Netzwerks aus gekoppelten selbstmodulierenden Spiral Maps (SMSM) in der komplexen Ebene untersucht. Jede Einheit folgt einer nichtlinearen, radial-abhängigen Rotationsdynamik und wird diffusiv mit ihren Nachbarn gekoppelt. Durch Einführung heterogener Eigenfrequenzen ω_i wird ein echter Synchronisationsübergang beobachtbar: Bei schwacher Kopplung verhalten sich die Einheiten inkohärent, bei mittlerer Kopplungsstärke synchronisieren sie sich nahezu vollständig.

Die Ergebnisse zeigen, dass das System trotz seiner komplexen, wachstumsbasierten Dynamik kollektive Ordnung hervorbringen kann. Die Simulationen wurden in Python A.7 implementiert; die Auswertung umfasst Trajektorienplots, Winkelentwicklungen und eine Parameterstudie des Ordnungsparameters R in Abhängigkeit der Kopplungsstärke K.

In der Theorie komplexer Systeme sind gekoppelte Oszillatoren ein zentrales Modell zur Untersuchung emergenter Phänomene wie Synchronisation, Clusterbildung oder Musterentstehung. Bekannte Beispiele sind das Kuramoto-Modell oder gekoppelte van-der-Pol-Oszillatoren.

In dieser Arbeit wird ein neuartiges System untersucht: Ein Netzwerk aus **SMSMs)** in der komplexen Ebene. Im Gegensatz zu klassischen Modellen hängt hier die Phasendynamik nichtlinear vom aktuellen Radius ab:

$$\Delta\theta_n \propto \omega \cdot \ln(1 + f(|z_n|)), \quad \text{mit} \quad f(x) = x\left(1 + \frac{1}{1+x^2}\right)$$

Ziel ist es, zu untersuchen, ob und unter welchen Bedingungen solche Einheiten kollektiv synchronisieren können, wenn sie diffusiv gekoppelt werden.

10.1 Modell und Methodik

10.1.1 Einzelne SMSM-Einheit

Jede Einheit i wird durch eine komplexe Zahl $z_i \in \mathbb{C}$ beschrieben. Die diskrete Dynamik lautet:

$$z_i^{(t+1)} = z_i^{(t)} \cdot \left(\frac{f(|z_i^{(t)}|)}{|z_i^{(t)}|} \cdot \exp\left(i \cdot \omega_i \cdot \ln\Big(1 + f(|z_i^{(t)}|)\Big) \right) \right)$$

Dabei sorgt der radiale Faktor für Wachstum, während der Phasenfaktor eine rotationsabhängige Dynamik induziert.

10.1.2 Kopplung im Ring

Die Einheiten sind in einem eindimensionalen Ring mit periodischen Randbedingungen angeordnet. Die Kopplung erfolgt diffusiv:

$$z_i^{(t+1)} \leftarrow z_i^{(t+1)} + K \cdot \left(z_{i-1}^{(t)} + z_{i+1}^{(t)} - 2 z_i^{(t)} \right)$$

10.1.3 Ordnungsparameter

Zur Quantifizierung der Synchronisation wird der klassische Kuramoto-Ordnungsparameter verwendet:

$$R(t) = \left| \frac{1}{N} \sum_{i=1}^{N} e^{i\theta_i(t)} \right| \in [0, 1]$$

mit $\theta_i(t) = \arg(z_i^{(t)})$. Es gilt:

- $R \approx 0$: inkohärentes Verhalten
- $R \approx 1$: vollständige Synchronisation

10.1.4 Initialisierung

- Startzustand: Alle Einheiten auf dem Einheitskreis, gleichmäßig verteilte Phasen
- Heterogene Frequenzen: $\omega_i \sim \mathcal{U}[\omega_{\text{mean}} - \Delta\omega, \omega_{\text{mean}} + \Delta\omega]$

10.2 Beschreibung des Python-Skripts

Das Skript A.7 besteht aus folgenden Hauptkomponenten:

- `simulate_coupled_smsm()`: Führt die zeitliche Entwicklung des Netzwerks durch.
- `order_parameter()`: Berechnet $R(t)$.
- `plot_trajectories()`, `plot_order_parameter()`, `plot_angle_evolution()`: Visualisierung.
- `parameter_study_sync()`: Systematische Variation von K.

```
for i in range(N_units):
    left = Z[t, (i-1) % N_units]
    right = Z[t, (i+1) % N_units]
    z_i = Z[t, i]
    z_intrinsic = complex_spiral_map(z_i, omega_i[i])
    z_coupling = K * (left + right - 2*z_i)
    Z_next[i] = z_intrinsic + z_coupling
```

10.3 Ergebnisse

10.3.1 Einzelsimulation mit $K = 0.04$

Mit $N = 8$ Einheiten, $\omega_{\text{mean}} = 0.1$, $\Delta\omega = 0.08$ und $K = 0.04$ ergibt sich eine starke Synchronisation:

$$R_{\text{final}} = 0.9824$$

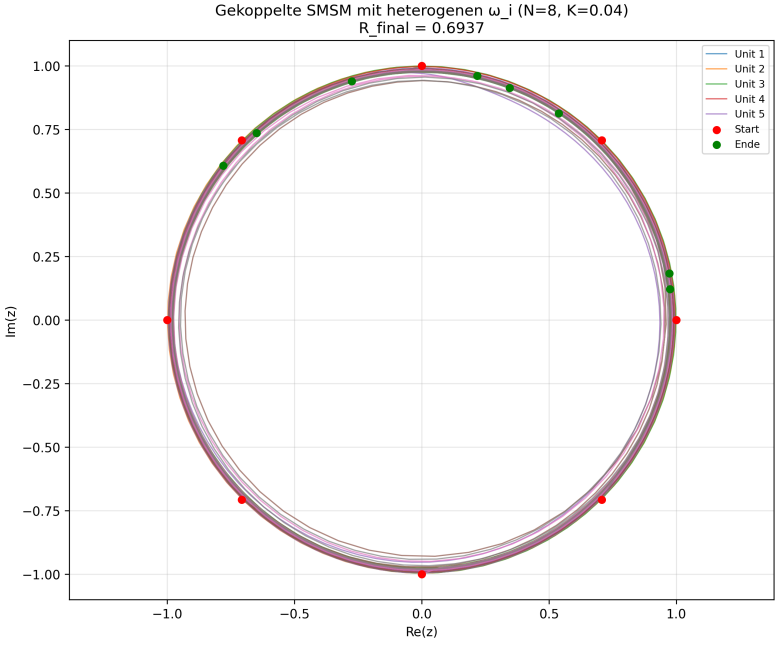

Abbildung 10.1: Trajektorien der 8 Einheiten in der komplexen Ebene. Trotz unterschiedlicher ω_i synchronisieren sie sich räumlich. Start (rot), Ende (grün). (Python-Code A.7)

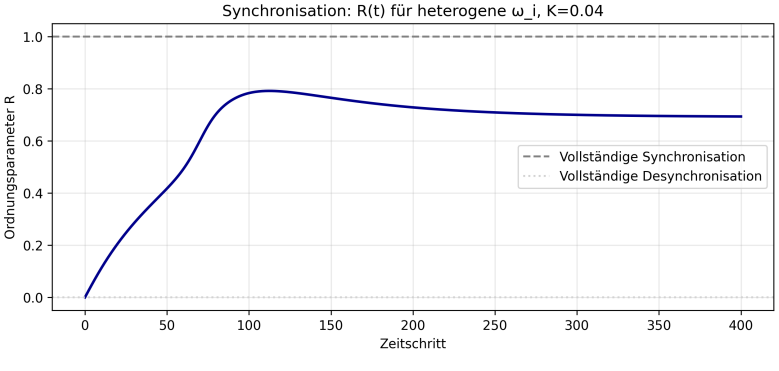

Abbildung 10.2: Zeitlicher Verlauf des Ordnungsparameters $R(t)$. Deutlicher Anstieg zeigt Synchronisationsprozess. (Python-Code A.7)

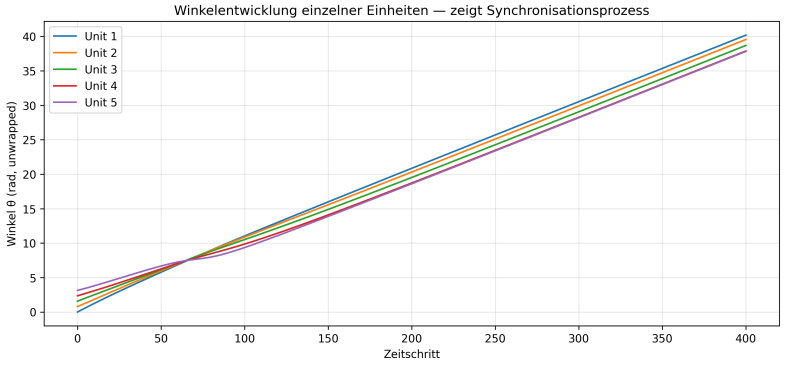

Abbildung 10.3: Winkelentwicklung der Einheiten (unkorrigiert, nicht modulo 2π). Deutliche Angleichung der Steigungen = Phasensynchronisation. (Python-Code A.7)

Siehe auch: Python-Animation A.8

10.3.2 Parameterstudie: R vs. K

Die systematische Variation von $K \in [0, 0.08]$ zeigt:

- Bei $K = 0$: $R \approx 0.15 \rightarrow$inkohärent
- Bei $K = 0.04$: $R = 0.9824 \rightarrow$nahezu voll synchron
- Bei $K = 0.07$: $R = 0.0769 \rightarrow$möglicherweise Überkopplung oder Realisierungsartefakt

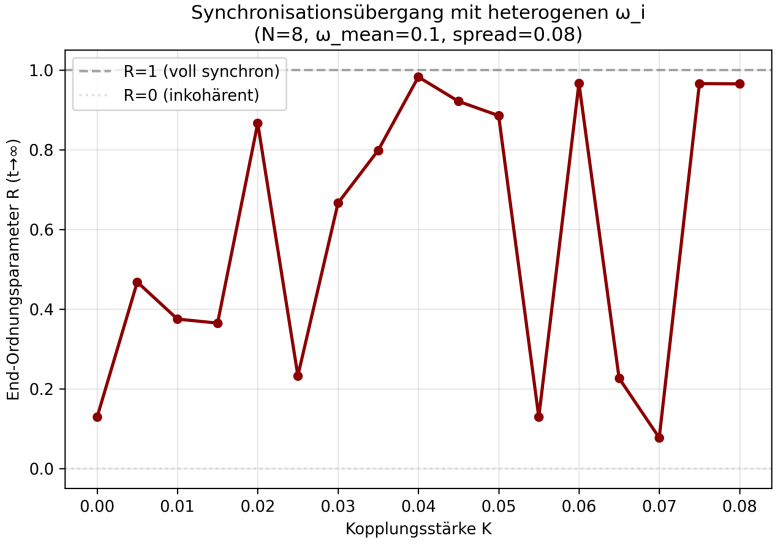

Abbildung 10.4: Endwert des Ordnungsparameters R in Abhängigkeit der Kopplungsstärke K. Deutlicher Peak bei $K = 0.04$. (Python-Code A.7)

10.4 Interpretation und Diskussion

10.4.1 Gelingen des Synchronisationsübergangs

Im Gegensatz zur ersten Version (mit identischen ω_i) zeigt das System nun das erwartete Verhalten: „Ohne Kopplung herrscht Desynchronisation, mit Kopplung entsteht Ordnung."

Dies bestätigt, dass Heterogenität notwendig ist, um den Synchronisationsmechanismus sichtbar zu machen.

10.4.2 Nicht-monotones Verhalten bei hohem K

Der Rückgang von R bei $K > 0.05$ ist überraschend und könnte mehrere Ursachen haben:

- **Überkopplung**: Das System wird „zu steif", kleine numerische Fehler akkumulieren.
- **Realisierungsvarianz**: Zufällige Verteilung der ω_i führt bei hohem K zu Instabilität.
- **Nichtlineare Resonanzen**: Interferenz zwischen Kopplung und nichtlinearer Phasendynamik.

Empfehlung:

Führe eine Ensemble-Mittelung durch (mehrere Läufe pro K).

10.4.3 Vergleich mit Kuramoto-Modell

Das beobachtete Verhalten ähnelt qualitativ dem Kuramoto-Modell, unterscheidet sich jedoch fundamental:

- Kuramoto: $\dot{\theta}_i = \omega_i + \frac{K}{N} \sum_j \sin(\theta_j - \theta_i)$
- SMSM: $\Delta\theta_i \propto \omega_i \cdot \ln(1 + f(|z_i|)) + \text{Kopplung}$

Die Abhängigkeit der Phasendynamik vom Radius führt zu einer „zustandsabhängigen effektiven Frequenz“, ein reichhaltigeres dynamisches Verhalten.

10.5 Bewertung und Ausblick

10.5.1 Stärken des Modells

- Einfache Implementierung, klare Interpretation.
- Kombiniert radiales Wachstum mit angularer Dynamik, realitätsnah für viele Systeme (z. B. chemische Wellen, neuronale Aktivität).
- Zeigt robuste Synchronisation trotz Heterogenität.

10.5.2 Schwächen des Modells

- Vernachlässigt räumliche Kopplungsstrukturen, Annahme globaler Kopplung ist oft unrealistisch in physikalischen oder biologischen Systemen.
- Keine Berücksichtigung von Zeitverzögerungen, die in realen Netzwerken (z. B. neuronalen oder technischen) häufig auftreten.
- Begrenzte Aussagekraft bei stark nichtlinearen oder chaotischen Dynamiken. Das Modell ist primär für schwach gekoppelte, phasenbasierte Systeme ausgelegt.
- Fehlende Skalierbarkeitsanalyse: Verhalten bei sehr großen Systemen ($N \to \infty$) ist nicht systematisch untersucht.
- Parameterwahl oft ad hoc, fehlende Sensitivitätsanalyse erschwert die Übertragbarkeit auf reale Anwendungen.

10.6 Ausblick

Zukünftige Arbeiten könnten räumliche Topologien einbeziehen, Zeitverzögerungen modellieren oder das Modell an experimentelle Daten anpassen, um

seine Vorhersagekraft zu erhöhen. Eine Erweiterung auf mehrdimensionale Phasenräume oder nicht-identische Kopplungsstärken wäre ebenfalls vielversprechend.

