Physik der sozio-ökonomischen Systeme *mit dem Computer*

JOHANN WOLFGANG GOETHE UNIVERSITÄT
23.01.2026

MATTHIAS HANAUSKE

FRANKFURT INSTITUTE FOR ADVANCED STUDIES
JOHANN WOLFGANG GOETHE UNIVERSITÄT
INSTITUT FÜR THEORETISCHE PHYSIK
ARBEITSGRUPPE RELATIVISTISCHE ASTROPHYSIK
D-60438 FRANKFURT AM MAIN
GERMANY

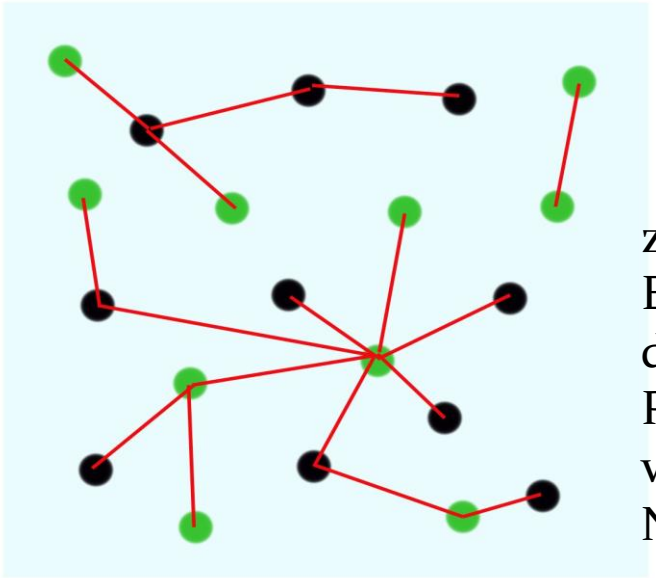
10. Vorlesung

Plan für die heutige Vorlesung

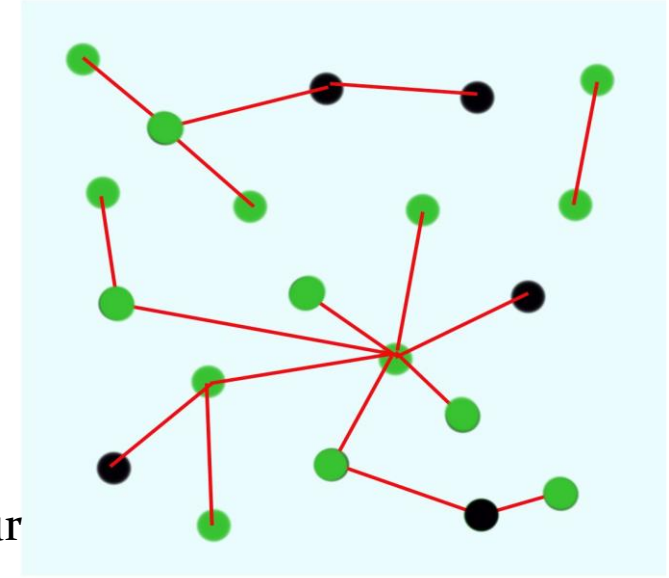
- Evolutionäre Spieltheorie auf komplexen Netzwerken
  - Symmetrische (2 x 2)-Spiele auf einem räumlichen Netzwerk
    - Dominante (2 x 2)-Spiele auf einem räumlichen Gitter
    - Räumliche Koordinations- und Anti-Koordinationsspiele
  - Symmetrische (2 x 3)-Spiele auf einem räumlichen Netzwerk
  - Spiele auf vollständig verbundenen Netzwerken
  - Spiele auf zufälligen, „kleine Welt“ und skalenfreien Netzwerken
- Einführung in die Objekt-orientierte Programmierung

Evolutionäre Spieltheorie auf komplexen Netzwerken

Viele in der Realität vorkommende evolutionäre Spiele werden auf einer definierten Netzwerkstruktur (Topologie) gespielt. Die Spieler der betrachteten Population sind hierbei nicht gleichwertig, sondern wählen als Spielpartner nur mit ihnen durch das Netzwerk verlinkte (verbundene) Partner aus.



zeitliche Entwicklung der Population auf vorgegebener Netzwerkstruktur



x(0)=0.5

x(10)=0.75

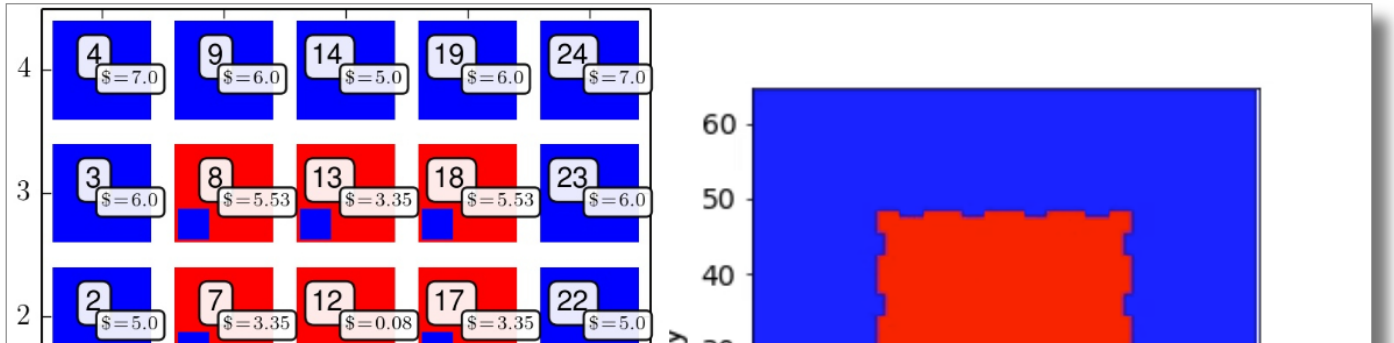
Mögliche Strategien: (grün , schwarz), Parameter t stellt die „Zeit“ dar.
x(t) : Anteil der Spieler, die im Zeitpunkt t die Strategie „grün“ spielen.
Die roten Verbindungslinien beschreiben die möglichen Spielpartner des Spielers

Vorlesung 9

Das in der vorigen Vorlesung betrachtete deterministische SIR-Model und die entsprechenden agenten-basierten Computersimulationen stellten eine Beispielanwendung der Theorie der komplexen Netzwerke dar. In dieser und der darauf folgenden Vorlesung werden wir die deterministische Beschreibung der evolutionäre Spieltheorie (siehe Vorlesungen 3-5) in ähnlicher Weise durch stochastische Computersimulationen darstellen, wobei wir uns in dieser Vorlesung auf symmetrische (2x2)-Spiele auf einem räumlichen Gitter beschränken. Die Verknüpfung der Theorie komplexer Netzwerke mit der evolutionären Spieltheorie stellt ein vielversprechendes mathematisches Modell dar, welches sowohl der interdisziplinären Grundlagenforschung, als auch der angewandten, empirischen Netzwerkforschung dienen kann. In diesem Kapitel wird die Vorgehensweise einer Miteinbeziehung komplexer Netzwerktopologien in die evolutionäre Spieltheorie beschrieben. Die dann auf einem solchen komplexen Netzwerk ablaufenden Entscheidungsprozesse können in den meisten Fällen nur mittels nummerischer, agenten-basierter Computersimulationen veranschaulicht werden. Nach einigen grundlegenden Vorbemerkungen zur *Evolutionären Spieltheorie auf komplexen Netzwerken*, werden wir zunächst die zeitliche Entwicklung von räumlichen dominanten Spielen untersuchen und mit den Lösungen der deterministischen evolutionären Spieltheorie vergleichen. Danach werden Koordinations- und Anti-Koordinationsspiele auf einem räumlichen Gitter simuliert und analysiert.

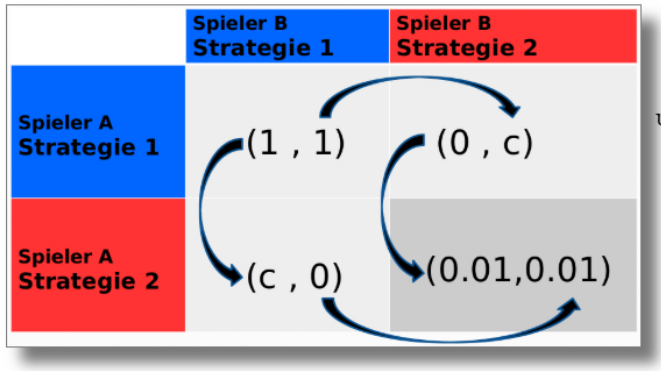
Dominante räumliche Spiele

Wir betrachten zunächst ein räumliches, leicht dominantes Spiel mit der im rechten Panel angegebenen Auszahlungsmatrix und wählen als Anfangskonfiguration der Strategienwahl der Spielerpopulation eine Konstellation, bei der nur ein Spieler die dominante rote Strategie spielt und alle anderen Spieler die blaue Strategienwahl. Die Simulationen zeigen, dass $\forall 1 < c < 1.2$ in der zweiten Spielperiode ein Rechteck aus 9 roten Knoten entsteht, welches aber dann schon in der dritten Spielperiode wieder in einen einzelnen roten Zentrumsknoten übergeht. Die linke untere Abbildung zeigt die räumliche Spielkonstellation zu diesem Zeitpunkt ($c = 1.1$), wobei die eingezeichneten \$-Werte den erzielten kumulierten Auszahlungswerten der Spieler entsprechen und die kleinen Vierecke innerhalb der großen Vierecke die zukünftige Strategienwahl der Spieler in der nächsten Spielperiode angeben.



Vorlesung 9

Die Verknüpfung der Theorie der komplexen Netzwerke mit der evolutionären Spieltheorie wird in dieser Vorlesung an mehreren Beispielen gezeigt. Die Entscheidungsprozesse der Spieler auf einem komplexen Netzwerk können mittels numerischer, agenten-basierter Computerprogramme simuliert werden.
Wir betrachten zunächst ein evolutionäres räumliches Spiel (siehe Spatial Games), wobei die Spieler einer endlich großen Population auf einem räumlichen Gitter angeordnet und jeder Spieler nur mit seinen nächsten Nachbarn spielen kann (*Moore Nachbarschaft*). Das zugrundeliegende Netzwerk der Spielerknoten besitzt somit eine reguläre Struktur und im betrachteten 2-dimensionalen Fall spielt jeder Spieler pro Spielperiode mit acht Spielern (Knotengrad $k_i = 8 \forall i \in \mathcal{I}$).



Wir beschränken uns im Folgenden auf symmetrische 2x2-Spiele und im ersten Unterpunkt dieser Vorlesung (siehe linkes Panel) betrachten wir im Speziellen ein 'leicht' dominantes Spiel mit nebenstehender Auszahlungsmatrix. Es handelt sich bei diesem Spiel um eine Version des Gefangenendilemmas, wobei der Parameter $c > 1$ die Stärke der Dominanz der Strategie $s_2 \hat{=} Rot$ über die Strategie $s_1 \hat{=} Blau$ quantifiziert (siehe *Bestantwort-Pfeile* in der nebenstehenden Auszahlungstabelle). Die Spielerknoten spielen pro Iteration mit jedem ihrer Nachbarn und am Ende von jedem Zeitschritt vergleichen die Spieler ihren summierten Gewinn/Verlust mit den Nachbarspielern ihres Umfeldes (*Update Rule*). Ist die Auszahlung eines Spielers höher als der eigene Auszahlungswert, so ändert der Spieler in der nächsten Spielperiode seine Strategie; ist sein eigener Wert der höchste, so bleibt er auch in der nächsten Iteration bei seiner gespielten Strategie. Die deterministische evolutionäre Spieltheorie sagt für dominante Spiele voraus, dass sich die zeitliche Entwicklung der Population zu einer evolutionär stabilen Strategie entwickelt bei der alle Personen die dominante Strategie spielen (siehe Vorlesung 3 und 4) - bei räumlichen Spielen ist dies nicht zwangsläufig der Fall

Python Programm

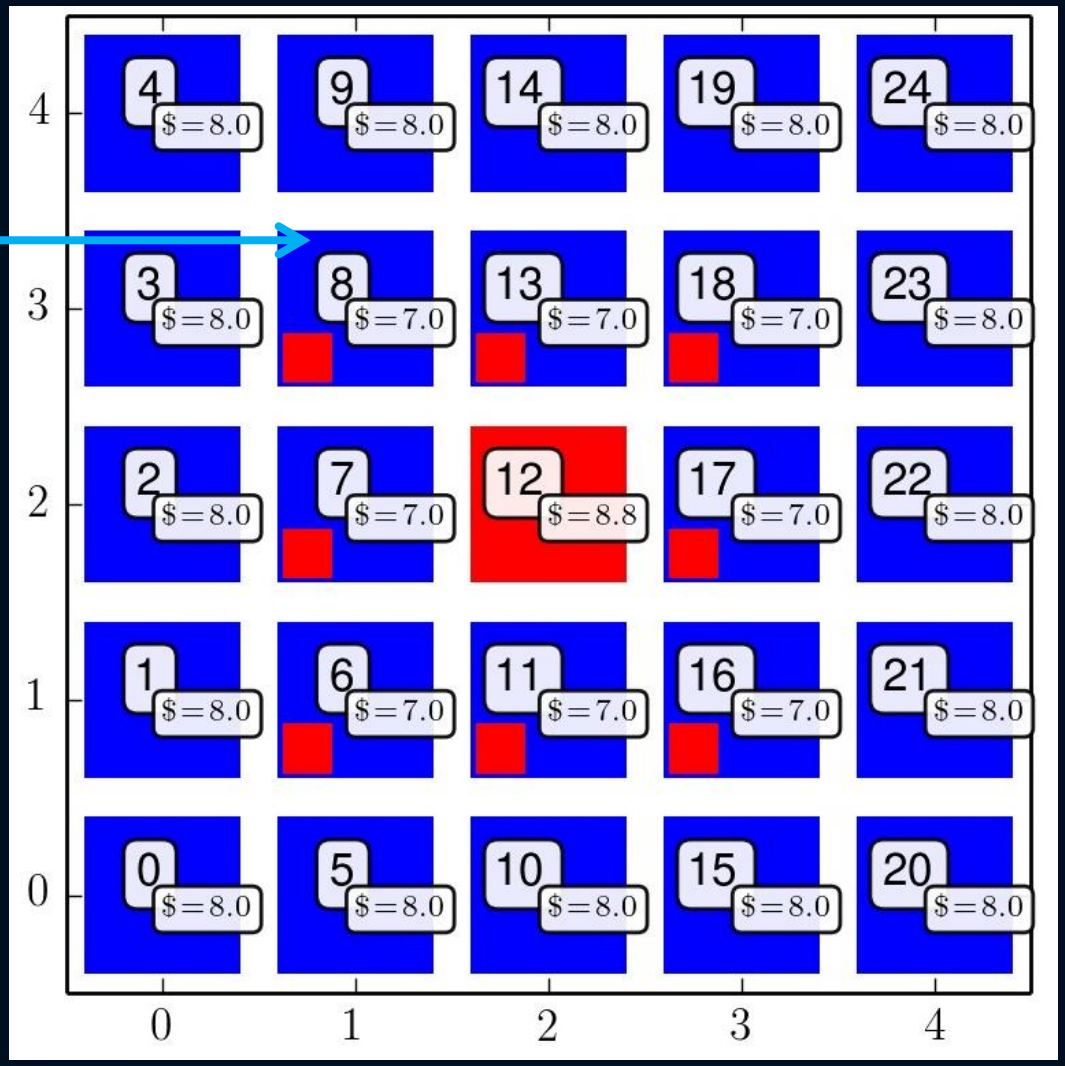
Spatial Games

In diesem Python Programm wird die Menge der Spieler (hier N=24) auf einem 2D-Gitter mit Moorschen Nachbarschaftsbedingungen angeordnet (siehe S:147 in M.A.Nowak, „Evolutionary Dynamics“). In jeder Iterations-periode spielt jeder Spieler mit seinen nächsten Nachbarn ein symmetrisches (2x2)-Spiel. Am Ende einer Periode vergleicht jeder Spieler seinen Gesamtgewinn mit seinen Nachbarn und bestimmt in einem „Update Rule“ seine Strategie in der nächsten Spielperiode.