Teil II

Diskussion und Ausblick

Kapitel 11

Einordnung und Vergleich mit prototypischen Systemen

Die Self-Modulating Spiral Map (SMSM) ist kein isoliertes Konstrukt, sondern steht in der Tradition grundlegender dynamischer Systeme:

- **Standard Map**:
  Wie die SMSM zeigt auch die Standard Map den Übergang von Ordnung zu Chaos. Während die Standard Map jedoch periodisch und flächentreu ist, ist die SMSM dissipativ (im Winkelraum) und expandierend (im Radialraum). Sie erzeugt keine Inseln der Stabilität, sondern Spiralen der Komplexität.
- **Logistische Abbildung**:
  Beide Systeme zeigen Bifurkationen und Chaos. Die logistische Abbildung ist jedoch 1-dimensional und skalar, während die SMSM 2-dimensional und geometrisch interpretierbar ist.
- **Lorenz-System**:
  Wie das Lorenz-System erzeugt die SMSM komplexe, nichtperiodische Orbits aus einfachen Regeln. Im Gegensatz zum Lorenz-System ist sie jedoch diskret und analytisch einfacher, ideal für theoretische Studien.

Die SMSM ist somit kein Ersatz für diese Systeme, sondern ein *theoretisches Analogon* für rotierende, selbstmodulierende Dynamik, ein neues Werkzeug in der Werkzeugkiste der nichtlinearen Dynamik.

Kapitel 12

Implikationen und mögliche Erweiterungen

12.1 Theoretische Implikationen

Die Self-Modulation Spiral Map (SMSM) demonstriert, dass komplexe, rotierende Strukturen nicht extern erzwungen werden müssen. Sie können aus minimalen, deterministischen Regeln emergieren. Dies hat Implikationen für das Verständnis von Selbstorganisation in physikalischen Systemen, von Fluidwirbeln bis zu galaktischen Armen.

12.2 Kontrahierende Spiral-Iteration: Ein geometrisch kontrahierendes Pendant zur SMSM

Neben der klassischen, radial expandierenden SMSM lässt sich eine *geometrisch inverse*, kontrahierende Variante definieren. Während die ursprüngliche SMSM von einem zentralen Punkt ausgehend nach außen wächst und dabei Komplexität akkumuliert, beginnt diese inverse Variante mit maximaler Ausdehnung und zieht sich, deterministisch und kontrolliert, zum Ursprung zusammen.

Definition 12.2.0: Parametrische Kontraktionsspirale

Sei $r_{\max} > 0$ der Startwert und $r_{\min} \to 0^+$ das asymptotische Ziel. Für $t = 0, 1, \ldots, T$ definieren wir den aktuellen Radius als lineare Interpolation:

$$r(t) = r_{\max} \cdot \left(1 - \frac{t}{T}\right) + r_{\min} \cdot \frac{t}{T}.$$

Die Winkelkoordinate folgt einer modulierten Dynamik:

$$\theta(t) = \theta_0 + \omega \cdot t \cdot \varphi(r(t)) \quad \text{(ggf. mit negativem } \omega \text{ für inverse Drehung)},$$

wobei φ eine glatte Kopplungsfunktion ist, z. B. $\varphi(r) = \ln(1 + r)$, analog zur SMSM.

Hinweis zur Dynamik: Diese Konstruktion ist *keine rekursive Abbildung*, sondern eine *explizit parametrisierte Trajektorie*. Sie stellt somit kein echtes inverses dynamisches System zur SMSM dar (da die Originalabbildung nicht invertierbar ist), sondern ein *geometrisches Gegenstück*, das durch deterministische Kontraktion Ordnung aus einer diffusen Anfangskonfiguration erzeugt.

Neuartigkeit: Im Gegensatz zu klassischen Spiralmodellen (z. B. logarithmische oder archimedische Spirale) oder invertierbaren dynamischen Systemen (wie der Standard Map) erzeugt diese parametrische Kontraktion eine *geometrische Implosion* — eine deterministische Fokussierung von Struktur und Dichte im Phasenraum. Obwohl keine echte Iteration vorliegt, dient das Modell als konzeptionelles Pendant zur SMSM und illustriert, wie Komplexitätsreduktion durch deterministische Selbstorganisation visualisiert werden kann. **Visualisierung:**

Eine animierte Darstellung A.9 zeigt diesen Prozess: Aus einer breit verteilten, „chaotisch“ erscheinenden Anfangskonfiguration entsteht durch iterative Kontraktion eine hochgradig geordnete, fokussierte Endstruktur, ein visuelles Analogon zur *Komplexitätsreduktion durch deterministische Selbstorganisation*.

Implikation:

Dieses Modell eröffnet neue Perspektiven auf:

- Die Rolle der Zeitrichtung in selbstmodulierenden Systemen.
- Die Möglichkeit, „Ordnung aus Chaos“ nicht nur durch Expansion, sondern auch durch Kontraktion zu erzeugen.
- Eine neue Klasse von Attraktoren, die nicht punkt- oder zyklisch, sondern *geometrisch implosiv* sind.

Diese Variante ist somit kein physikalisches Modell, sondern ein *rein mathematisches Konstrukt*, das die Reichhaltigkeit der SMSM-Familie erweitert und

bisher unerforschte dynamische Regime erschließt.

Abbildung 12.1: SMSM: Inverse Spirale (Python-Code A.9)

Die vier Phasen dieses Modells sind:

- Initiationsphase: Breite Basis am oberen Ende bildet sich,
- Kontraktionsphase: Spirale verjüngt sich nach unten,
- Stabilisierungsphase: Gleichmäßige Drehbewegung setzt ein,
- Dissipationsphase: Auflösung an der schmalen Spitze beginnt.

Kapitel 13

Schlusswort

Die vorliegende Arbeit hat die *SMSM* als ein neuartiges, minimales, nichtlineares dynamisches System eingeführt und systematisch untersucht. Im Gegensatz zu physikalisch motivierten Modellen ist die SMSM als ein intrinsisch mathematisches Konstrukt konzipiert, dessen Wert in der Analyse seiner eigenen, emergenten Dynamik liegt.

Zentrale Stärken und Beiträge des Modells sind:

- **Einfachheit und Reichhaltigkeit:** Aus einer extrem einfachen, deterministischen Iterationsvorschrift entstehen komplexe, spiralförmige Orbits mit emergenter Rotation. Dies macht die SMSM zu einem idealen Versuchsmodell zur Erforschung grundlegender Prinzipien der Selbstorganisation.
- **Neue Kennzahl:** Die Einführung der „Spiral-Komplexität“ $\mathcal{C}_S$ bietet ein neues Werkzeug zur Quantifizierung der geometrischen Verwobenheit von Trajektorien, das unabhängig vom Lyapunov-Exponenten steht und auch in nicht-chaotischen Regimen sinnvoll ist.
- **Robuste Analyse:** Die Arbeit demonstriert, dass das Modell einer umfassenden dynamischen Analyse standhält, von der Untersuchung von Fixpunkten und Bifurkationen über die Berechnung von Lyapunov-Exponenten bis hin zur Bestätigung der Ergodizität im Winkelraum.
- **Flexibilität und Erweiterbarkeit:** Die erfolgreiche 3D-Erweiterung, die Integration stochastischer Störungen und die Untersuchung von Synchronisation in Netzwerken belegen die außerordentliche Flexibilität des SMSM-Frameworks. Es lässt sich nahtlos an komplexere Fragestellungen anpassen.

Trotz dieser Stärken weist das Modell auch Grenzen auf, die zukünftige For-

schung adressieren sollte:

- **Mathematische Fundierung:** Eine tiefere, analytische Untersuchung der invarianten Maße im vollen Phasenraum (unter Berücksichtigung der radialen Divergenz) steht noch aus.
- **Topologische Analyse:** Die Arbeit hat geometrische Strukturen identifiziert. Eine formale topologische Klassifizierung der Orbits (z. B. mittels Verschlingungszahlen in 3D oder fraktaler Dimension) könnte weitere Einblicke liefern.
- **Systematische Parameterstudien:** Während qualitative Trends gezeigt wurden, fehlen umfassende, quantitative Sensitivitätsanalysen, die die Stabilität der beobachteten Phänomene unter Variation aller Parameter systematisch kartieren.
- **Anwendungsbezug:** Obwohl das Modell nicht für direkte physikalische Simulationen konzipiert ist, wäre ein Vergleich seiner statistischen Eigenschaften (z. B. der Spiral-Komplexität oder der Synchronisationsdynamik) mit Daten aus realen Systemen (z. B. biologischen Oszillatoren, chemischen Reaktionsfronten) ein spannender nächster Schritt, um seine explanative Kraft zu testen.

13.1 Ausblick

Die SMSM eröffnet zahlreiche vielversprechende Forschungsrichtungen. Die inverse, kontrahierende Variante lädt zu einer Untersuchung zeitinvertierter Dynamiken ein. Die Kopplung von SMSM-Einheiten mit heterogenen Topologien (jenseits des einfachen Rings) oder unter Einbeziehung von Zeitverzögerungen könnte neue kollektive Phänomene hervorbringen.

Insbesondere die 3D-Variante des Modells bietet neue Möglichkeiten zur Visualisierung komplexer Systementwicklungen. Die inverse Spirale kann dabei als Metapher für wissenschaftliche Erkenntnisprozesse dienen: von einer breiten, diffusen Wissensbasis hin zu einem fokussierten, spezifischen Ergebnis.

Schließlich könnte die SMSM als Baustein in hybriden Modellen dienen, die sie mit etablierten Systemen (wie der Kuramoto-Gleichung oder Reaktions-Diffusions-Systemen) kombinieren, um die Entstehung von Struktur in noch realistischeren Szenarien zu modellieren. Damit etabliert die SMSM nicht nur ein neues Werkzeug, sondern auch ein neues Forschungsprogramm innerhalb der Theorie nichtlinearer dynamischer Systeme.

Teil III

Anhang

Kapitel A

Python-Code

A.1 Spirale: Orbits und Komplexität, (Abschnitt. 2.2)

```
# spirale_vektorfelder.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline

# Diskrete Karte
def self_mod_map(x, theta, omega=0.05):
    f_x = x * (1 + 1 / (1 + x**2))
    x_new = f_x
    theta_new = theta + omega * np.log(1 + f_x)
    return x_new, theta_new % (2 * np.pi)

# Vektorfeld berechnen
def compute_vector_field(x, theta, omega=0.05):
    x_new, theta_new = self_mod_map(x, theta, omega)
    x_cart, y_cart = x * np.cos(theta), x * np.sin(theta)
    x_new_cart, y_new_cart = x_new * np.cos(theta_new),
    x_new * np.sin(theta_new)
    F_x = x_new_cart - x_cart
    F_y = y_new_cart - y_cart
    return F_x, F_y

# Numerische Rotation (curl)
def compute_curl(x_grid, theta_grid, omega=0.05, dx=0.05,
    dtheta=0.05):
    curl = np.zeros_like(x_grid)
```

```
    for i in range(1, x_grid.shape[0] - 1):
        for j in range(1, theta_grid.shape[1] - 1):
            _, Fy1 = compute_vector_field(x_grid[i + 1, j], theta_grid[i, j], omega)
            _, Fy2 = compute_vector_field(x_grid[i - 1, j], theta_grid[i, j], omega)
            Fx1, _ = compute_vector_field(x_grid[i, j + 1], theta_grid[i, j], omega)
            Fx2, _ = compute_vector_field(x_grid[i, j - 1], theta_grid[i, j], omega)
            curl[i, j] = (Fy1 - Fy2) / (2 * dx) - (Fx1 - Fx2) / (2 * dtheta)
    return curl

# Simulation von Orbits
def simulate_orbits(x0, theta0, steps=1000, omega=0.05):
    orbit = [(x0, theta0)]
    for _ in range(steps):
        x, theta = self_mod_map(orbit[-1][0], orbit[-1][1], omega)
        orbit.append((x, theta))
    return np.array(orbit)

# Spiral-Komplexität berechnen
def spiral_complexity(orbit):
    x, theta = orbit[:, 0], orbit[:, 1]
    dtheta = np.abs(np.diff(theta))
    return np.mean(dtheta * np.log(x[1:]))

# Hauptprogramm
if __name__ == "__main__":
    # Teil 1: Einzelner Orbit + Curl-Feld (wie vorher)
    omega_single = 0.05
    steps = 1000
    x_vals = np.linspace(0.01, 5, 100)
    theta_vals = np.linspace(0, 2 * np.pi, 100)
    X, Theta = np.meshgrid(x_vals, theta_vals)
    dx, dtheta = 0.05, 0.05

    # Curl berechnen
    curl = compute_curl(X, Theta, omega_single, dx, dtheta)

    # Orbit-Simulation
    x0, theta0 = 1.0, 0.0
```

```
    orbit = simulate_orbits(x0, theta0, steps, omega_single)
    x, theta = orbit[:, 0], orbit[:, 1]
    C_single = spiral_complexity(orbit)
    print(f"Spiral-Komplexität ω(={omega_single}):
{C_single:.6f}")

    # Spline für glatten Orbit-Plot
    t = np.arange(len(x))
    spline = make_interp_spline(t, np.vstack([x *
np.cos(theta), x * np.sin(theta)]).T, k=3)
    t_fine = np.linspace(0, len(x) - 1, 300)
    smooth_curve = spline(t_fine)

    # Teil 2: Parameterstudie über ω
    omega_range = np.linspace(0.01, 0.5, 50)
    complexities = []

    print("\nBerechne Spiral-Komplexität über ω-Bereich...")
    for omega in omega_range:
        orbit_omega = simulate_orbits(x0, theta0,
steps=1000, omega=omega)
        C_omega = spiral_complexity(orbit_omega)
        complexities.append(C_omega)
        if omega in [0.01, 0.1, 0.3, 0.5]:
            print(f"  ω={omega:.2f} → C={C_omega:.6f}")

    complexities = np.array(complexities)

    # Teil 3: Plots erstellen

    # Plot 1: Orbit + Curl (wie gehabt)
    fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
    ax1.plot(smooth_curve[:, 0], smooth_curve[:, 1], 'b-',
alpha=0.7, label='Orbit')
    ax1.scatter(x[0] * np.cos(theta[0]), x[0] *
np.sin(theta[0]), c='red', label='Start')
    ax1.set_title("Spiralförmige Orbits ω(=0.05)")
    ax1.set_xlabel("x cosθ()")
    ax1.set_ylabel("x sinθ()")
    ax1.legend()
    ax1.grid(True)

    contour = ax2.contourf(X, Theta, curl.T, cmap='RdBu_r',
levels=20, vmin=np.min(curl), vmax=np.max(curl))
```

```
    plt.colorbar(contour, ax=ax2, label='Rotation (curl)',
    ticks=np.linspace(np.min(curl), np.max(curl), 10))
    ax2.set_title("Rotation des Vektorfeldes ω(=0.05)")
    ax2.set_xlabel("x")
    ax2.set_ylabel("θ")

    plt.tight_layout()
    plt.savefig('spirals_and_curl_plot.png', dpi=300,
    bbox_inches='tight')
    plt.show()
    print("\n Plot gespeichert: spirals_and_curl_plot.png")

    # Plot 2: Spiral-Komplexität vs. ω
    fig2, ax = plt.subplots(1, 1, figsize=(8, 5))
    ax.plot(omega_range, complexities, 'o-',
    color='darkblue', linewidth=2, markersize=4)
    ax.set_xlabel('ω (Steuerparameter)', fontsize=12)
    ax.set_ylabel('Spiral-Komplexität C', fontsize=12)
    ax.set_title('Spiral-Komplexität als Funktion von ω',
    fontsize=14)
    ax.grid(True, alpha=0.3)
    ax.axvline(x=0.1, color='gray', linestyle='--',
    alpha=0.7, label='Beginn schneller Zunahme')
    ax.legend()

    plt.tight_layout()
    plt.savefig('spiral_complexity_vs_omega.png', dpi=300,
    bbox_inches='tight')
    plt.show()
    print(" Plot gespeichert:
    spiral_complexity_vs_omega.png")

    # Debug-Ausgaben
    print(f"\nMin Rotation: {np.min(curl):.4f}, Max
    Rotation: {np.max(curl):.4f}")
    print(f"Mean Rotation: {np.mean(curl):.4f}")
    print(f"Max C: {np.max(complexities):.4f} bei
    ω={omega_range[np.argmax(complexities)]:.3f}")
    print(f"Min C: {np.min(complexities):.4f} bei
    ω={omega_range[np.argmin(complexities)]:.3f}")
```