	Spieler B Strategie 1 y=1	Spieler B Strategie 1 y=0
Spieler A Strategie 1 x=1	(a , a)	(b , c)
Spieler A Strategie 2 x=0	(c , b)	(d , d)

Die rechte Simulation benutzte die folgenden Werte der Auszahlungs-matrix (siehe linke Abb.): a=1, b=0, c=1.1 und d=0.01

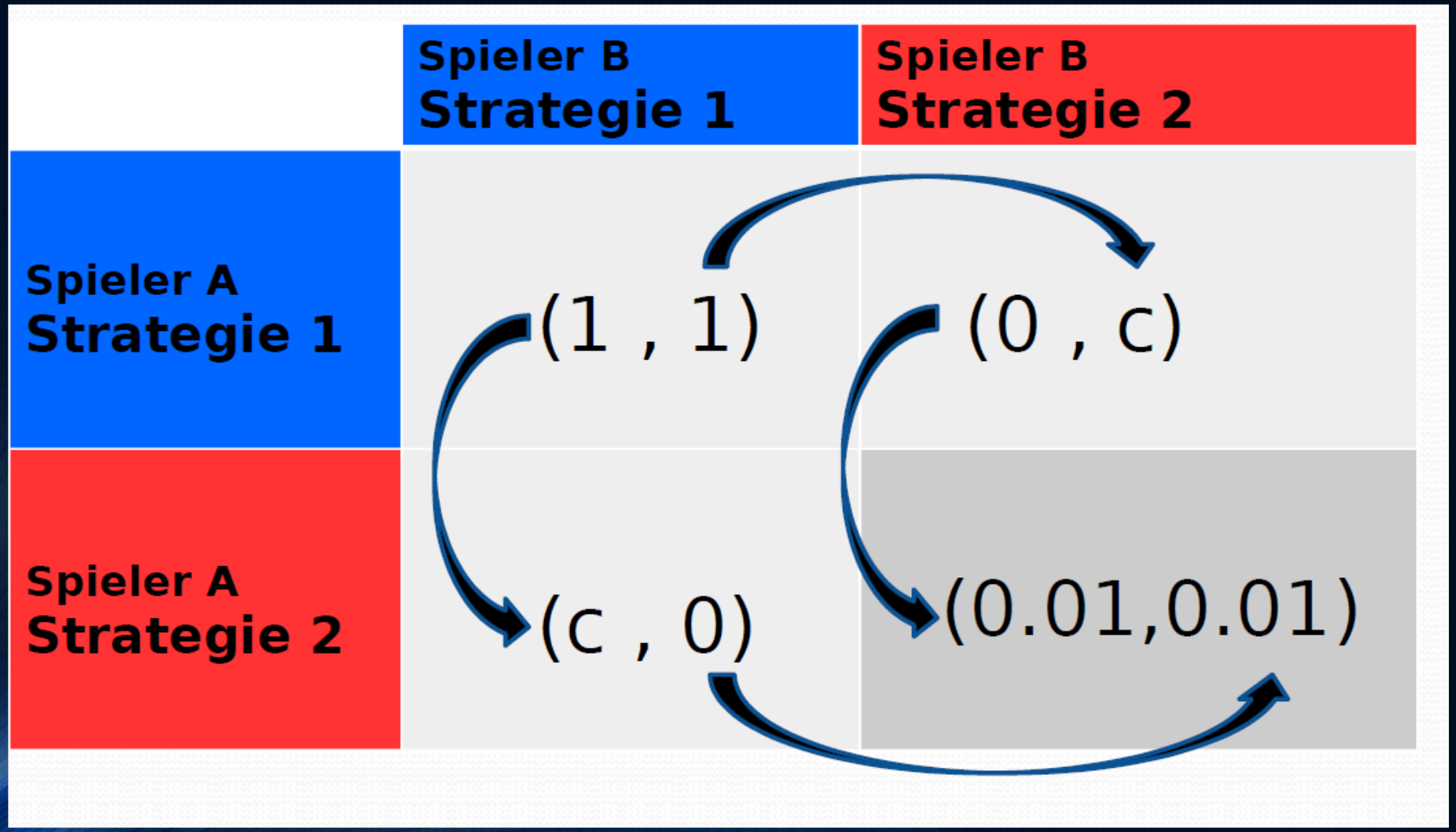
Beachte!: Definition von b und c ist in M.A.Nowak, „Evolutionary Dynamics“ vertauscht.

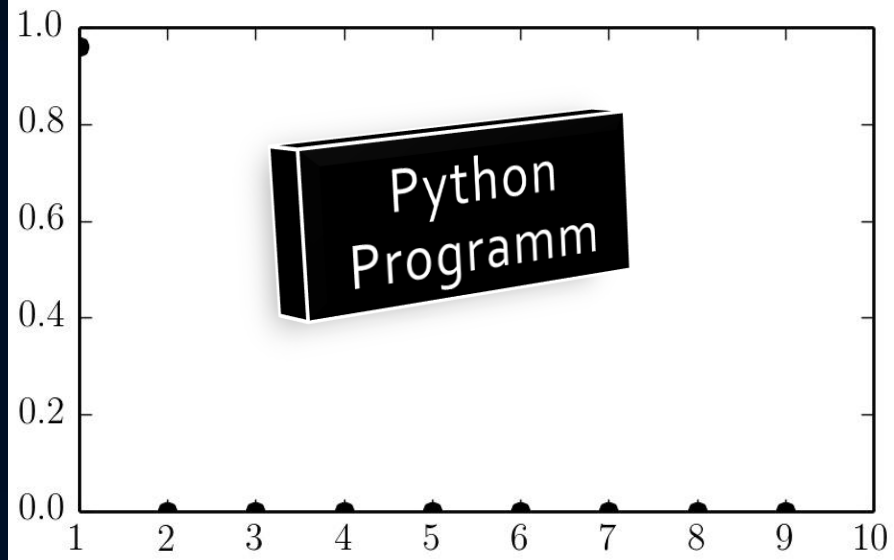


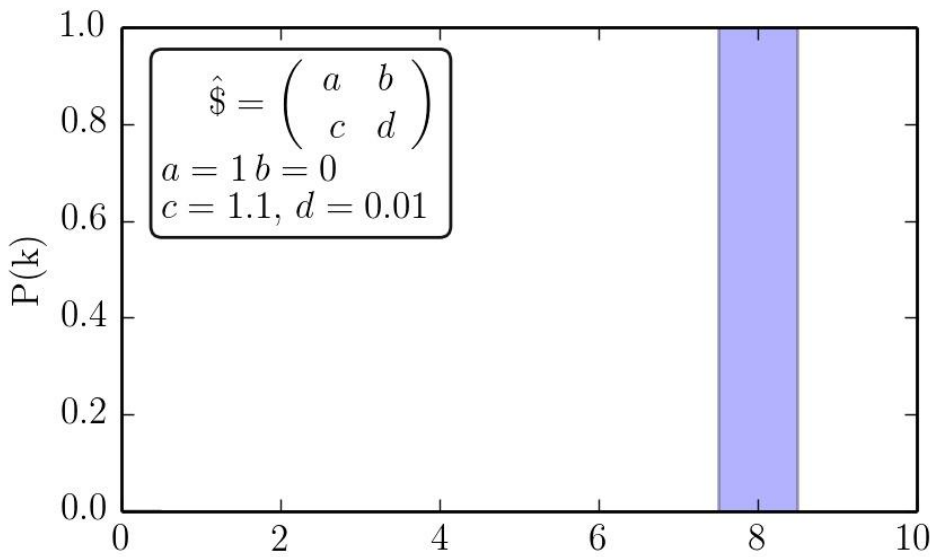
Update Rules und der Entscheidungsprozess

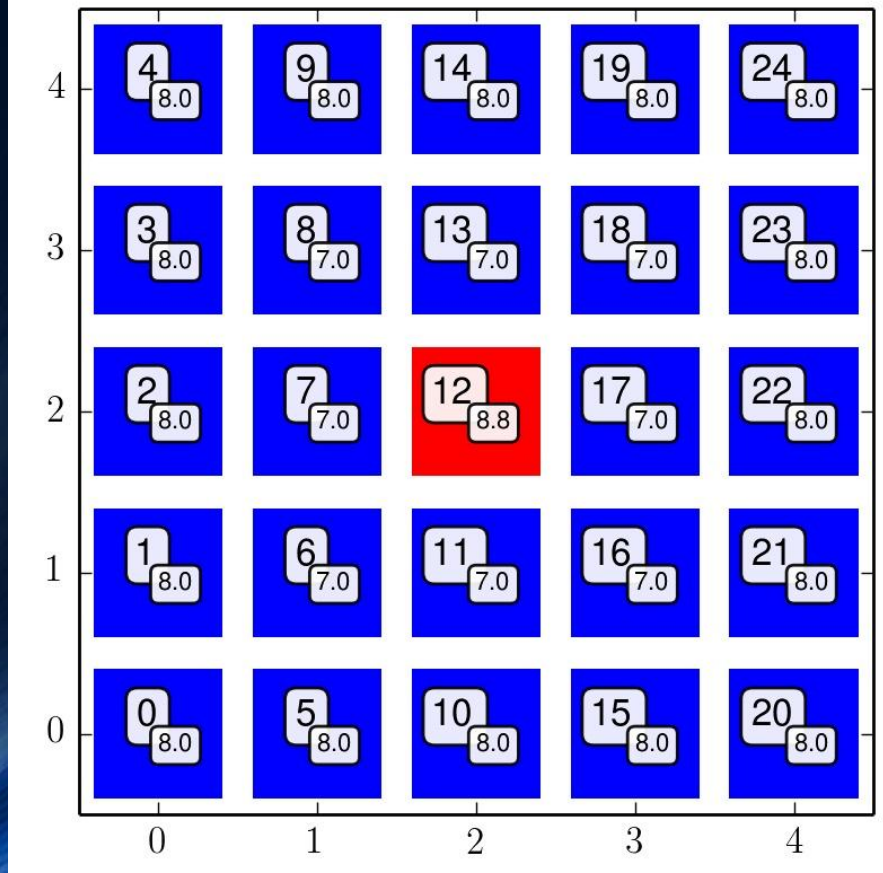
Spieler mit Knotennummer 8 hatte in der aktuellen Periode Strategie „blau“ gespielt und eine gesamte Auszahlung von $=7 erhalten. Er wird in der nächsten Periode „rot“ spielen (siehe kleines rotes Kästchen), da einer seiner nächsten Nachbarn (Knoten 12) eine höhere Auszahlung als er hatte und dieser die Strategie „rot“ spielte.

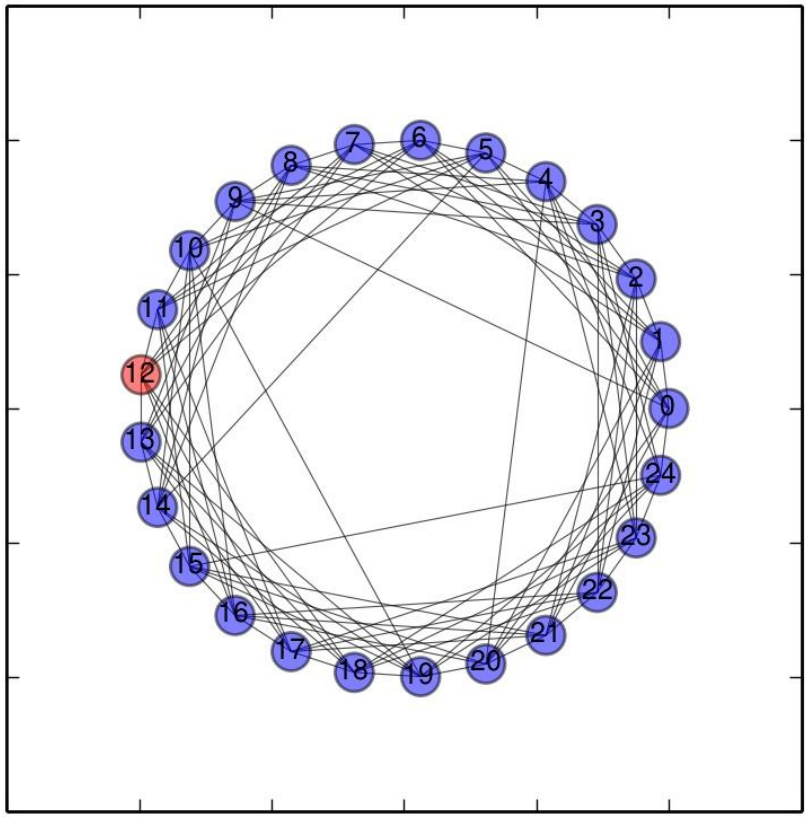
Betrachtetes Gefangenendilemma-ähnliches (2x2)-Spiel



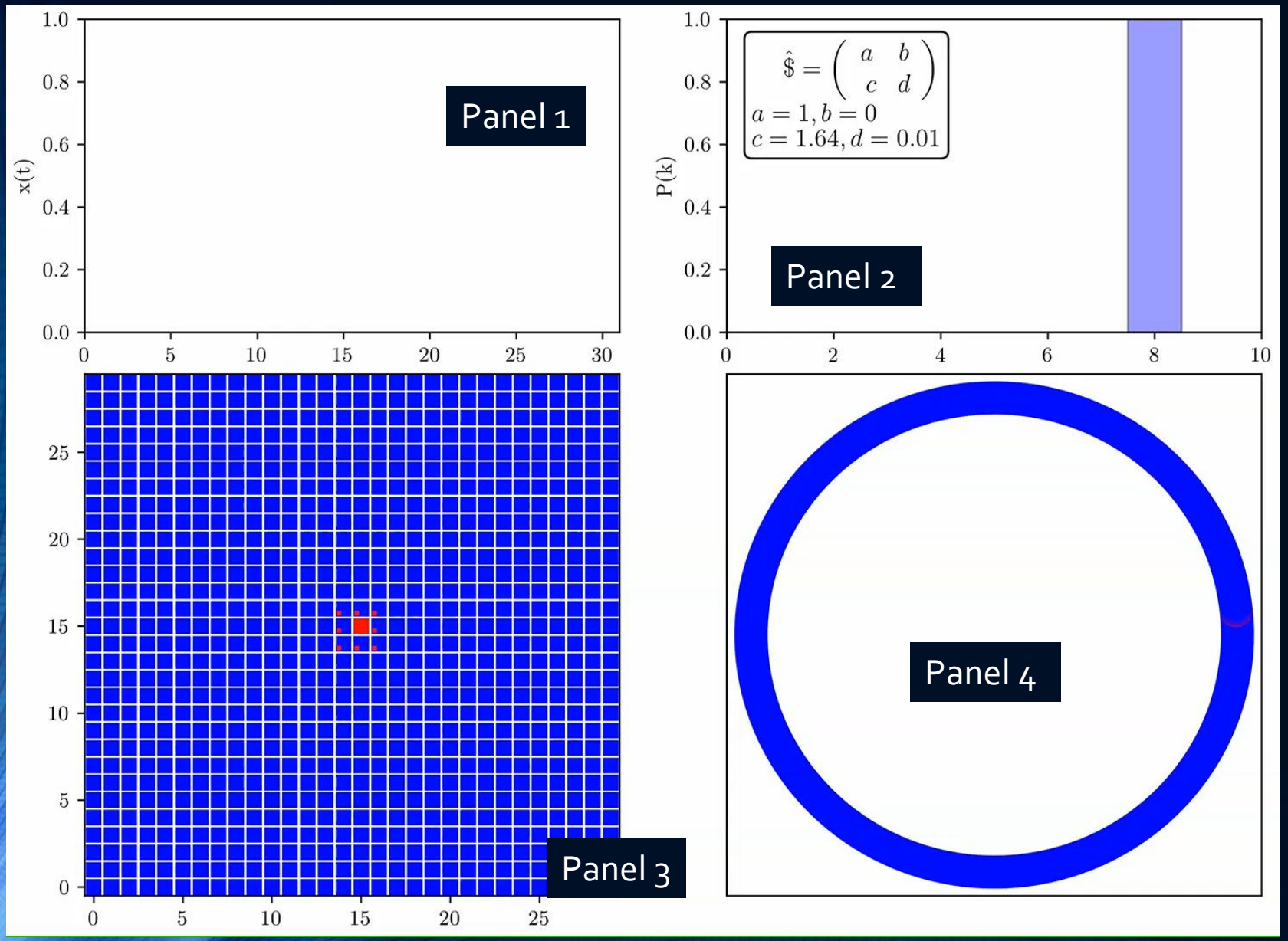

Python
Programm


$\hat{\$} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$
$a = 1\, b = 0$
$c = 1.1,\ d = 0.01$
P(k)


4 8.0
9 8.0
14 8.0
19 8.0
24 8.0
3 8.0
8 7.0
13 7.0
18 7.0
23 8.0
2 8.0
7 7.0
12 8.8
17 7.0
22 8.0
1 8.0
6 7.0
11 7.0
16 7.0
21 8.0
0 8.0
5 8.0
10 8.0
15 8.0
20 8.0



Evolutionäre Spieltheorie auf komplexen Netzwerken



Das Python Programm visualisiert in vier unterschiedlichen „Panels" die Evolution des „Spatial Games". In Panel 1 wird die zeitliche Entwicklung des Populationsvektors x(t) veranschaulicht. Panel 2 zeigt die Verteilungsfunktion der Knotengrade P(k) des zugrundeliegenden Moorschen Netzwerks. Panel 3 zeigt die Entwicklung der Strategieentscheidung der einzelnen Spielerknoten in der benutzten räumlichen Anordnung. Panel 4 veranschaulicht dagegen die Menge der Spieler in einem Kreis, geordnet nach ihrer Knotenzahl.