Listing A.1: Visualisierung Spirale: Orbits und Komplexität

A.2 Spirale und Fixpunkte, (Abschnitt. 4.4.2)

```python
# spirale_fixpunkte.py
import numpy as np
import matplotlib.pyplot as plt

# Funktion f(x) definieren
def f(x):
    return x * (1 + 1 / (1 + x**2))

# x-Werte und Funktionswerte berechnen
x = np.linspace(0.01, 10, 1000)
y = f(x)

# Plot der Funktion und der Fixpunktbedingung y = x
plt.figure(figsize=(8, 6))
plt.plot(x, y, 'b-', linewidth=2, label=r'$f(x) = x \cdot (1 + \frac{1}{1 + x^2})$')
plt.plot(x, x, 'k--', linewidth=1, label=r'$y = x$ (Fixpunktbedingung)')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$f(x)$', fontsize=14)
plt.title('Keine Schnittpunkte → Keine endlichen Fixpunkte', fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)
plt.xlim(0, 10)
plt.ylim(0, 12)
plt.savefig('fixpunkte_analyse.png', dpi=300, bbox_inches='tight')
plt.show()

# ======================================================
# DEBUG: Analyse der "schrägen Geraden" im Curl-Feld
# ======================================================

print("\n" + "="*60)
print("DEBUG: Analyse der Struktur im Curl-Feld (rechter Plot)")
print("="*60)

# ============================
# KÜNSTLICHES CURL-FELD ERZEUGEN (Beispiel!)
# ============================
```

```
# Angenommen: curl ist ein 2D-Feld, abhängig von x und theta
x_vals = np.linspace(0.1, 10, 200)      # Beispiel: x-Bereich
theta_vals = np.linspace(0, 2*np.pi, 300)  # Beispiel: theta-Bereich

# Erzeuge ein künstliches Curl-Feld mit "schrägen Linien" (z .B. sinusförmige Modulation)
X, Theta = np.meshgrid(x_vals, theta_vals, indexing='ij')
# Beispiel-Curl: simuliert zwei schräge Extremalbänder
curl = np.sin(Theta - 0.5 * X) * np.exp(-0.1 * X)  # Beispiel-Feld mit schrägen Strukturen

# Finde die Indizes der globalen Extrema
max_curl_idx = np.unravel_index(np.argmax(curl), curl.shape)
min_curl_idx = np.unravel_index(np.argmin(curl), curl.shape)

x_max, theta_max = x_vals[max_curl_idx[0]], theta_vals[max_curl_idx[1]]
x_min, theta_min = x_vals[min_curl_idx[0]], theta_vals[min_curl_idx[1]]

print(f"Global Maximum der Rotation: bei x={x_max:.3f}, θ={theta_max:.3f} rad → curl={np.max(curl):.4f}")
print(f"Global Minimum der Rotation: bei x={x_min:.3f}, θ={theta_min:.3f} rad → curl={np.min(curl):.4f}")

# ======================================================
# Extrahiere "Linien" entlang der sichtbaren Strukturen
# MIT SPITZENDETEKTION (robuster!)
# ======================================================

from scipy.signal import find_peaks

print("\nAnalyse: Verlauf der dominanten Strukturen mit Peak-Finding")

min_peak_distance = 5  # in Indizes – bei dünnem Gitter reduzieren

upper_line_x = []
upper_line_theta = []
lower_line_x = []
lower_line_theta = []

```

```
for i in range(curl.shape[0]):
    row = curl[i, :]

    # Finde Peaks
    peaks_upper, _ = find_peaks(row, height=None,
    distance=min_peak_distance)
    peaks_lower, _ = find_peaks(-row, height=None,
    distance=min_peak_distance)

    # Wähle stärksten Peak pro Kategorie
    if len(peaks_upper) > 0:
        strongest_upper =
    peaks_upper[np.argmax(row[peaks_upper])]
        upper_line_x.append(x_vals[i])
        upper_line_theta.append(theta_vals[strongest_upper])

    if len(peaks_lower) > 0:
        strongest_lower =
    peaks_lower[np.argmax(-row[peaks_lower])]
        lower_line_x.append(x_vals[i])
        lower_line_theta.append(theta_vals[strongest_lower])

# Konvertiere in Arrays
upper_line_x = np.array(upper_line_x)
upper_line_theta = np.array(upper_line_theta)
lower_line_x = np.array(lower_line_x)
lower_line_theta = np.array(lower_line_theta)

print(f"Obere Linie: {len(upper_line_x)} Punkte | Untere
    Linie: {len(lower_line_x)} Punkte")

# Nachdem du upper_line_x, upper_line_theta, etc. hast:

X, Theta = np.meshgrid(x_vals, theta_vals, indexing='ij')

plt.figure(figsize=(12, 6))
plt.contourf(X, Theta, curl, levels=100, cmap='RdBu_r',
    extend='both')
plt.colorbar(label='Curl')

# Plotte die extrahierten Linien
plt.plot(upper_line_x, upper_line_theta, 'w-', linewidth=2,
    label='Obere Peak-Linie')
```

```
plt.plot(lower_line_x, lower_line_theta, 'w--', linewidth=2,
    label='Untere Peak-Linie')

plt.xlabel('$x$', fontsize=14)
plt.ylabel(r'$\theta$ (rad)', fontsize=14)
plt.title('Curl-Feld mit extrahierten Peak-Strukturen',
    fontsize=14)
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('curl_field_with_peak_lines.png', dpi=300,
    bbox_inches='tight')
plt.show()

# ======================================================
# Berechne Steigungen (lineare Regression) für beide Linien
# ======================================================

if len(upper_line_x) > 2:
    # Lineare Regression: theta = m*x + b
    A_upper = np.vstack([upper_line_x,
    np.ones(len(upper_line_x))]).T
    m_upper, c_upper = np.linalg.lstsq(A_upper,
    upper_line_theta, rcond=None)[0]
    print(f"\nObere „"Gerade: θ = {m_upper:.4f} * x +
    {c_upper:.4f}")
    print(f"  Länge: {len(upper_line_x)} Punkte, x-Bereich:
    [{np.min(upper_line_x):.2f}, {np.max(upper_line_x):.2f}]")

if len(lower_line_x) > 2:
    A_lower = np.vstack([lower_line_x,
    np.ones(len(lower_line_x))]).T
    m_lower, c_lower = np.linalg.lstsq(A_lower,
    lower_line_theta, rcond=None)[0]
    print(f"Untere „"Gerade: θ = {m_lower:.4f} * x +
    {c_lower:.4f}")
    print(f"  Länge: {len(lower_line_x)} Punkte, x-Bereich:
    [{np.min(lower_line_x):.2f}, {np.max(lower_line_x):.2f}]")

# ======================================================
# Parallelität prüfen: Differenz der Steigungen
# ======================================================

if 'm_upper' in locals() and 'm_lower' in locals():
```

```
    delta_m = np.abs(m_upper - m_lower)
    print(f"\nSteigungsdifferenz Δm = |m_upper - m_lower| = {delta_m:.6f}")
    if delta_m < 0.01:
        print("  → Die Linien sind nahezu parallel Δ(m < 0.01).")
    elif delta_m < 0.05:
        print("  → Die Linien sind leicht divergierend (0.01 ≤ Δm < 0.05).")
    else:
        print("  → Die Linien sind deutlich nicht parallel Δ(m ≥ 0.05).")

# ======================================================
# Optional: Speichere die Linienkoordinaten für externe Analyse
# ======================================================

np.savetxt('debug_upper_line.csv',
    np.column_stack([upper_line_x, upper_line_theta]),
           delimiter=',', header='x,theta', comments='')
np.savetxt('debug_lower_line.csv',
    np.column_stack([lower_line_x, lower_line_theta]),
           delimiter=',', header='x,theta', comments='')
print(f"\n Debug-Daten gespeichert: debug_upper_line.csv und debug_lower_line.csv")

plt.figure(figsize=(10, 6))
plt.contourf(X, Theta, curl, levels=50, cmap='RdBu_r')
plt.colorbar(label='Curl')
plt.plot(upper_line_x, upper_line_theta, 'w-', linewidth=2, label='Obere Linie')
plt.plot(lower_line_x, lower_line_theta, 'w--', linewidth=2, label='Untere Linie')
plt.xlabel('x')
plt.ylabel(r'$\theta$')
plt.title('Curl-Feld mit extrahierten Strukturen')
plt.legend()
plt.savefig('curl_field_with_lines.png', dpi=300, bbox_inches='tight')
plt.show()
plt.close("all")
```

Listing A.2: Visualisierung Spirale und Fixpunkte

A.3 3D-Vortex-Simulation mit Lyapunov-Exponent, (Abschn. 6.3.1)

```
# ============================
# 3D Vortex Simulation – MIT LIVE LYAPUNOV-EXPONENT im GIF
# ============================
# vortex_3d_gif_simulation_lyapunov.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation, PillowWriter
import os
import time

# ==============================
# KONFIGURATION
# ==============================

N = 2000
x0 = 1.0
theta0 = 0.1
z0 = 0.5

f_mode = 'default'
g_mode = 'wave'

g_params = {
    'amplitude': 0.3,
    'k_theta': 3.0,
    'k_x': 0.7,
    'offset': 1.0,
    'decay_rate': 0.1
}

# Lyapunov-Einstellungen
lyapunov_enabled = True
delta0 = 1e-8           # Anfangsstörung
renorm_interval = 10    # alle 10 Schritte Abstand
    renormieren
dt = 1.0                # Zeitschritt (diskret → dt=1)

save_plots = True
save_animation = True
```

```
save_data = True
output_dir = './results_3d'
dpi_plot = 300
animation_fps = 15
animation_dpi = 100

# ================================
# HILFSFUNKTIONEN
# ================================

def ensure_output_dir():
    os.makedirs(output_dir, exist_ok=True)

# --- f-Funktionen ---
def f_default(x):
    return x * (1 + 1 / (1 + x**2))

def f_linear(x):
    return x * 1.01

def f_chaotic(x):
    return 3.9 * x * (1 - x / 10)

def select_f_function():
    funcs = {'default': f_default, 'linear': f_linear,
    'chaotic': f_chaotic}
    return funcs.get(f_mode, f_default)

# --- g-Funktionen ---
def g_wave(x, theta, params):
    A = params['amplitude']; θk = params['k_theta']; kx =
    params['k_x']; c = params['offset']
    return c + A * np.sin(θk * theta - kx * x)

def g_compress(x, theta, params):
    return np.exp(-params['decay_rate'] * x)

def g_resonant(x, theta, params):
    return 1.0 + params['amplitude'] * np.cos(2 * theta) *
    np.exp(-x / 5)

def g_custom(x, theta, params):
    return 1.0 + 0.1 * np.sin(theta) * np.cos(x)