Neben der Auszahlungsmatrix, den implementierten Update Rules und der zugrundeliegenden Netzwerkstruktur hängt die zeitliche Entwicklung auch von den gewählten Anfangsbedingungen ab (hier wurde ein roter Spieler in einem Umfeld von blauen Spieler angeordnet).

Jupyter Notebook: Evolutionäre räumliche Spiele *Klasse der dominanten Spiele*

Physik der sozio-ökonomischen Systeme mit dem Computer

(Physics of Socio-Economic Systems with the Computer)

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Wintersemester 2025/26)

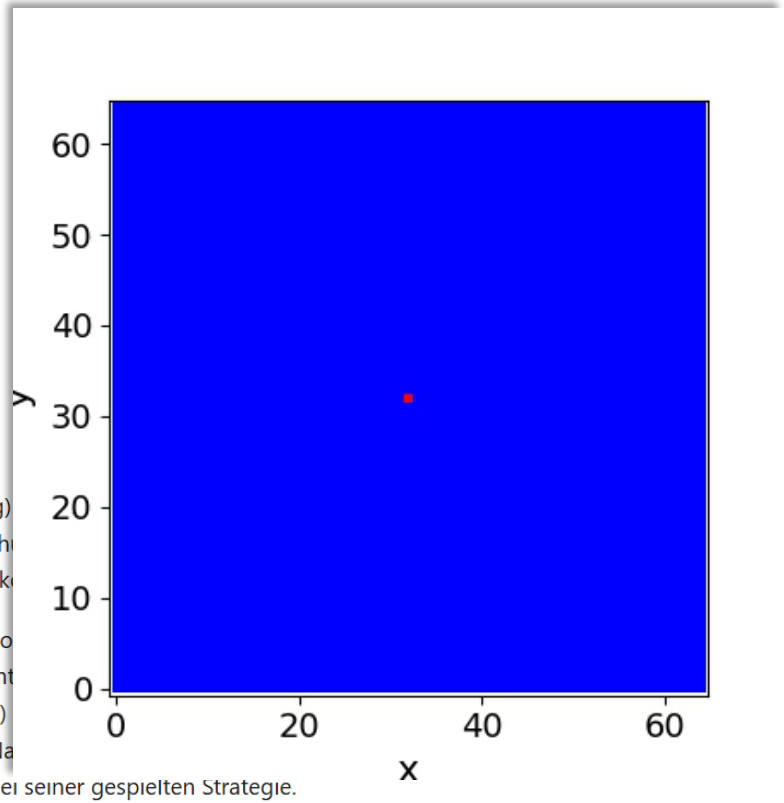
von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

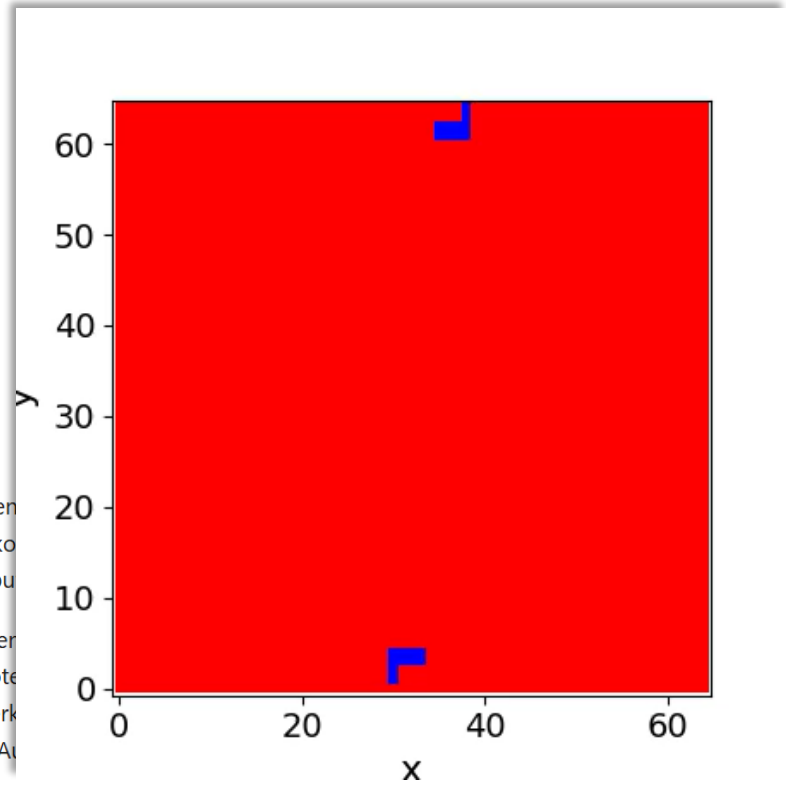
Frankfurt am Main 14.01.2026

Dritter Vorlesungsteil:

Evolutionäre räumliche Spiele (spatial games)

Beispiel: Dominante Spiele





Einführung

Die Verknüpfung der Theorie komplexer Netzwerke (siehe Teil II der Vorlesung) ersprechen Grundlagenforschung, als auch der angewandten, empirischen Netzwerkforsch iehung ko ie dann auf einem solchen komplexen Netzwerk ablaufenden Entscheidungsprozesse k er Compu

In diesem Jupyter Notebook werden die Spieler einer endlich großen Populatio n nächster werk der Spielerknoten besitzt somit eine einfache reguläre Struktur und im betracht ern (Knote (2x2)- Spiele und benutzen den Ansatz eines allgemeinen (2 Personen)-(2 Strategien) ie Spielerk m Zeitschritt vergleichen die Spieler ihren summierten Gewinn/Verlust mit den Na eigene Au ategie; ist sein eigener Wert der höchste, so bleibt er auch in der nächsten Iteration bei seiner gespielten Strategie.

Im Folgenden betrachten wir ein Beispiel, das an das 9. Kapitel des Buches Martin A. Nowak, Evolutionary Dynamics - Exploring the Equations of Life, 2006 angelehnt ist und ein Gefangenendilemma auf einem räumlichen 2-dimensionalen Gitter beschreibt. In Abhängigkeit der Stärke der Dominanz der Strategie und der Anfangskonfiguration der Strategienwahl der Spieler sind unterschiedliche zeitlichen Entwicklungen der Population möglich. Wir nehmen im Folgenden ein dominantes, symmetrisches 2x2-Spiel mit folgender Auszahlungsmatrix an:

$$\hat{\$} = \begin{pmatrix} 1 & 0 \\ c & 0.01 \end{pmatrix}$$

Physik der sozio-ökonomischen Systeme mit dem Computer

(Physics of Socio-Economic Systems with the Computer)

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Wintersemester 2025/26)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 14.01.2026

Dritter Vorlesungsteil:

Evolutionäre räumliche Spiele (spatial games)

Beispiel: Koordinations- und Anti-Koordinationsspiele

Einführung

Die Verknüpfung der Theorie komplexer Netzwerke (siehe Teil II der Vorlesung) mit der evolutionären Spieltheorie (siehe
Grundlagenforschung, als auch der angewandten, empirischen Netzwerkforschung dienen kann. In diesem Kapitel wird
auf einem solchen komplexen Netzwerk ablaufenden Entscheidungsprozesse können in den meisten Fällen lediglich mit

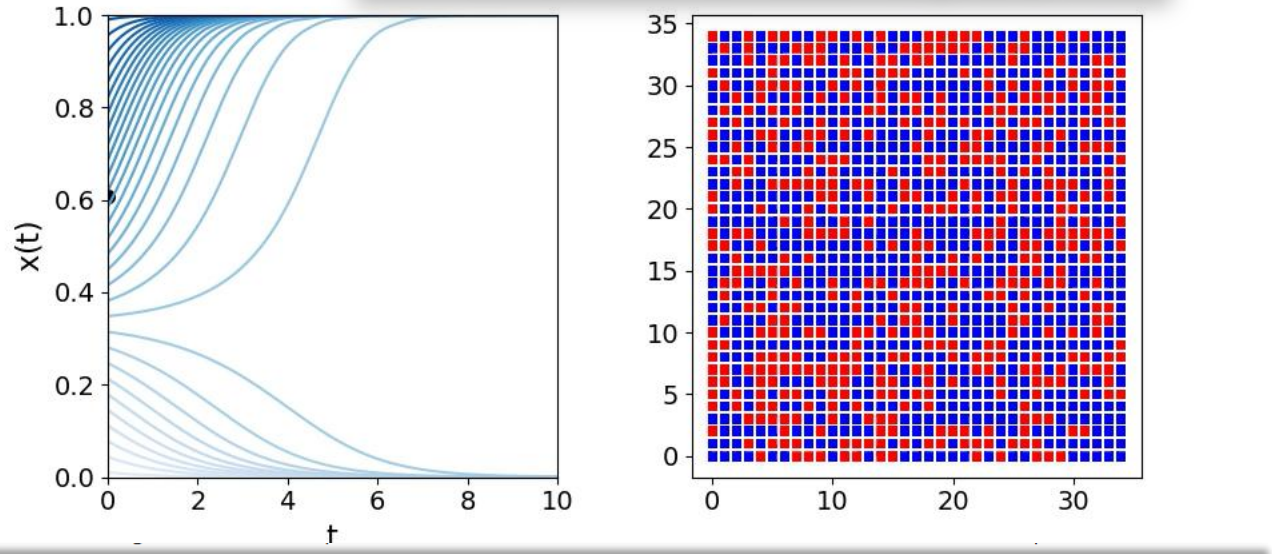
In diesem Jupyter Notebook werden die Spieler einer endlich großen Population auf einem räumlichen Gitter angeordne
der Spielerknoten besitzt somit eine einfache reguläre Struktur und im betrachteten 2-dimensionalen Fall spielt jeder Sp
Spiele und benutzen den Ansatz eines allgemeinen (2 Personen)-(2 Strategien) Spiels mit symmetrischer Auszahlungsma
Zeitschritt vergleichen die Spieler ihren summierten Gewinn/Verlust mit den Nachbarspielern ihres Umfeldes. Ist die Aus
ist sein eigener Wert der höchste, so bleibt er auch in der nächsten Iteration bei seiner gespielten Strategie.

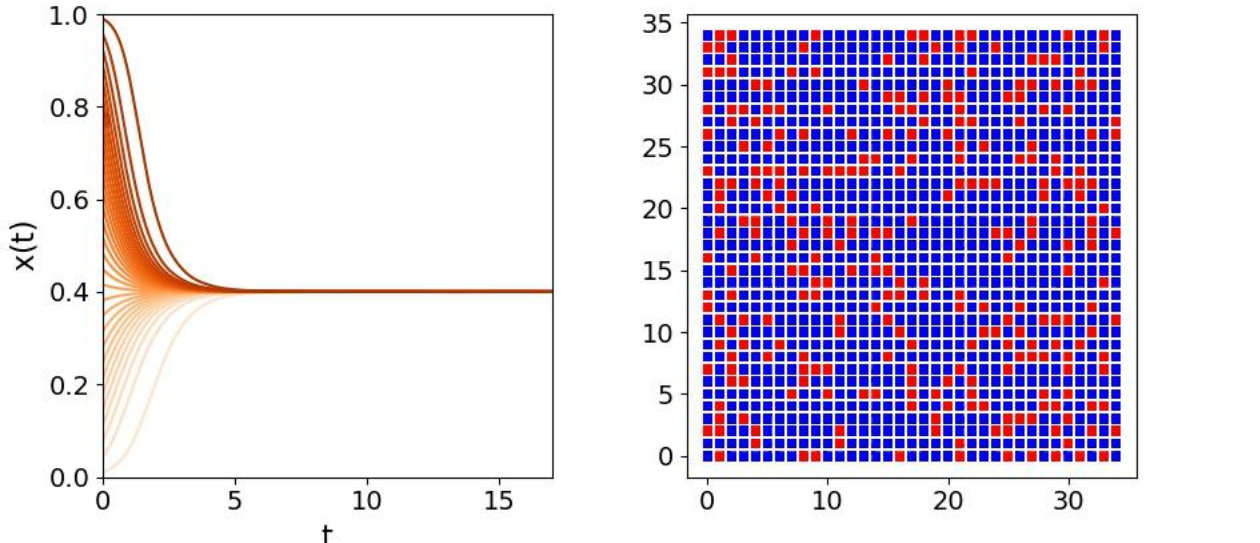
Im Folgenden betrachten wir Beispiele von Koordinations- und Anti-Koordinationsspielen und vergleichen die zeitliche E
evolutionären Spieltheorie (siehe Teil I der Vorlesung) betrachtete man eine unendlich große Population von Spielern, w
Replikatordynamik konnten wir dann das zeitliche Verhalten des Populationsvektors $x(t)$ (Anteil der Spieler, die die Stra

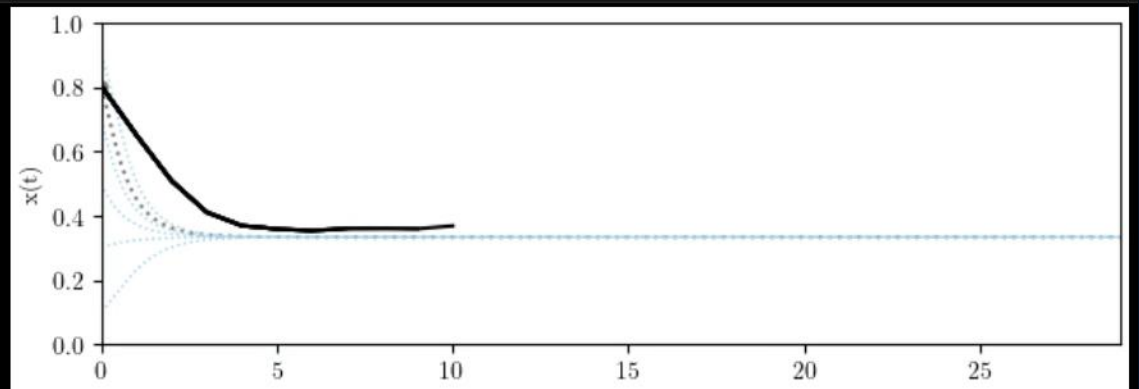
Wir nehmen im Folgenden ein allgemeines symmetrisches (2x2)-Spiel mit folgender Auszahlungsmatrix an:

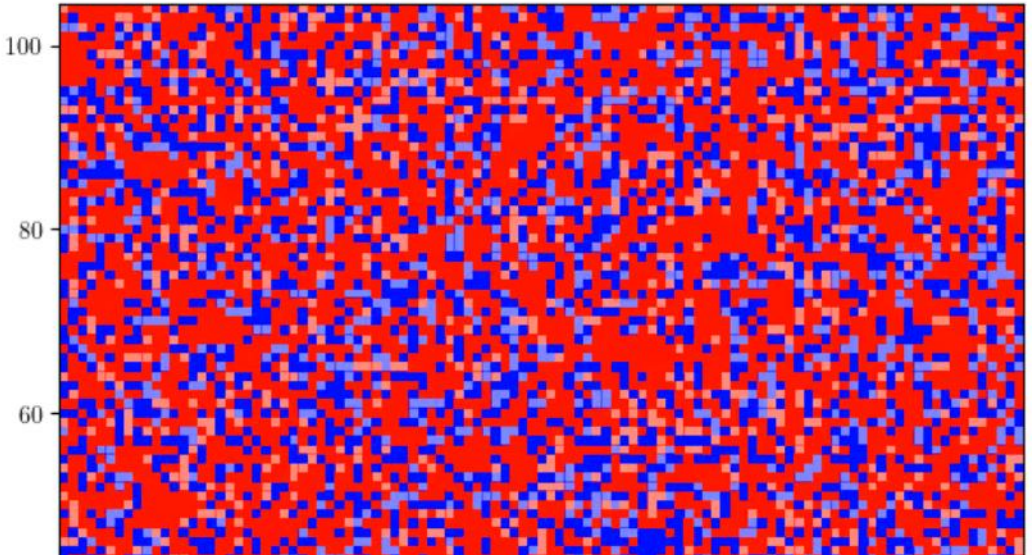
$$\hat{\$} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

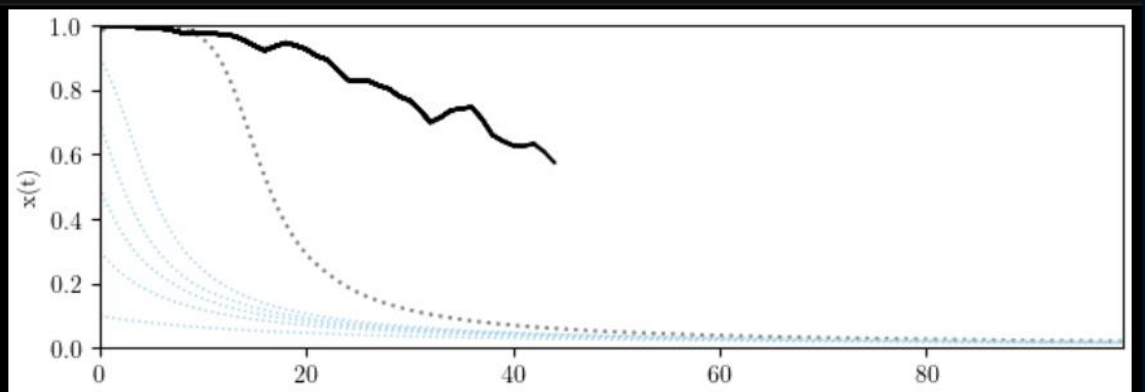
Jupyter Notebook:
Evolutionäre räumliche Spiele
Klasse der Koordinations- und Anti-Koordinationsspiele