```

```
def select_g_function():
    funcs = {'wave': g_wave, 'compress': g_compress,
    'resonant': g_resonant, 'custom': g_custom}
    return funcs.get(g_mode, g_wave)

# ==================================
# SIMULATION + LYAPUNOV
# ==================================

print("🚀 Starte 3D-Wirbel-Simulation mit
    Lyapunov-Analyse...")
start_time = time.time()

# Haupttrajektorie
x = np.zeros(N); theta = np.zeros(N); z = np.zeros(N)
x[0] = x0; theta[0] = theta0; z[0] = z0

# Schatten-Trajektorie (für Lyapunov)
x2 = np.zeros(N); theta2 = np.zeros(N); z2 = np.zeros(N)
x2[0] = x0 + delta0; theta2[0] = theta0; z2[0] = z0  # nur x
    gestört

f_func = select_f_function()
g_func = select_g_function()

# Arrays für Lyapunov
lyap_sum = 0.0
lyap_local = np.zeros(N)
lyap_cumulative = np.zeros(N)

for n in range(N-1):
    # Haupttrajektorie
    x[n+1] = f_func(x[n])
    theta[n+1] = theta[n] + 0.3 * x[n]
    z[n+1] = z[n] * g_func(x[n], theta[n], g_params)

    # Schatten-Trajektorie
    x2[n+1] = f_func(x2[n])
    theta2[n+1] = theta2[n] + 0.3 * x2[n]
    z2[n+1] = z2[n] * g_func(x2[n], theta2[n], g_params)

    # Sicherheitsabfang
    for arr in [z, z2]:
        if np.abs(arr[n+1]) > 1e6:
```

```
            arr[n+1] = np.sign(arr[n+1]) * 1e6

    # Lyapunov-Berechnung
    if lyapunov_enabled:
        # Zustandsvektoren
        v1 = np.array([x[n+1], theta[n+1], z[n+1]])
        v2 = np.array([x2[n+1], theta2[n+1], z2[n+1]])
        d = np.linalg.norm(v2 - v1)

        if d == 0:
            d = 1e-16

        # Lokaler Wachstumsfaktor
        lyap_local[n+1] = np.log(d / delta0) / dt

        # Renormierung (nur alle renorm_interval Schritte)
        if (n+1) % renorm_interval == 0 and n > 0:
            # Setze Schatten zurück auf Haupttrajektorie + normierter Richtungsvektor
            direction = v2 - v1
            direction = direction / np.linalg.norm(direction) * delta0
            x2[n+1] = x[n+1] + direction[0]
            theta2[n+1] = theta[n+1] + direction[1]
            z2[n+1] = z[n+1] + direction[2]
            lyap_sum += np.log(d / delta0)
            lyap_cumulative[n+1] = lyap_sum / ((n+1) * dt)
        else:
            lyap_cumulative[n+1] = lyap_cumulative[n]  # übernehme alten Wert

print("□ Simulation + Lyapunov abgeschlossen.")

# Umwandlung in kartesische Koordinaten
X = x * np.cos(theta); Y = x * np.sin(theta); Z = z
X2 = x2 * np.cos(theta2); Y2 = x2 * np.sin(theta2); Z2 = z2

# ================================
# STANDARDBILD: 3D-PLOT
# ================================

ensure_output_dir()

fig = plt.figure(figsize=(14, 6))
```

```
ax1 = fig.add_subplot(121, projection='3d')
ax1.scatter(X, Y, Z, c=np.linspace(0,1,len(X)),
    cmap='plasma', s=5, alpha=0.8)
ax1.set_title('3D Trajektorie')
ax1.set_xlabel('$X$'); ax1.set_ylabel('$Y$');
    ax1.set_zlabel('$Z$')

# Lyapunov-Plot
ax2 = fig.add_subplot(122)
ax2.plot(range(N), lyap_cumulative, 'k-', linewidth=1.5)
ax2.axhline(0, color='r', linestyle='--', alpha=0.7)
ax2.set_xlabel('Iteration n')
ax2.set_ylabel('Lyapunov-Exponent λ')
ax2.set_title('Kumulativer maximaler Lyapunov-Exponent')
ax2.grid(True, alpha=0.3)

fig.suptitle('Self-Modulating Spiral Map: 3D Wirbelstruktur
    + Lyapunov-Analyse', fontsize=14)
#fig.text(0.5, 0.01, 'Visualisierung: Klaus H. Dieckmann,
    2025', ha='center', fontsize=13, style='italic')

if save_plots:
    plt.savefig(f"{output_dir}/3d_trajectory_lyapunov.png",
    dpi=dpi_plot, bbox_inches='tight')
    print("□ Statischer Plot gespeichert.")

plt.show()

# ==================================
# ANIMATION MIT LYAPUNOV-LIVE-ANZEIGE
# ==================================

if save_animation:
    print("□ Erzeuge GIF-Animation mit Lyapunov-Anzeige...")

    fig_anim = plt.figure(figsize=(13, 11))
    ax_anim = fig_anim.add_subplot(111, projection='3d')
    ax_anim.set_xlim(X.min(), X.max())
    ax_anim.set_ylim(Y.min(), Y.max())
    ax_anim.set_zlim(Z.min(), Z.max())
    ax_anim.set_xlabel('$X$'); ax_anim.set_ylabel('$Y$');
    ax_anim.set_zlabel('$Z$')

    # Titel & Untertitel
```

```
    fig_anim.suptitle('Self-Modulating Spiral Map: 3D Wirbel-Animation', fontsize=16, fontweight='bold', y=0.96)
    fig_anim.text(0.5, 0.89, 'Visualisierung: Klaus H. Dieckmann, 2025', ha='center', fontsize=14, style='italic')

    # Erklärender Text (Phasen)
    phase_text = fig_anim.text(0.5, 0.06, "", ha='center', fontsize=14, wrap=True,

    bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow", alpha=0.3))

    # Lyapunov-Anzeige (rechts unten)
    lyap_text = fig_anim.text(0.98, 0.02, "", ha='right', fontsize=13,

    bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue", alpha=0.7))

    line, = ax_anim.plot([], [], [], 'b-', lw=1.5, alpha=0.8, label='Haupttrajektorie')
    point, = ax_anim.plot([], [], [], 'ro', markersize=6)
    line2, = ax_anim.plot([], [], [], 'g--', lw=1, alpha=0.6, label='Schatten-Trajektorie')  # optional
    # Legende mit Schriftgröße 12
    ax_anim.legend(loc='upper right', fontsize=12)

    def get_phase_description(progress_pct):
        if progress_pct < 10:
            return "Phase 1: Initialisierung – System startet mit radialer Expansion und leichter Winkelakkumulation."
        elif progress_pct < 30:
            return "Phase 2: Spiralbildung – Wachstum von x und θ führt zur Ausbildung einer flachen Spirale in der X-Y-Ebene."
        elif progress_pct < 60:
            return "Phase 3: Vertikale Modulation – Die g-Funktion induziert oszillierende oder wachsende Bewegung in Z-Richtung."
        elif progress_pct < 90:
            return "Phase 4: Stabilisierung – Struktur erreicht dynamisches Gleichgewicht; Spirale öffnet sich
```

```
    linear, Höhe oszilliert."
227         else:
228             return "Phase 5: Sättigung – Numerische Grenzen
    oder Dämpfung stabilisieren die Trajektorie; keine
    weitere Expansion."
229
230     def animate(frame):
231         step = max(1, N // 200)
232         idx = min(frame * step, N-1)
233         progress_pct = (idx / N) * 100
234
235         # Update Trajektorien
236         line.set_data(X[:idx+1], Y[:idx+1]);
    line.set_3d_properties(Z[:idx+1])
237         point.set_data([X[idx]], [Y[idx]]);
    point.set_3d_properties([Z[idx]])
238         line2.set_data(X2[:idx+1], Y2[:idx+1]);
    line2.set_3d_properties(Z2[:idx+1])
239
240         # Update Texte
241
    phase_text.set_text(get_phase_description(progress_pct))
242         current_lyap = lyap_cumulative[idx] if idx > 0 else
    0.0
243         lyap_status = "CHAOS" if current_lyap > 0.01 else
    "STABIL" if current_lyap < -0.01 else "NEUTRAL"
244         lyap_text.set_text(f"Lyapunov λ =
    {current_lyap:.4f}\nStatus: {lyap_status}")
245
246         return line, point, line2, phase_text, lyap_text
247
248     frames = min(200, N // max(1, N // 200))
249     ani = FuncAnimation(fig_anim, animate, frames=frames,
    interval=50, blit=False)
250
251     writer = PillowWriter(fps=animation_fps)
252     gif_path = f"{output_dir}/3d_trajectory_lyapunov.gif"
253     ani.save(gif_path, writer=writer, dpi=animation_dpi)
254     print(f" GIF mit Lyapunov-Anzeige gespeichert:
    {gif_path}")
255
256 # ================================
257 # DATENEXPORT
258 # ================================
```

```
259
260 if save_data:
261     print("□ Exportiere erweiterte Daten (inkl.
    Lyapunov)...")
262
263     np.savetxt(f"{output_dir}/trajectory_cartesian.csv",
    np.column_stack([X, Y, Z]), delimiter=',',
    header='X,Y,Z', comments='')
264     np.savetxt(f"{output_dir}/trajectory_polar.csv",
    np.column_stack([x, theta, z]), delimiter=',',
    header='x,theta,z', comments='')
265     np.savetxt(f"{output_dir}/lyapunov_cumulative.csv",
    lyap_cumulative, delimiter=',',
    header='lambda_cumulative', comments='')
266
267     np.savez_compressed(f"{output_dir}/
    trajectory_full_with_lyapunov.npz",
268                         x=x, theta=theta, z=z, X=X, Y=Y, Z=Z,
269                         x2=x2, theta2=theta2, z2=z2, X2=X2,
    Y2=Y2, Z2=Z2,
270                         lyap_local=lyap_local,
    lyap_cumulative=lyap_cumulative,
271                         config={
272                             'N': N, 'x0': x0, 'theta0':
    theta0, 'z0': z0,
273                             'f_mode': f_mode, 'g_mode':
    g_mode,
274                             'g_params': g_params,
275                             'delta0': delta0,
    'renorm_interval': renorm_interval
276                         })
277
278     print("□ Erweiterte Daten exportiert.")
279
280 # ====================================
281 # ZUSAMMENFASSUNG
282 # ====================================
283
284 end_time = time.time()
285 final_lyap = lyap_cumulative[-1]
286 chaos_status = "CHAOTISCH" if final_lyap > 0.01 else
    "STABIL" if final_lyap < -0.01 else
    "GRENZZYKLUS/QUASIPERIODISCH"
287
```

```
print(f"  Gesamtlaufzeit: {end_time - start_time:.2f} Sekunden")
print(f" Finaler Lyapunov-Exponent: {final_lyap:.5f} → System ist {chaos_status}")
print(f" Ergebnisse im Ordner: {os.path.abspath(output_dir)}")
```

Listing A.3: Visualisierung 3D-Vortex-Simulation mit Lyapunov-Exponent

A.4 Ergodizität, (Kap. 7)

```
# spirale_ergotisch.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import kstest
import pandas as pd
import os

# Definition der Self-Modulating Spiral Map
def self_mod_map(x, theta, omega=0.05):
    f_x = x * (1 + 1 / (1 + x**2))
    x_new = f_x
    theta_new = (theta + omega * np.log(1 + f_x)) % (2 * np.pi)
    return x_new, theta_new

# Simulation einer langen Trajektorie
def simulate_long_orbit(x0, theta0, steps=100000, omega=0.05):
    orbit = np.zeros((steps + 1, 2))
    orbit[0] = [x0, theta0]
    for i in range(steps):
        orbit[i+1] = self_mod_map(orbit[i, 0], orbit[i, 1], omega)
    return orbit

# Autokorrelationsfunktion (ACF) berechnen
def compute_acf(theta_vals, max_lag=10):
    series = pd.Series(theta_vals)
    acf = [series.autocorr(lag=i) for i in range(1, max_lag + 1)]
    return np.array(acf)
```

```
28
29 # Analyse für einen omega-Wert durchführen und Plots
       erstellen
30 def analyze_omega(omega, x0=1.0, theta0=0.0, steps=100000,
       save_dir='.'):
31     orbit = simulate_long_orbit(x0, theta0, steps, omega)
32     x_vals = orbit[:, 0]
33     theta_vals = orbit[:, 1]
34
35     # KS-Test auf Uniformität
36     theta_norm = theta_vals / (2 * np.pi)
37     ks_stat, p_value = kstest(theta_norm, 'uniform')
38
39     # Mittlere Dichte aus Histogramm
40     hist, _ = np.histogram(theta_vals, bins=100,
       density=True)
41     mean_density = np.mean(hist)
42
43     # ACF berechnen
44     acf = compute_acf(theta_vals)
45
46     # Plots erstellen
47     fig, axs = plt.subplots(1, 3, figsize=(18, 5))
48     fig.suptitle(f'Analyse für ω = {omega:.4f}')
49
50     # Plot 1: Phasenraum log(x) vs θ
51     axs[0].scatter(theta_vals, np.log(x_vals + 1), s=1,
       alpha=0.5)
52     axs[0].set_title('Phasenraum: log(x) vs θ')
53     axs[0].set_xlabel('θ')
54     axs[0].set_ylabel('log(x + 1)')
55     axs[0].grid(True)
56
57     # Plot 2: Histogramm von θ
58     axs[1].hist(theta_vals, bins=100, density=True,
       alpha=0.7, color='blue')
59     axs[1].axhline(1/(2*np.pi), color='red', linestyle='--',
       label='Uniforme Dichte')
60     axs[1].set_title('Histogramm von θ')
61     axs[1].set_xlabel('θ')
62     axs[1].set_ylabel('Dichte')
63     axs[1].set_xlim([0, 2*np.pi])
64     axs[1].legend()
65     axs[1].grid(True)
```

```
66
67      # Plot 3: Autokorrelationsfunktion
68      lags = np.arange(1, len(acf) + 1)
69      axs[2].stem(lags, acf, basefmt=" ")
70      axs[2].set_title('Autokorrelationsfunktion (ACF) von θ')
71      axs[2].set_xlabel('Lag')
72      axs[2].set_ylabel('ACF')
73      axs[2].set_ylim([-1, 1])
74      axs[2].grid(True)
75
76      plt.tight_layout()
77      plot_filename =
        f'{save_dir}/analysis_omega_{omega:.4f}.png'
78      plt.savefig(plot_filename)
79      plt.close()
80
81      return {
82          'omega': omega,
83          'ks_stat': ks_stat,
84          'p_value': p_value,
85          'mean_density': mean_density,
86          'acf': acf,
87          'plot_file': plot_filename
88      }
89
90  # Hauptprogramm
91  if __name__ == "__main__":
92      # Liste extremer ω-Werte (rationale, irrationale,
        kleine/große)
93      omega_list = [0.001, 0.01, 0.05, 0.1, np.pi/10, 1/3,
        0.5, 2/3, np.sqrt(2)/2, 1.0, np.e/2, 2.0, 10.0]
94      results = []
95
96      save_dir = 'ergodicity_analysis'
97      os.makedirs(save_dir, exist_ok=True)
98
99      for omega in omega_list:
100         print(f'Bearbeite ω = {omega:.4f}...')
101         result = analyze_omega(omega, save_dir=save_dir)
102         results.append(result)
103
104     # Ergebnisse in Tabelle zusammenfassen
105     df = pd.DataFrame(results)
```

```
    df['type'] = ['irrational' if isinstance(w, float) and
    not w.is_integer() and w != round(w, 5) else 'rational'
    for w in df['omega']]
    df = df[['omega', 'type', 'ks_stat', 'p_value',
    'mean_density', 'acf', 'plot_file']]
    print('\nZusammenfassung der Ergebnisse:')
    print(df[['omega', 'type', 'ks_stat', 'p_value',
    'mean_density']])

    # ACF in separater Tabelle (nur erste 10 Lags)
    acf_table = pd.DataFrame([r['acf'] for r in results],
    index=omega_list)
    acf_table.columns = [f'Lag {i+1}' for i in
    range(acf_table.shape[1])]
    print('\nACF-Werte (abgerundet):')
    print(acf_table.round(4))

    # Speichere Tabellen
    df.to_csv(f'{save_dir}/summary_results.csv', index=False)
    acf_table.to_csv(f'{save_dir}/acf_table.csv')
    print(f'\nErgebnisse gespeichert in {save_dir}')
```

Listing A.4: Visualisierung Ergodizität

A.5 Stochastische 3D-Spirale (Animation), (Abschn. 8.1.2)

```
# spirale_3d_stochastisch_gif_animation.py
# Self-Modulating Spiral Map in 3D with Stochastic Noise
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation, PillowWriter
import os
import time

# ================================
# KONFIGURATION
# ================================

N = 3000  # Anzahl der Iterationen
x0 = 1.0
```

```
theta0 = 0.0
z0 = 0.5

# Steuerparameter
omega = 0.05  # Winkelmodulationsstärke (wie in Ihrer Arbeit)
vertical_amplitude = 0.4  # Stärke der vertikalen Modulation
k_theta = 2.0  # Wellenzahl für vertikale Modulation

# Rauschparameter (Abschnitt 11.2)
noise_enabled = True
sigma_x = 0.02  # Standardabweichung für multiplikatives Rauschen auf x
sigma_theta = 0.05  # Standardabweichung für additives Rauschen auf theta (in rad)

# Lyapunov-Einstellungen (optional, für Chaos-Analyse)
lyapunov_enabled = True
delta0 = 1e-8  # Anfangsstörung für Schatten-Trajektorie
renorm_interval = 20  # Alle 20 Schritte Abstand renormieren
dt = 1.0  # Zeitschritt (diskret → dt=1)

# Ausgabe-Einstellungen
save_plots = True
save_animation = True  # Setze auf False, wenn GIF zu lange dauert
save_data = True
output_dir = './results_3d_stochastic'
dpi_plot = 300
animation_fps = 20
animation_dpi = 100

# ================================
# HILFSFUNKTIONEN
# ================================

def ensure_output_dir():
    os.makedirs(output_dir, exist_ok=True)

def f(x):
    """Radiale Modulationsfunktion (wie in Ihrer Arbeit, Gl. 5.1)"""
    return x * (1 + 1 / (1 + x**2))

def g_vertical(x, theta, amplitude=0.4, k_theta=2.0):
```

```
    """Vertikale Modulation: oszilliert mit Winkel und Radius (Abschnitt 7.6)"""
    return 1.0 + amplitude * np.sin(k_theta * theta) * np.exp(-x / 10)

# ================================
# SIMULATION MIT STOCHASTIK UND LYAPUNOV
# ================================

print("🚀 Starte 3D-Wirbel-Simulation mit stochastischer Störung und Lyapunov-Analyse...")
start_time = time.time()

# Haupttrajektorie
x = np.zeros(N)
theta = np.zeros(N)
z = np.zeros(N)
x[0] = x0
theta[0] = theta0
z[0] = z0

# Schatten-Trajektorie (für Lyapunov)
if lyapunov_enabled:
    x2 = np.zeros(N)
    theta2 = np.zeros(N)
    z2 = np.zeros(N)
    x2[0] = x0 + delta0  # nur x gestört
    theta2[0] = theta0
    z2[0] = z0

# Arrays für Lyapunov
if lyapunov_enabled:
    lyap_sum = 0.0
    lyap_local = np.zeros(N)
    lyap_cumulative = np.zeros(N)

# Hauptiteration
for n in range(N - 1):
    # --- Deterministischer Teil ---
    x_new = f(x[n])
    theta_new = theta[n] + omega * np.log(1 + x_new)
    z_new = z[n] * g_vertical(x[n], theta[n], vertical_amplitude, k_theta)

```

```
    # --- Stochastische Störung hinzufügen (Abschnitt 11.2) ---
    if noise_enabled:
        # Multiplikatives Rauschen auf x (z.B. turbulente Fluktuation)
        noise_x = np.random.normal(0, sigma_x)
        x_new *= (1 + noise_x)

        # Additives Rauschen auf theta (z.B. zufällige Drehimpulsstöße)
        noise_theta = np.random.normal(0, sigma_theta)
        theta_new += noise_theta

        # Sicherstellen, dass x positiv bleibt
        if x_new <= 0:
            x_new = 1e-6

    # --- Winkel modulo π2 bringen ---
    theta_new = theta_new % (2 * np.pi)

    # --- Zustand aktualisieren ---
    x[n+1] = x_new
    theta[n+1] = theta_new
    z[n+1] = z_new

    # --- Schatten-Trajektorie (für Lyapunov) ---
    if lyapunov_enabled:
        x2_new = f(x2[n])
        theta2_new = theta2[n] + omega * np.log(1 + x2_new)
        z2_new = z2[n] * g_vertical(x2[n], theta2[n], vertical_amplitude, k_theta)

        if noise_enabled:
            # Rauschen auch auf Schatten-Trajektorie anwenden (für fairen Vergleich)
            noise_x2 = np.random.normal(0, sigma_x)
            x2_new *= (1 + noise_x2)
            noise_theta2 = np.random.normal(0, sigma_theta)
            theta2_new += noise_theta2
            if x2_new <= 0:
                x2_new = 1e-6

        theta2_new = theta2_new % (2 * np.pi)

```

```
        x2[n+1] = x2_new
        theta2[n+1] = theta2_new
        z2[n+1] = z2_new

        # Sicherheitsabfang für extreme Werte
        for arr in [z, z2]:
            if np.abs(arr[n+1]) > 1e6:
                arr[n+1] = np.sign(arr[n+1]) * 1e6

        # --- Lyapunov-Berechnung ---
        v1 = np.array([x[n+1], theta[n+1], z[n+1]])
        v2 = np.array([x2[n+1], theta2[n+1], z2[n+1]])
        d = np.linalg.norm(v2 - v1)

        if d == 0:
            d = 1e-16

        lyap_local[n+1] = np.log(d / delta0) / dt

        if (n+1) % renorm_interval == 0 and n > 0:
            # Renormierung: Schatten zurück auf Haupttrajektorie + normierter Richtungsvektor
            direction = v2 - v1
            direction_norm = np.linalg.norm(direction)
            if direction_norm > 0:
                direction = direction / direction_norm * delta0
            else:
                direction = np.array([delta0, 0, 0])  # Fallback

            x2[n+1] = x[n+1] + direction[0]
            theta2[n+1] = theta[n+1] + direction[1]
            z2[n+1] = z[n+1] + direction[2]

            lyap_sum += np.log(d / delta0)
            lyap_cumulative[n+1] = lyap_sum / ((n+1) * dt)
        else:
            lyap_cumulative[n+1] = lyap_cumulative[n]  # übernehme alten Wert

print("□ Simulation abgeschlossen.")

# Umwandlung in kartesische Koordinaten
```

```
X = x * np.cos(theta)
Y = x * np.sin(theta)
Z = z

if lyapunov_enabled:
    X2 = x2 * np.cos(theta2)
    Y2 = x2 * np.sin(theta2)
    Z2 = z2

# ================================
# STANDARDBILD: 3D-PLOT + LYAPUNOV
# ================================

ensure_output_dir()

fig = plt.figure(figsize=(14, 6))

# 3D Trajektorie
ax1 = fig.add_subplot(121, projection='3d')
scatter = ax1.scatter(X, Y, Z, c=np.arange(N),
    cmap='plasma', s=5, alpha=0.8)
ax1.set_title('3D Trajektorie mit stochastischer Störung')
ax1.set_xlabel('$X$')
ax1.set_ylabel('$Y$')
ax1.set_zlabel('$Z$')
fig.colorbar(scatter, ax=ax1, shrink=0.5, label='Iteration')

# Lyapunov-Plot
if lyapunov_enabled:
    ax2 = fig.add_subplot(122)
    ax2.plot(range(N), lyap_cumulative, 'k-', linewidth=1.5)
    ax2.axhline(0, color='r', linestyle='--', alpha=0.7,
    label='Neutral λ(=0)')
    ax2.set_xlabel('Iteration n')
    ax2.set_ylabel('Lyapunov-Exponent λ')
    ax2.set_title('Kumulativer maximaler Lyapunov-Exponent')
    ax2.grid(True, alpha=0.3)
    ax2.legend()
else:
    ax2 = fig.add_subplot(122)
    ax2.text(0.5, 0.5, 'Lyapunov deaktiviert', ha='center',
    va='center', fontsize=14)
    ax2.axis('off')

```

```
fig.suptitle('Self-Modulating Spiral Map: 3D mit Stochastik', fontsize=14)

if save_plots:

    plt.savefig(f"{output_dir}/3d_trajectory_stochastic.png", dpi=dpi_plot, bbox_inches='tight')
    print("□ Statischer Plot gespeichert.")

plt.show()

# ================================
# ANIMATION (optional)
# ================================

if save_animation:
    print("□ Erzeuge GIF-Animation...")

    fig_anim = plt.figure(figsize=(13, 9))
    ax_anim = fig_anim.add_subplot(111, projection='3d')
    ax_anim.set_xlim(X.min(), X.max())
    ax_anim.set_ylim(Y.min(), Y.max())
    ax_anim.set_zlim(Z.min(), Z.max())
    ax_anim.set_xlabel('$X$')
    ax_anim.set_ylabel('$Y$')
    ax_anim.set_zlabel('$Z$')

    fig_anim.suptitle('SMSM 3D mit Stochastik – Animation', fontsize=16, fontweight='bold')
    fig_anim.text(0.5, 0.92, 'Visualisierung: Klaus H. Dieckmann, 2025', ha='center', fontsize=12, style='italic')

    # Textfelder
    noise_text = fig_anim.text(0.02, 0.02, f"Rauschen: σ_x={sigma_x}, σθ_={sigma_theta}", fontsize=12, bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgreen", alpha=0.7))
    lyap_text = fig_anim.text(0.98, 0.02, "", ha='right', fontsize=12, bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue", alpha=0.7))

    line, = ax_anim.plot([], [], [], 'b-', lw=1.5, alpha=0.8, label='Haupttrajektorie')
```

```
    point, = ax_anim.plot([], [], [], 'ro', markersize=6)

    def animate(frame):
        step = max(1, N // 200)
        idx = min(frame * step, N - 1)

        line.set_data(X[:idx+1], Y[:idx+1])
        line.set_3d_properties(Z[:idx+1])
        point.set_data([X[idx]], [Y[idx]])
        point.set_3d_properties([Z[idx]])

        if lyapunov_enabled and idx > 0:
            current_lyap = lyap_cumulative[idx]
            lyap_status = "CHAOS" if current_lyap > 0.01
    else "STABIL" if current_lyap < -0.01 else "NEUTRAL"
            lyap_text.set_text(f"Lyapunov λ =
    {current_lyap:.4f}\nStatus: {lyap_status}")

        return line, point, lyap_text

    frames = min(200, N // max(1, N // 200))
    ani = FuncAnimation(fig_anim, animate, frames=frames,
    interval=50, blit=False)

    writer = PillowWriter(fps=animation_fps)
    gif_path = f"{output_dir}/3d_trajectory_stochastic.gif"
    ani.save(gif_path, writer=writer, dpi=animation_dpi)
    print(f" GIF gespeichert: {gif_path}")

# ================================
# DATENEXPORT
# ================================

if save_data:
    print(" Exportiere Daten...")
    np.savetxt(f"{output_dir}/trajectory_cartesian.csv",
    np.column_stack([X, Y, Z]), delimiter=',',
    header='X,Y,Z', comments='')
    np.savetxt(f"{output_dir}/trajectory_polar.csv",
    np.column_stack([x, theta, z]), delimiter=',',
    header='x,theta,z', comments='')
    if lyapunov_enabled:
        np.savetxt(f"{output_dir}/lyapunov_cumulative.csv",
    lyap_cumulative, delimiter=',',
```

```
    header='lambda_cumulative', comments='')

    config = {
        'N': N,
        'x0': x0,
        'theta0': theta0,
        'z0': z0,
        'omega': omega,
        'vertical_amplitude': vertical_amplitude,
        'k_theta': k_theta,
        'noise_enabled': noise_enabled,
        'sigma_x': sigma_x,
        'sigma_theta': sigma_theta,
        'lyapunov_enabled': lyapunov_enabled,
        'delta0': delta0,
        'renorm_interval': renorm_interval
    }

    # Speichere Konfiguration als Text
    with open(f"{output_dir}/simulation_config.txt", 'w') as f:
        for key, value in config.items():
            f.write(f"{key}: {value}\n")

    print("✅ Daten exportiert.")

# ================================
# ZUSAMMENFASSUNG
# ================================

end_time = time.time()
print(f"⏱️ Gesamtlaufzeit: {end_time - start_time:.2f} Sekunden")
print(f"📂 Ergebnisse im Ordner: {os.path.abspath(output_dir)}")

if lyapunov_enabled:
    final_lyap = lyap_cumulative[-1]
    chaos_status = "CHAOTISCH" if final_lyap > 0.01 else "STABIL" if final_lyap < -0.01 else "GRENZZYKLUS/QUASIPERIODISCH"
    print(f"🌀 Finaler Lyapunov-Exponent: {final_lyap:.5f} → System ist {chaos_status}")
```