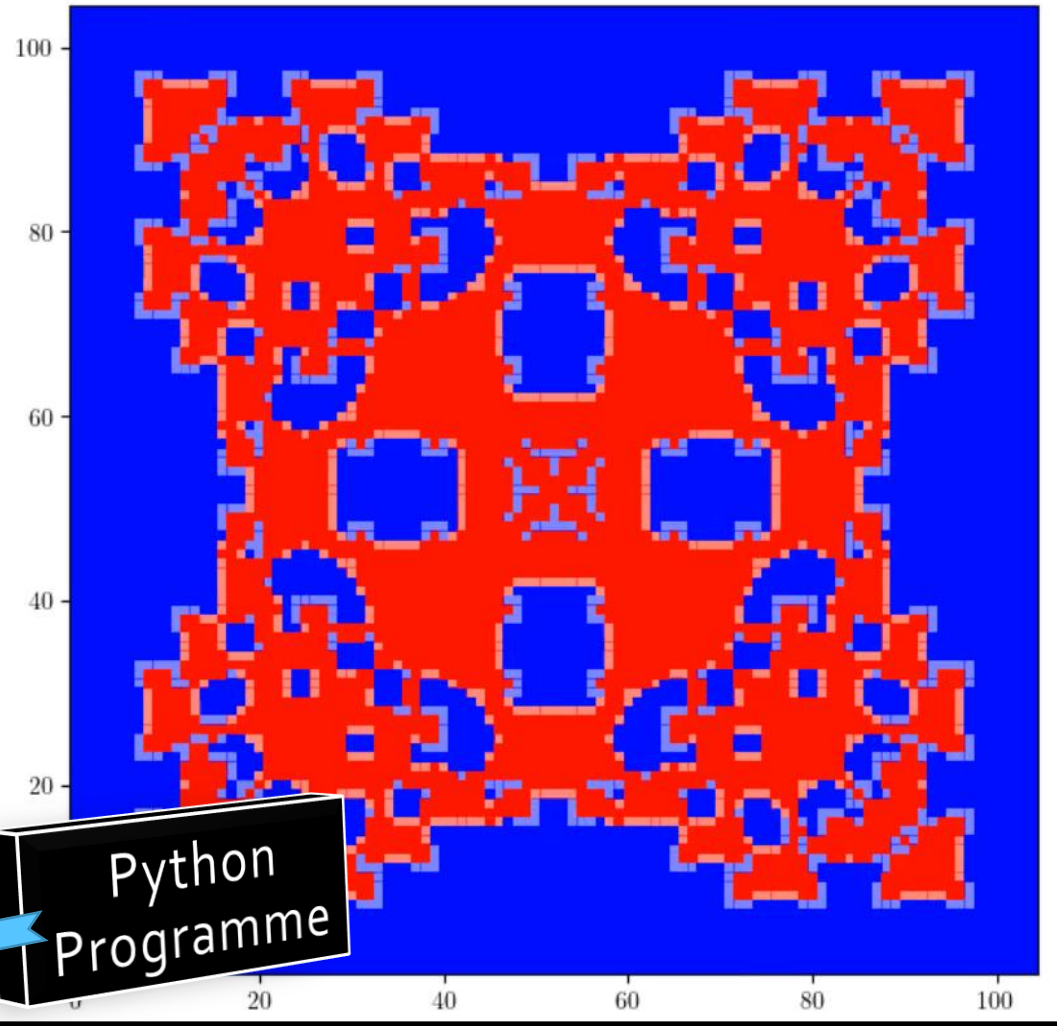












Weiterführende Links

Folien der 9.Vorlesung
Vorlesungsaufzeichnung der 9.Vorlesung: WS 2022/23 bzw. WS 2021/22
View Jupyter Notebook: Evolutionäre dominante räumliche Spiele
Download Jupyter Notebook: Evolutionäre dominante räumliche Spiele
View Jupyter Notebook: Evolutionäre räumliche Spiele: Koordinations- und Anti-Koordinationsspiele
Download Jupyter Notebook: Evolutionäre räumliche Spiele: Koordinations- und Anti-Koordinationsspiele
Download Python Programm: Räumliches Spiel (kleines Gitter mit Auszahlungen)
Download Python Programm: Räumliches Spiel (mittleres Gitter, Walker-Anfangskonfiguration)
Download Python Programm: Räumliches Spiel (großes Gitter): Version 1, Version 2

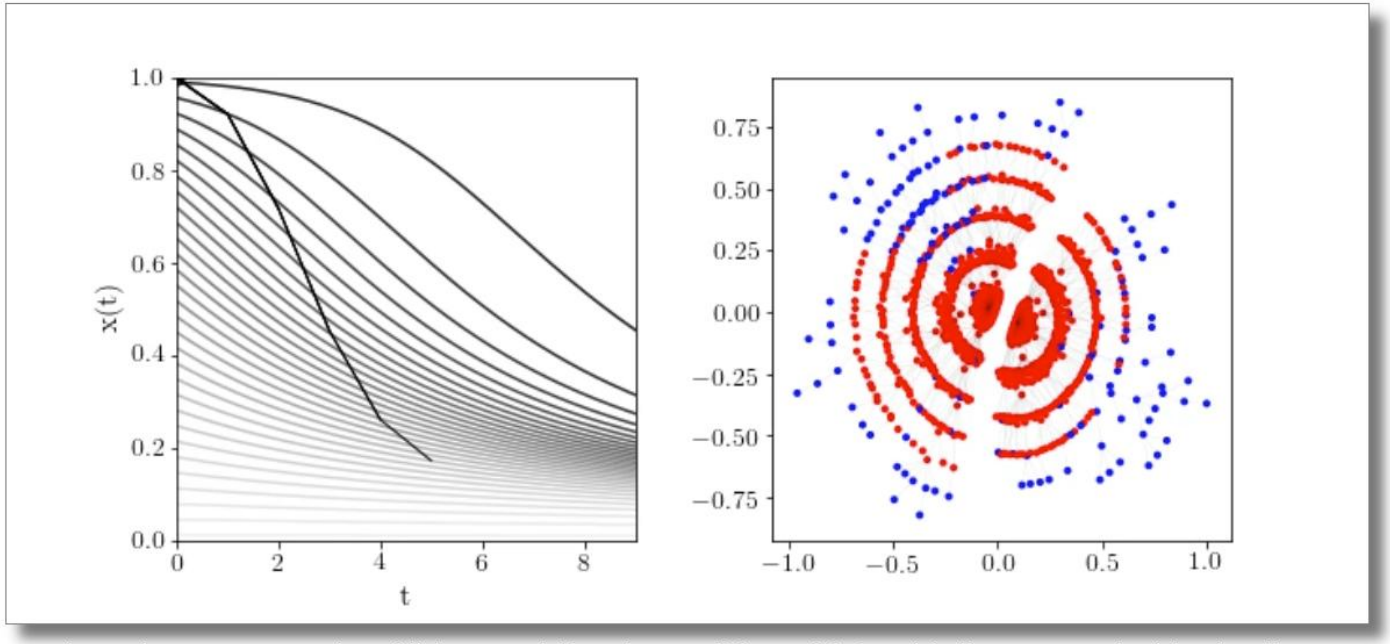
Python Programme

Vorlesung 10

In dieser Vorlesung wird nun die zeitliche Entwicklung der Strategienwahl der Population auf unterschiedlichen Netzwerktopologien analysiert und mit den Lösungen der deterministischen evolutionären Spieltheorie verglichen. Zunächst betrachten wir evolutionäre symmetrische (2×2) Spiele und dann, im zweiten Unterpunkt, analysieren wir symmetrische (2×3) Spielen auf räumlichen Gitterstrukturen und anderen Netzwerk-Topologien.