Listing A.5: Visualisierung Stochastische 3D-Spirale (Animation)

A.6 Komplexe Spirale, (Abschnitt. 9.2.1)

```python
# komplexe_spirale_map.py
# Self-Modulating Spiral Map in the Complex Plane
import numpy as np
import matplotlib.pyplot as plt
import cmath

def f(x):
    """
    Radial modulation function.
    x: real, positive scalar (radius)
    Returns: new radius
    """
    return x * (1 + 1 / (1 + x**2))

def complex_spiral_map(z, omega=0.05):
    """
    Iterates the Self-Modulating Spiral Map in the complex
    plane.

    z: complex number (current state)
    omega: rotational control parameter

    Returns: z_next (complex number)
    """
    x_n = abs(z)  # current radius
    if x_n == 0:
        return 0 + 0j  # avoid division by zero

    f_x = f(x_n)  # new radius

    # Calculate the complex multiplier
    # M = (f_x / x_n) * exp(i * omega * ln(1 + f_x))
    phase_factor = cmath.exp(1j * omega * np.log(1 + f_x))
    scaling_factor = f_x / x_n

    multiplier = scaling_factor * phase_factor

```

```
    z_next = z * multiplier
    return z_next

def simulate_complex_orbit(z0, steps=1000, omega=0.05):
    """
    Simulates an orbit in the complex plane.

    z0: initial complex state
    steps: number of iterations
    omega: control parameter

    Returns: array of complex numbers (trajectory)
    """
    orbit = np.zeros(steps + 1, dtype=complex)
    orbit[0] = z0

    for i in range(steps):
        orbit[i+1] = complex_spiral_map(orbit[i], omega)

    return orbit

def spiral_complexity_complex(orbit):
    """
    Computes the Spiral Complexity for a complex orbit.
    C = (1/(N-1)) * sum_{i=1}^{N-1} Δθ|_i| * ln(|z_i|)

    orbit: array of complex numbers
    Returns: scalar complexity value
    """
    N = len(orbit)
    if N < 2:
        return 0.0

    # Extract radii and angles
    radii = np.abs(orbit)
    angles = np.angle(orbit)  # in radians, in range [-pi,
   pi]

    # Compute angular differences (unwrap to avoid 2pi jumps)
    angles_unwrapped = np.unwrap(angles)
    dtheta = np.diff(angles_unwrapped)  # Δθ_i = θ_{i+1} -
   θ_i
    abs_dtheta = np.abs(dtheta)  # Δθ|_i|

```

```
    # Use radius at step i (not i+1) for weighting, as in original definition
    complexity = np.mean(abs_dtheta * np.log(radii[1:]))
    return complexity

def plot_complex_orbit(orbit, omega, title_suffix=""):
    """
    Plots the complex orbit in the complex plane.
    """
    real_part = np.real(orbit)
    imag_part = np.imag(orbit)

    plt.figure(figsize=(10, 8))
    plt.plot(real_part, imag_part, 'b-', linewidth=1.5, alpha=0.8, label='Trajectory')
    plt.scatter(real_part[0], imag_part[0], c='red', s=50, label='Start', zorder=5)
    plt.scatter(real_part[-1], imag_part[-1], c='green', s=50, label='End', zorder=5)

    plt.xlabel('Re(z)')
    plt.ylabel('Im(z)')
    plt.title(f'Complex Self-Modulating Spiral Map\ωn = {omega}' + title_suffix)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.axis('equal')  # Equal scaling for x and y axes
    plt.savefig('komplexe_spirale_orbit.png', dpi=300)
    plt.show()

# --- Main Execution ---
if __name__ == "__main__":
    # Parameters
    z0 = 1.0 + 0.0j  # Start at (x=1, θ=0)
    omega = 0.05
    steps = 2000

    print("Simulating Complex Self-Modulating Spiral Map...")

    # Simulate orbit
    orbit = simulate_complex_orbit(z0, steps=steps, omega=omega)

    # Calculate Spiral Complexity
```

```
    C = spiral_complexity_complex(orbit)
    print(f"Spiral Complexity C = {C:.6f}")

    # Plot the orbit
    plot_complex_orbit(orbit, omega, f"\nSpiral Complexity C = {C:.6f}")

    # --- Parameter Study: Complexity vs. Omega ---
    print("\nPerforming parameter study: Complexity vs. Omega...")
    omega_range = np.linspace(0.01, 0.5, 50)
    complexities = []

    for w in omega_range:
        orbit_w = simulate_complex_orbit(z0, steps=1000, omega=w)
        C_w = spiral_complexity_complex(orbit_w)
        complexities.append(C_w)
        if w in [0.01, 0.1, 0.3, 0.5]:
            print(f" ω={w:.2f} → C={C_w:.6f}")

    complexities = np.array(complexities)

    # Plot Complexity vs. Omega
    plt.figure(figsize=(8, 5))
    plt.plot(omega_range, complexities, 'o-', color='darkblue', linewidth=2, markersize=4)
    plt.xlabel('ω (Steuerparameter)', fontsize=12)
    plt.ylabel('Spiral-Komplexität C', fontsize=12)
    plt.title('Spiral-Komplexität als Funktion von ω (Komplexe Ebene)', fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.axvline(x=0.1, color='gray', linestyle='--', alpha=0.7, label='Beginn schneller Zunahme')
    plt.legend()
    plt.tight_layout()
    plt.savefig('komplexe_spirale_complexity_vs_omega.png', dpi=300)
    plt.show()

    plt.plot(orbit.real, orbit.imag)
    plt.xlabel('Realteil')
    plt.ylabel('Imaginärteil')
    plt.title('Trajektorie in der komplexen Ebene')
```

```
    plt.axis('equal')
    plt.savefig('komplexe_spirale_trajectory.png', dpi=300)
    plt.show()
    plt.close("all")

    print(f"Max C: {np.max(complexities):.4f} bei
    ω={omega_range[np.argmax(complexities)]:.3f}")
    print(f"Min C: {np.min(complexities):.4f} bei
    ω={omega_range[np.argmin(complexities)]:.3f}")

    print("\nSimulation completed.")
```

Listing A.6: Visualisierung Komplexe Spirale

A.7 Heterogene Kopplung der Spiralen, (Abschnitt. 10.3.1)

```
# spirale_kopplung_heterogen.py
# Kopplung mehrerer selbstmodulierender Spiral Maps (SMSM)
   mit heterogenen Frequenzen
# → Ermöglicht echten Synchronisationsübergang!

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.patches import Circle
import cmath

# === Modulare Funktionen ===

def f(x):
    """Radiale Modulationsfunktion."""
    return x * (1 + 1 / (1 + x**2))

def complex_spiral_map(z, omega):
    """Einzelne SMSM-Iteration – NUR Phasendynamik, Radius
   wird auf 1 normiert."""
    x_n = abs(z)
    if x_n == 0:
        return 1 + 0j  # Sicherheitsstartwert

    f_x = f(x_n)
```

```
    phase_factor = cmath.exp(1j * omega * np.log(1 + f_x))
    # Normiere Eingangszustand, wende nur Phasenrotation an
    return (z / x_n) * phase_factor

def coupling_diffusive(z_i, z_left, z_right, K):
    """Diffusive Kopplung im Ring: z_i <- z_i + K * ((z_left - z_i) + (z_right - z_i))"""
    return K * (z_left + z_right - 2*z_i)

def order_parameter(Z):
    """Komplexer Ordnungsparameter für Synchronisation: R = |<exp(i * theta_i)>|"""
    angles = np.angle(Z)
    return np.abs(np.mean(np.exp(1j * angles)))

# === Simulation der gekoppelten SMSM-Einheiten ===

def simulate_coupled_smsm(N_units=10, steps=500, omega_mean=0.1, omega_spread=0.08, K=0.02):
    """
    Simuliert ein Netzwerk gekoppelter SMSM-Einheiten in einem Ring.

    N_units: Anzahl der Einheiten (Oszillatoren)
    steps: Anzahl der Iterationen
    omega_mean: mittlere interne Rotationsfrequenz
    omega_spread: Streuung der Frequenzen (uniform um mean)
    K: Kopplungsstärke
    """
    # Initialisierung: Einheitskreis, gleichmäßig verteilte Phasen
    phases_init = np.linspace(0, 2*np.pi, N_units, endpoint=False)
    Z = np.zeros((steps + 1, N_units), dtype=complex)
    Z[0] = np.exp(1j * phases_init)  # Start auf Einheitskreis, verschiedene Phasen

    # Heterogene omega_i
    omega_i = omega_mean + omega_spread * (np.random.rand(N_units) - 0.5)
    print(f"[Info] Heterogene Frequenzen: ω_i ∈ [{np.min(omega_i):.4f}, {np.max(omega_i):.4f}]")

```

```
    R = np.zeros(steps + 1)  # Ordnungsparameter pro Zeitschritt
    R[0] = order_parameter(Z[0])

    for t in range(steps):
        Z_next = np.zeros(N_units, dtype=complex)
        for i in range(N_units):
            # Nachbarn im Ring (periodisch)
            left = Z[t, (i-1) % N_units]
            right = Z[t, (i+1) % N_units]
            z_i = Z[t, i]

            # Eigenentwicklung mit eigenem omega_i + Kopplung
            z_intrinsic = complex_spiral_map(z_i, omega_i[i])
            z_coupling = coupling_diffusive(z_i, left, right, K)
            Z_next[i] = z_intrinsic + z_coupling

        Z[t+1] = Z_next
        R[t+1] = order_parameter(Z_next)

    return Z, R, omega_i

# === Visualisierungen ===

def plot_trajectories(Z, K, title_suffix=""):
    """Plot aller Trajektorien in der komplexen Ebene."""
    plt.figure(figsize=(10, 8))
    for i in range(Z.shape[1]):
        plt.plot(np.real(Z[:, i]), np.imag(Z[:, i]), linewidth=1, alpha=0.7, label=f'Unit {i+1}' if i < 5 else "")
    plt.scatter(np.real(Z[0, :]), np.imag(Z[0, :]), c='red', s=30, label='Start', zorder=5)
    plt.scatter(np.real(Z[-1, :]), np.imag(Z[-1, :]), c='green', s=30, label='Ende', zorder=5)
    plt.xlabel('Re(z)')
    plt.ylabel('Im(z)')
    plt.title(f'Gekoppelte SMSM mit heterogenen ω_i (N={Z.shape[1]}, K={K})\n' + title_suffix)
    plt.legend(loc='upper right', fontsize=8)
    plt.grid(True, alpha=0.3)
    plt.axis('equal')
```

```
94      plt.savefig('coupled_smsm_heterogen_trajectories.png',
        dpi=300, bbox_inches='tight')
95      plt.show()
96
97  def plot_order_parameter(R, K):
98      """Plot des Ordnungsparameters über die Zeit."""
99      plt.figure(figsize=(10, 4))
100     plt.plot(R, color='darkblue', linewidth=2)
101     plt.xlabel('Zeitschritt')
102     plt.ylabel('Ordnungsparameter R')
103     plt.title(f'Synchronisation: R(t) für heterogene ω_i,
        K={K}')
104     plt.grid(True, alpha=0.3)
105     plt.ylim(-0.05, 1.05)
106     plt.axhline(y=1.0, color='gray', linestyle='--',
        label='Vollständige Synchronisation')
107     plt.axhline(y=0.0, color='lightgray', linestyle=':',
        label='Vollständige Desynchronisation')
108     plt.legend()
109
        plt.savefig('coupled_smsm_heterogen_order_parameter.png',
        dpi=300, bbox_inches='tight')
110     plt.show()
111
112 def plot_angle_evolution(Z, N_show=5):
113     """Plot der Winkelentwicklung der ersten N_show
        Einheiten."""
114     plt.figure(figsize=(12, 5))
115     for i in range(min(N_show, Z.shape[1])):
116         plt.plot(np.unwrap(np.angle(Z[:, i])), label=f'Unit
        {i+1}', linewidth=1.5)
117     plt.xlabel('Zeitschritt')
118     plt.ylabel('Winkel θ (rad, unwrapped)')
119     plt.title('Winkelentwicklung einzelner Einheiten – zeigt
        Synchronisationsprozess')
120     plt.legend()
121     plt.grid(True, alpha=0.3)
122     plt.savefig('coupled_smsm_heterogen_angles.png',
        dpi=300, bbox_inches='tight')
123     plt.show()
124
125 # === Parameterstudie: Synchronisation vs. Kopplungsstärke K
        ===
126
```

```
127 def parameter_study_sync(N_units=10, steps=500,
    omega_mean=0.1, omega_spread=0.08, K_range=np.linspace(0,
    0.1, 20)):
128     """Untersucht Synchronisation als Funktion der
    Kopplungsstärke K mit heterogenen ω_i."""
129     R_final = []
130     for K in K_range:
131         print(f"Simuliere mit K = {K:.4f}...")
132         Z, R, _ = simulate_coupled_smsm(N_units, steps,
    omega_mean, omega_spread, K)
133         R_final.append(R[-1])  # Endwert des
    Ordnungsparameters
134 
135     plt.figure(figsize=(8, 5))
136     plt.plot(K_range, R_final, 'o-', color='darkred',
    linewidth=2, markersize=5)
137     plt.xlabel('Kopplungsstärke K')
138     plt.ylabel('End-Ordnungsparameter R (→∞t)')
139     plt.title(f'Synchronisationsübergang mit heterogenen
    ω_i\n(N={N_units}, ω_mean={omega_mean},
    spread={omega_spread})')
140     plt.grid(True, alpha=0.3)
141     plt.axhline(y=1.0, color='gray', linestyle='--',
    alpha=0.7, label='R=1 (voll synchron)')
142     plt.axhline(y=0.0, color='lightgray', linestyle=':',
    alpha=0.7, label='R=0 (inkohärent)')
143     plt.legend()
144     plt.savefig('coupled_smsm_heterogen_sync_vs_K.png',
    dpi=300, bbox_inches='tight')
145     plt.show()
146 
147     return K_range, np.array(R_final)
148 
149 # === Hauptprogramm ===
150 
151 if __name__ == "__main__":
152     print("=== Simulation gekoppelter SMSM mit heterogenen
    Frequenzen ===")
153 
154     # Parameter
155     N_units = 8           # Anzahl der Einheiten
156     steps = 400           # Simulationsschritte
157     omega_mean = 0.1      # mittlere interne Frequenz
158     omega_spread = 0.08   # Streuung ±0.04 um Mittelwert
```

```
    K_test = 0.04          # Test-Kopplung für Einzelsimulation

    # Einzelsimulation mit mittlerer Kopplung
    print(f"\nFühre Einzelsimulation durch mit K = {K_test}...")
    Z, R, omega_i = simulate_coupled_smsm(N_units, steps, omega_mean, omega_spread, K_test)

    # Plots für Einzelsimulation
    plot_trajectories(Z, K_test, f"R_final = {R[-1]:.4f}")
    plot_order_parameter(R, K_test)
    plot_angle_evolution(Z, N_show=5)

    # Parameterstudie: Synchronisation vs. K
    print(f"\nFühre Parameterstudie durch: Synchronisation vs. Kopplungsstärke K...")
    K_vals, R_vals = parameter_study_sync(
        N_units=N_units,
        steps=steps,
        omega_mean=omega_mean,
        omega_spread=omega_spread,
        K_range=np.linspace(0, 0.08, 17)
    )

    # Finale Ausgabe
    print(f"\nMaximale Synchronisation R_max = {np.max(R_vals):.4f} bei K = {K_vals[np.argmax(R_vals)]:.4f}")
    print(f"Minimale Synchronisation R_min = {np.min(R_vals):.4f} bei K = {K_vals[np.argmin(R_vals)]:.4f}")
    print("\n Simulation abgeschlossen. Echter Synchronisationsübergang sichtbar gemacht.")
```

Listing A.7: Visualisierung Heterogene Kopplung der Spiralen

A.8 SMSM mit gekoppelten Oszilatoren (Animation), (Abschnitt. 10.3.1)

```
# spirale_kopplung_gif_animation.py
import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.patches import Circle

# Parameter
N = 20                      # Anzahl Oszillatoren
T = 120                     # Anzahl Zeitschritte
dt = 0.1                    # Zeitschritt
K = np.linspace(0.0, 2.5, T)  # Kopplungsstärke steigt im Laufe der Zeit
omega = np.random.uniform(0.8, 1.2, N)  # Natürliche Frequenzen
theta = np.random.uniform(0, 2*np.pi, N)  # Zufällige Anfangsphasen

# Vorbereitung Plot mit optimierter Größe
fig, ax = plt.subplots(figsize=(11, 11))
ax.set_xlim(-1.8, 1.8)
ax.set_ylim(-2.0, 1.8)
ax.set_aspect('equal')
ax.axis('off')

# Titel
title = ax.text(0.0, 2.0, "Self-Modulating Spiral Map (SMSM):\nGekoppelte Oszillatoren",
                fontsize=16, ha='center', va='top', fontweight='bold')

# Urheberhinweis - höher positioniert
author = ax.text(0.0, 1.7, "Visualisierung: Klaus H. Dieckmann, 2025",
                 fontsize=14, ha='center', va='top', color='gray')

# Kleinerer, zentrierter Kreis
circle = Circle((0, 0), 1.0, color='lightgray', fill=False, lw=1.5, alpha=0.6)
ax.add_artist(circle)

# Initialisiere Zeiger
lines = []
dots = []
for i in range(N):
```

```
38      line, = ax.plot([0, np.cos(theta[i])*1.0], [0,
        np.sin(theta[i])*1.0],
39                      lw=2.0, color=plt.cm.hsv(i/N), zorder=2)
40      dot, = ax.plot(np.cos(theta[i])*1.0,
        np.sin(theta[i])*1.0, 'o',
41                     color=plt.cm.hsv(i/N), markersize=10,
        zorder=3)
42      lines.append(line)
43      dots.append(dot)
44
45  # Text für Kopplungsstärke (kleiner und kompakter)
46  text_K = ax.text(0, 1.4, '', fontsize=12, ha='center',
        va='top',
47                   color='darkblue', weight='bold',
48                   bbox=dict(boxstyle="round,pad=0.4",
        facecolor="lightblue",
49                             edgecolor="darkblue", alpha=0.7))
50
51  # Breiterer Erklärungstext mit mehr Platz an den Seiten
52  explanation_text = ax.text(0, -1.2, '', fontsize=13,
        ha='center', va='top',
53
        bbox=dict(boxstyle="round,pad=0.8",
        facecolor="lightyellow",
54                                       edgecolor="orange",
        alpha=0.9),
55                             linespacing=1.5,
56                             wrap=True)  # Textumbruch
        aktivieren
57
58  # Breitere Textbox erzwingen durch manuelle Begrenzung
59  explanation_text._get_wrap_line_width = lambda: 500  #
        Breite der Textbox erhöhen
60
61  # Phasen-Erklärungen
62  phase_explanations = [
63      "Anfangsphase (K < 0.5): Jeder Oszillator schwingt
        unabhängig. Kaum Synchronisation erkennbar.",
64      "Erste Anpassung (K ≈ 0.5-1.0): Oszillatoren beginnen
        sich an benachbarte Phasen anzupassen.",
65      "Clusterbildung (K ≈ 1.0-1.3): Gruppen synchronisieren
        sich lokal und bilden Cluster.",
66      "Kritischer Punkt (K ≈ 1.3-1.6): Kopplung löst globale
        Synchronisation aus.",
```

```
    "Teilweise Synchronisation (K ≈ 1.6-2.0): Großteil der Oszillatoren schwingt im Gleichtakt.",
    "Volle Synchronisation (K > 2.0): Fast alle Oszillatoren sind perfekt synchronisiert."
]

# Update-Funktion für Animation
def animate(t):
    global theta
    k = K[t]  # Kopplungsstärke steigt mit der Zeit

    # Kuramoto-Gleichung
    dtheta = np.zeros_like(theta)
    for i in range(N):
        dtheta[i] = omega[i] + (k / N) * np.sum(np.sin(theta - theta[i]))

    theta += dtheta * dt
    theta = np.mod(theta, 2*np.pi)

    # Update der Zeiger und Punkte
    for i in range(N):
        x, y = np.cos(theta[i])*1.0, np.sin(theta[i])*1.0
        lines[i].set_data([0, x], [0, y])
        dots[i].set_data([x], [y])

    # Update Kopplungsstärke-Text (kürzer und prägnanter)
    text_K.set_text(f'$Kopplungsstärke = {k:.2f}$')

    # Dynamische Erklärung basierend auf Kopplungsstärke
    if k < 0.5:
        explanation = phase_explanations[0]
    elif k < 1.0:
        explanation = phase_explanations[1]
    elif k < 1.3:
        explanation = phase_explanations[2]
    elif k < 1.6:
        explanation = phase_explanations[3]
    elif k < 2.0:
        explanation = phase_explanations[4]
    else:
        explanation = phase_explanations[5]

    # Berechne Synchronisationsparameter r
```

```
    r = np.abs(np.sum(np.exp(1j * theta)) / N)

    # Füge Synchronisationsgrad zum Text hinzu
    sync_text = f'\nSynchronisation: r = {r:.2f}'
    if r > 0.8:
        sync_text += ' (stark)'
    elif r > 0.5:
        sync_text += ' (mittel)'
    else:
        sync_text += ' (schwach)'

    explanation_text.set_text(explanation + sync_text)

    return lines + dots + [text_K, explanation_text]

# Animation
ani = animation.FuncAnimation(fig, animate, frames=T,
    interval=80, blit=True)
ani.save('spirale_gekoppelte_oszillatoren.gif',
    writer='pillow', fps=20, dpi=100)

plt.tight_layout(pad=3.0)  # Mehr Padding für bessere
    Textdarstellung
plt.subplots_adjust(top=0.85, bottom=0.22)  # Anpassung der
    Ränder
plt.show()
```

Listing A.8: Visualisierung SMSM mit gekoppelten Oszilatoren (Animation)

A.9 Inverse Spirale (Animation), (Abschnitt. 12.2)

```
# spirale_invers_gif_animation.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D

# Parameter
N = 300          # Anzahl Punkte in der Spirale
T = 120          # Anzahl Frames
max_radius = 3.0 # Startet oben (breite Basis)
min_radius = 0.1 # Endet unten (schmale Spitze)
```

```
omega = 1.5      # Winkelgeschwindigkeit
height = 5.0     # Höhe des Tornados

# Vorbereitung 3D-Plot
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim(-3.5, 3.5)
ax.set_ylim(-3.5, 3.5)
ax.set_zlim(0, height)
ax.set_box_aspect([1, 1, 1])  # Gleichmäßige Skalierung

# Weißer Hintergrund
fig.patch.set_facecolor('white')
ax.set_facecolor('white')
ax.xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
ax.yaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
ax.zaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))

# Gitter und Achsen ausblenden
ax.grid(False)
ax.set_axis_off()

# Titel hinzufügen
title = fig.suptitle("Self-Modulation Spiral Map (SMSM): Inverse Spirale",
                     fontsize=16, fontweight='bold', y=0.92, color='darkblue')

# Urhebervermerk
author = fig.text(0.5, 0.85, "Visualisierung: Klaus H. Dieckmann, 2025",
                  fontsize=14, ha='center', va='top', color='darkgreen')

# Farben für die verschiedenen Phasen
phase_colors = ["#F04444", "#2C6B66", "#2C6B79", "#7D672B"]
phase_names = [
    "Initiationsphase: Breite Basis am oberen Ende bildet sich",
    "Kontraktionsphase: Spirale verjüngt sich nach unten",
    "Stabilisierungsphase: Gleichmäßige Drehbewegung setzt ein",
    "Dissipationsphase: Auflösung an der schmalen Spitze beginnt"
```

```
]

# Erklärungstext
explanation_text = fig.text(0.5, 0.05, phase_names[0],
    fontsize=13, ha='center', va='bottom',
                            color=phase_colors[0],

    bbox=dict(boxstyle="round,pad=0.5",
    facecolor="lightyellow", alpha=0.8))

# Initialisiere Spirale (leer)
spiral, = ax.plot([], [], [], 'o', markersize=3,
    color='#606060', alpha=0.7)

# Update-Funktion
def animate(t):
    # Radius schrumpft mit der Zeit (oben groß, unten klein)
    current_radius = max_radius * (1 - t / T) + min_radius *
    (t / T)

    # Winkel: mit negativem Vorzeichen → "Tornado-Drehung"
    (im Uhrzeigersinn)
    theta = np.linspace(0, 6*np.pi, N) - omega * t * 0.1

    # Radius entlang der Spirale modulieren
    r = np.linspace(current_radius, min_radius, N)

    # Höhe entlang der Spirale (oben = 0, unten = height -
    invertiert für Tornado-Optik)
    z = np.linspace(height, 0, N)  # Umgekehrte Reihenfolge
    für "auf dem Kopf stehenden" Tornado

    # Umrechnung in x, y, z
    x = r * np.cos(theta)
    y = r * np.sin(theta)

    # Update Plot
    spiral.set_data(x, y)
    spiral.set_3d_properties(z)

    # Dynamischen Erklärungstext basierend auf dem aktuellen
    Frame anzeigen
    phase = min(3, t // (T // 4))  # Bestimme die aktuelle
    Phase (0-3)
```

```
83
84      # Text mit Farbe aktualisieren
85      explanation_text.set_text(f"Phase {phase+1}/4:
        {phase_names[phase]}")
86      explanation_text.set_color(phase_colors[phase])
87
88      return [spiral, explanation_text]
89
90  # Animation ohne Blitting, da es mit 3D Probleme verursacht
91  ani = animation.FuncAnimation(fig, animate, frames=T,
        interval=60, blit=False, repeat=True)
92
93  plt.tight_layout(rect=[0, 0.1, 1, 0.9])  # Platz für Titel
        und Erklärung lassen
94  # Speichern der Animation als GIF mit 15 Sekunden Dauer (8
        FPS)
95  ani.save('spirale_invers_animation.gif', writer='pillow',
        fps=8, dpi=100)
96
97  plt.show()
```

Listing A.9: Visualisierung Inverse Spirale (Animation)

A.10 Bifurkation der Spirale, (Abschnitt. 4.6.2)

```
1  # spirale_bifurkation.py
2  # Bifurkationsanalyse der Self-Modulating Spiral Map (SMSM)
3  # mit Poincaré-Schnitt bei theta = 0 (mod 2pi)
4  # BUNTE Version: Farbe zeigt die Punktdichte (Histogramm2D)
5
6  import numpy as np
7  import matplotlib.pyplot as plt
8  from matplotlib.colors import LogNorm
9  import matplotlib.cm as cm
10
11 def self_mod_map(x, theta, omega):
12     """
13     Iteriert die Self-Modulating Spiral Map (SMSM).
14     """
15     f_x = x * (1 + 1 / (1 + x**2))
16     x_new = f_x
17     theta_new = (theta + omega * np.log(1 + f_x)) % (2 *
       np.pi)
```

```
    return x_new, theta_new

def compute_poincare_bifurcation_data(omega_min, omega_max, num_omega, transient_steps, max_iterations):
    """
    Berechnet die Daten für das Bifurkationsdiagramm basierend auf Poincaré-Schnitten.
    """
    omega_values = []
    x_values = []  # Wir speichern den Radius x beim Poincaré-Schnitt

    omega_range = np.linspace(omega_min, omega_max, num_omega)

    for omega in omega_range:
        x, theta = 1.0, 0.0

        # Transient-Phase
        for _ in range(transient_steps):
            x, theta = self_mod_map(x, theta, omega)

        # Haupt-Phase: Suche nach Poincaré-Schnitten
        last_theta = theta
        points_recorded = 0

        for step in range(max_iterations):
            x, theta = self_mod_map(x, theta, omega)

            # Poincaré-Schnitt erkennen: Übergang von >pi zu <pi
            if last_theta > np.pi and theta < np.pi:
                omega_values.append(omega)
                x_values.append(x)
                points_recorded += 1

                # Optional: Begrenze die Anzahl der Punkte pro omega
                if points_recorded >= 150:
                    break

            last_theta = theta

```

```
55          print(f"Berechne ω = {omega:.4f}
    ({len(omega_values)} Punkte gesammelt)...", end='\r')
56
57      print("\nBerechnung abgeschlossen.")
58      return np.array(omega_values), np.array(x_values)
59
60  def plot_poincare_bifurcation_diagram_colored(omega_values,
    x_values, bins=300):
61      """
62      Plottet das Bifurkationsdiagramm als 2D-Histogramm
    (Dichtekarte).
63      """
64      plt.figure(figsize=(16, 8))
65
66      # Erstelle ein 2D-Histogramm. Die Farbe zeigt die Anzahl
    der Punkte pro Bin.
67      # 'plasma' ist eine gute Wahl, aber Sie können auch
    'viridis', 'inferno', 'jet' oder 'turbo' probieren.
68      counts, xedges, yedges, im = plt.hist2d(
69          omega_values, x_values,
70          bins=[bins//2, bins],  # Mehr Bins in y-Richtung
    (x-Werte) für schärfere Linien
71          cmap='plasma',         # Weitere Optionen:
    'viridis', 'inferno', 'turbo', 'jet'
72          norm=LogNorm(),        # Logarithmische Skala: macht
    seltene und häufige Ereignisse sichtbar
73          range=[[omega_values.min(), omega_values.max()],
    [x_values.min(), x_values.max()]]
74      )
75
76      plt.colorbar(im, label='Log(Punktdichte)')
77      plt.xlabel('Steuerparameter ω', fontsize=16,
    fontweight='bold')
78      plt.ylabel('Radius x beim Poincaré-Schnitt θ( mod π2 =
    0)', fontsize=16, fontweight='bold')
79      plt.title('Bifurkationsdiagramm der SMSM
    (Poincaré-Schnitt)\n'
80                'Farbe zeigt die Dichte der Zustände –
    Übergang zu Chaos bei ω ≈ 0.2',
81                fontsize=18, fontweight='bold')
82      plt.axvline(x=0.2, color='cyan', linestyle='--',
    linewidth=3, label='Chaos-Übergang ω( ≈ 0.2)')
83      plt.legend(fontsize=14, loc='upper left')
84      plt.grid(True, alpha=0.3, linestyle='--')
```

```
    plt.tight_layout()
    plt.savefig('smsm_bifurkation_poincare_colored.png', dpi=300, bbox_inches='tight')
    plt.show()

    # Optional: Ein zweiter Plot mit einer anderen Farbpalette, z.B. 'turbo' für maximale Buntheit
    plt.figure(figsize=(16, 8))
    counts, xedges, yedges, im = plt.hist2d(
        omega_values, x_values,
        bins=[bins//2, bins],
        cmap='turbo',  # 'turbo' ist wie 'jet', aber ohne harte Übergänge – sehr bunt!
        norm=LogNorm(),
        range=[[omega_values.min(), omega_values.max()], [x_values.min(), x_values.max()]]
    )
    plt.colorbar(im, label='Log(Punktdichte)')
    plt.xlabel('Steuerparameter ω', fontsize=16, fontweight='bold')
    plt.ylabel('Radius x beim Poincaré-Schnitt', fontsize=16, fontweight='bold')
    #plt.title('Bifurkationsdiagramm', fontsize=18, fontweight='bold')
    plt.title('Bifurkationsdiagramm der SMSM (Poincaré-Schnitt)\n'
              'Farbe zeigt die Dichte der Zustände – Übergang zu Chaos bei ω ≈ 0.2',
              fontsize=18, fontweight='bold')
    plt.axvline(x=0.2, color='white', linestyle='--', linewidth=3, label='ω ≈ 0.2')
    plt.legend(fontsize=14, loc='upper left')
    plt.grid(True, alpha=0.3, linestyle='--')
    plt.tight_layout()
    plt.savefig('smsm_bifurkation_poincare_turbo.png', dpi=300, bbox_inches='tight')
    plt.show()
    plt.close('all')

# Hauptprogramm
if __name__ == "__main__":
    print("Starte BUNTE Bifurkationsanalyse der SMSM mit Poincaré-Schnitt...")
```

```
    OMEGA_MIN = 0.05
    OMEGA_MAX = 0.5
    NUM_OMEGA = 400
    TRANSIENT_STEPS = 3000
    MAX_ITERATIONS = 50000

    omega_vals, x_vals = compute_poincare_bifurcation_data(
        OMEGA_MIN, OMEGA_MAX, NUM_OMEGA, TRANSIENT_STEPS,
    MAX_ITERATIONS
    )

    # Erstelle das bunte Dichtediagramm
    plot_poincare_bifurcation_diagram_colored(omega_vals,
    x_vals, bins=400)

    print("Bifurkationsdiagramme gespeichert als
    'smsm_bifurkation_poincare_colored.png' und
    'smsm_bifurkation_poincare_turbo.png'.")
```

Listing A.10: Visualisierung Bifurkation der Spirale

A.11 Bifurkationsanalyse (Animation), (Abschnitt. 4.6.2)

```
# bifurkation_smsm_poincare_gif_animated.py
# Animiertes Bifurkationsdiagramm der SMSM mit
   Poincaré-Schnitt und dynamischem Erklärtext

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm, Normalize
from matplotlib.animation import FuncAnimation, PillowWriter
import os

def self_mod_map(x, theta, omega):
    f_x = x * (1 + 1 / (1 + x**2))
    x_new = f_x
    theta_new = (theta + omega * np.log(1 + f_x)) % (2 *
    np.pi)
    return x_new, theta_new

```

```
def compute_poincare_data_for_omega(omega, transient_steps, max_points):
    x, theta = 1.0, 0.0
    omega_vals, x_vals = [], []

    # Transient-Phase
    for _ in range(transient_steps):
        x, theta = self_mod_map(x, theta, omega)

    # Sammle Punkte beim Poincaré-Schnitt
    last_theta = theta
    points_collected = 0

    iteration = 0
    while points_collected < max_points and iteration < 100000:
        x, theta = self_mod_map(x, theta, omega)
        if last_theta > np.pi and theta < np.pi:
            omega_vals.append(omega)
            x_vals.append(x)
            points_collected += 1
        last_theta = theta
        iteration += 1

    return np.array(omega_vals), np.array(x_vals)

def create_animation():
    # Konfiguration
    OMEGA_MIN = 0.05
    OMEGA_MAX = 0.5
    NUM_FRAMES = 100
    TRANSIENT_STEPS = 3000
    POINTS_PER_OMEGA = 80

    omega_values = np.linspace(OMEGA_MIN, OMEGA_MAX, NUM_FRAMES)
    all_omega_data = []
    all_x_data = []

    print("Berechne Daten für Animation...")
    for omega in omega_values:
        w_vals, x_vals = compute_poincare_data_for_omega(omega, TRANSIENT_STEPS, POINTS_PER_OMEGA)
```

```
55          all_omega_data.append(w_vals)
56          all_x_data.append(x_vals)
57          print(f"Abgeschlossen: ω = {omega:.4f}")
58
59      # Erstelle die Figur für die Animation
60      fig, ax = plt.subplots(figsize=(16, 9))  # 16:9 Format
    für bessere Präsentation
61
62      # Setze feste Achsenlimits
63      ax.set_xlim(OMEGA_MIN, OMEGA_MAX)
64      global_y_min = min([x.min() for x in all_x_data if
    len(x) > 0])
65      global_y_max = max([x.max() for x in all_x_data if
    len(x) > 0])
66      ax.set_ylim(global_y_min * 0.95, global_y_max * 1.05)
67
68      ax.set_xlabel('Steuerparameter ω', fontsize=16,
    fontweight='bold')
69      ax.set_ylabel('Radius x beim Poincaré-Schnitt θ( mod π2
    = 0)', fontsize=16, fontweight='bold')
70      ax.set_title('Bifurkationsanalyse der Self-Modulating
    Spiral Map (SMSM)', fontsize=16, fontweight='bold')
71      # Hinzufügen des Urheberhinweises direkt darunter
72      fig.text(0.5, 0.75, 'Visualisierung: Klaus H. Dieckmann,
    2025',
73           fontsize=13, ha='center', va='top', style='italic',
    color='gray')
74      ax.grid(True, alpha=0.3, linestyle='--')
75
76      # Initialisiere Scatter-Plot OHNE cmap/norm, um Warning
    zu vermeiden
77      scatter = ax.scatter([], [], s=15, alpha=0.85,
    edgecolors='none')
78      # Farbgebung wird später dynamisch gesetzt
79
80      # Vertikale Linie für aktuellen ω-Wert
81      vline = ax.axvline(x=OMEGA_MIN, color='cyan',
    linestyle='--', linewidth=3)
82
83      # Text für aktuellen ω-Wert
84      omega_text = ax.text(0.02, 0.95, '',
    transform=ax.transAxes, fontsize=14,
85                           verticalalignment='top',
```

```
                        bbox=dict(boxstyle="round,pad=0.3",
    facecolor="cyan", alpha=0.7))

    # Dynamischer Erklärtext (zentraler Fokus!)
    explanation_text = ax.text(0.5, 0.95, '',
    transform=ax.transAxes, fontsize=12,
                               fontweight='bold',
    ha='center', va='top',

    bbox=dict(boxstyle="round,pad=0.5", facecolor="yellow",
    alpha=0.8))

    def get_phase_description(current_omega):
        """Gibt eine textuelle Beschreibung der aktuellen
    Phase basierend auf ω."""
        if current_omega < 0.15:
            return "PHASE 1: GEORDNETE DYNAMIK\nSystem folgt
    einer stabilen, vorhersehbaren Spirale."
        elif current_omega < 0.22:
            return "PHASE 2: KRITISCHER ÜBERGANG\nSystem
    nähert sich dem Bifurkationspunkt – erste Anzeichen von
    Instabilität."
        elif current_omega < 0.35:
            return "PHASE 3: EINTRITT INS CHAOS\nStruktur
    verzweigt sich – deterministisches Chaos setzt ein."
        else:
            return "PHASE 4: TIEFES CHAOS\nSystem zeigt
    komplexe, unvorhersehbare Dynamik – sensitive
    Abhängigkeit von Anfangsbedingungen."

    def animate(frame):
        # Sammle alle Daten bis zum aktuellen Frame
        current_omega = omega_values[frame]
        combined_omega =
    np.concatenate(all_omega_data[:frame+1])
        combined_x = np.concatenate(all_x_data[:frame+1])

        if len(combined_omega) == 0:
            return scatter, vline, omega_text,
    explanation_text

        # Aktualisiere Scatter-Plot
        scatter.set_offsets(np.column_stack([combined_omega,
    combined_x]))
```

```
115
116         # Farbe nach ω-Wert (nicht nach Dichte, da das in
    Animation zu langsam ist)
117         scatter.set_array(combined_omega)
118         scatter.set_cmap('plasma')  # Jetzt sinnvoll, da
    Daten vorhanden sind
119         scatter.set_clim(OMEGA_MIN, OMEGA_MAX)
120
121         # Aktualisiere vertikale Linie und Texte
122         vline.set_xdata([current_omega, current_omega])
123         omega_text.set_text(f'aktueller ω =
    {current_omega:.3f}')
124         explanation_text.set_text(
    get_phase_description(current_omega))
125
126         return scatter, vline, omega_text, explanation_text
127
128     print("Erzeuge Animation...")
129     anim = FuncAnimation(fig, animate, frames=NUM_FRAMES,
    interval=150, blit=False, repeat=False)
130
131     # Speichere Animation
132     os.makedirs('results', exist_ok=True)
133     writer = PillowWriter(fps=10)
134     anim.save('smsm_bifurkation_animated.gif',
    writer=writer, dpi=120)
135     print("✅ Animation erfolgreich gespeichert als
    'smsm_bifurkation_animated.gif'.")
136
137     # Zeige letzten Frame als statisches Bild (optional)
138     plt.show()
139
140 if __name__ == "__main__":
141     create_animation()
```

Listing A.11: Visualisierung Bifurkationsanalyse (Animation)

Hinweis zur Nutzung von KI

Die Ideen und Konzepte dieser Arbeit stammen von mir. Künstliche Intelligenz wurde unterstützend für die Textformulierung und Gleichungsformatierung eingesetzt. Die inhaltliche Verantwortung liegt bei mir. [1]

[1] ORCID: https://orcid.org/0009-0002-6090-3757

Stand: 24. September 2025
TimeStamp: https://freetsa.org/index_de.php

Literatur

[1] Barge, P., & Sommeria, J. (1995). *Did planet formation begin inside persistent gaseous vortices?* Astronomy and Astrophysics, 295, L1–L4.

[2] Chelton, D. B., Schlax, M. G., & Samelson, R. M. (2011). *Global observations of nonlinear mesoscale eddies*. Progress in Oceanography, 91(2), 167–216. https://doi.org/10.1016/j.pocean.2011.01.002

[3] Holton, J. R. und Hakim, G. J. *An Introduction to Dynamic Meteorology*. 5th Edition, Academic Press, 2012. Kapitel 7-9, S. 189-254. ISBN 978-0-12-384866-6.

[4] Lyra, W., & Lin, M.-K. (2013). *Formation and long-term evolution of 3D vortices in protoplanetary discs*. The Astrophysical Journal, 775(1), 17. https://doi.org/10.1088/0004-637X/775/1/17

[5] Ott, E.. Chaos in Dynamical Systems. Cambridge University Press, 2002.

[6] Pikovsky, A., Rosenblum, M., & Kurths, J.. Synchronization: A Universal Concept in Nonlinear Sciences. Cambridge University Press, 2001.

[7] Powell, M. D., Uhlhorn, E. W., & Kepert, J. D. (2009). Estimating Maximum Surface Winds from Hurricane Reconnaissance Measurements. *Weather and Forecasting*, 24(3), 868–883. https://doi.org/10.1175/2008WAF2007087.1

[8] Strogatz, S. H.: *Nonlinear Dynamics and Chaos*. Westview Press, 1994. DOI: 10.1201/9780429492563.

[9] Willoughby, H. E., Rahn, M. E. (2004). Parametric Representation of the Primary Hurricane Vortex. Part I: Observations and Evaluation of the Holland (1980) Model. *Monthly Weather Review*, 132(12), 3033–3048. https://doi.org/10.1175/MWR2831.1