Evolutionäre symmetrische (2×2) Spiele auf unterschiedlichen Netzwerk-Topologien

Im Jupyter Notebook Evolutionäre Spiele auf unterschiedlichen Netzwerk-Topologien werden evolutionäre, symmetrische (2×2) Spiele auf unterschiedlichen Netzwerkklassen simuliert. Die evolutionäre Entwicklung der Strategienwahl der Spieler wird sowohl auf zufälligen, 'kleine Welt', skalenfreien, als auch auf vollständig verbundene Netzwerkstrukturen simuliert und mit den Resultaten der deterministischen evolutionären Spieltheorie verglichen. Es zeigt sich hierbei, dass die simulierten Spiele auf vollständig verbundene Netzwerken gut mit den Ergebnissen der Replikatordynamik übereinstimmen. Die unten dargestellte Animation zeigt die evolutionäre Entwicklung eines leicht dominanten Spiels ($c = 1.66$, siehe rechtes Panel der vorigen Vorlesung) auf einem skalenfreien Netzwerk. Bei den räumlichen Spielen entwickelte sich die Population zu einem Endzustand, bei dem alle Spieler die rote Strategie wählten. Die zeitliche Ausbreitung der roten Strategie auf einem skalenfreien Netzwerk hängt stark von der Anfangskonfiguration ab. In der unten dargestellten Animation wurde eine Anfangskonfiguration gewählt, bei der lediglich der Spieler mit dem größten Knotengrad (der Hub des skalenfreien Netzwerks) die rote Strategie wählt.

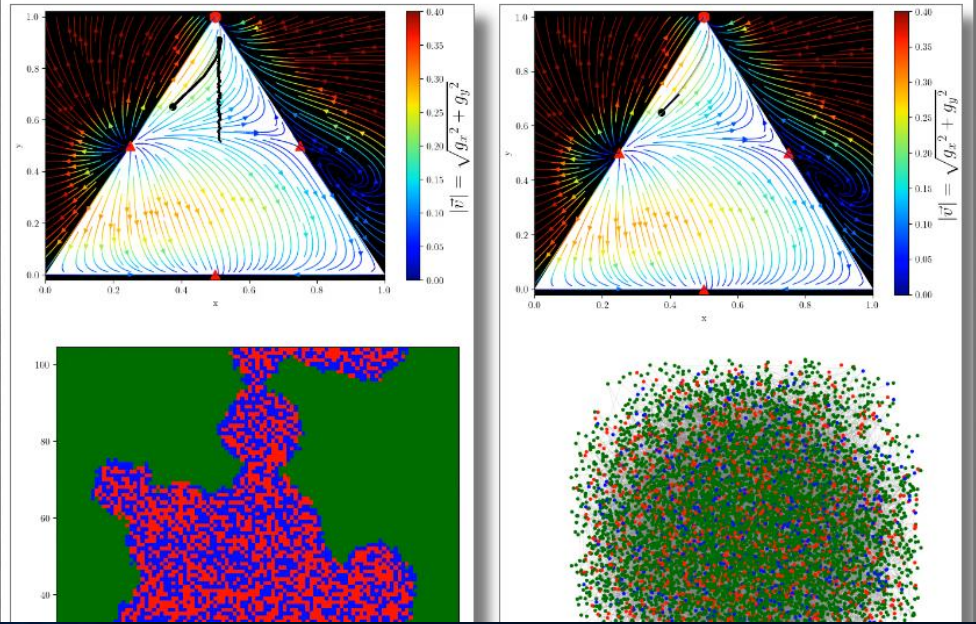


Man erkennt, im Gegensatz zu der zeitlichen Entwicklung im räumlichen Spiel, dass sich die rote Strategie nicht über die gesamte Population ausbreiten kann.

Vorlesung 10

Die zeitliche Entwicklung der Strategienwahl einer Population mittels eines analytischen mathematischen Modells abzubilden, ist das Bestreben der evolutionären Spieltheorie und die numerischen Lösungen der Replikatordynamik stellen Vorhersagen innerhalb der Modellvorstellung dieses Modells dar. Die in der Vorlesung besprochenen Modelleinschränkungen des deterministischen analytischen Modells legten dann eine stochastische Agenten-basierte Simulation nahe und in dieser Vorlesung analysieren wir evolutionäre Spiele auf unterschiedlichen Netzwerk-Topologien und vergleichen die simulierte Entwicklung der Population mit den Lösungen der deterministischen evolutionären Spieltheorie.

In dieser Vorlesung werden sowohl evolutionäre symmetrische (2×2), als auch (2×3) Spiele auf unterschiedlichen Netzwerk-Topologien simuliert (siehe linkes Panel dieser Vorlesung). Dabei zeigen die räumlichen und Netzwerk-Simulationen oft eine qualitative Übereinstimmung mit den Vorhersagen der deterministischen evolutionären Spieltheorie, wobei bei einigen Systemkonstellationen jedoch auch Unterschiede auftreten. Die unten dargestellten Animationen stellen zwei Simulationen symmetrischer (2×3) Spiele der Zeeman-Klasse 18 (siehe Vorlesung 5) dar, wobei bei der linken Abbildung eine räumliche Gitterstruktur zugrunde liegt und die rechte Simulation ein zufälliges Netzwerk verwenden.



Jupyter Notebook: Evolutionäre Spiele auf unterschiedlichen komplexen Netzwerken

Physik der sozio-ökonomischen Systeme mit dem C

(Physics of Socio-Economic Systems with the Comp

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Wintersemester 2025/26)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 14.01.2026

Dritter Vorlesungsteil:

Evolutionäre Spiele auf komplexen Netzwerken

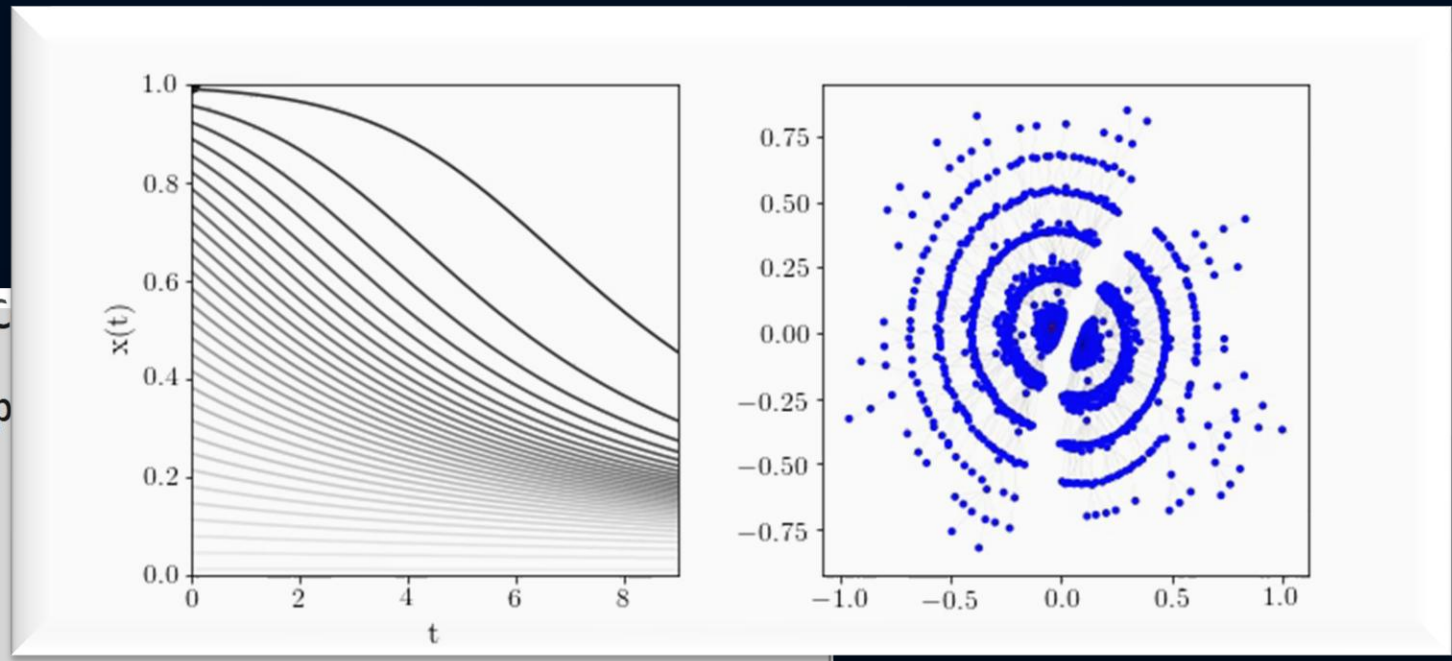
Beispiel: Evolutionäre Spiele auf unterschiedlichen Netzwerk-Topol

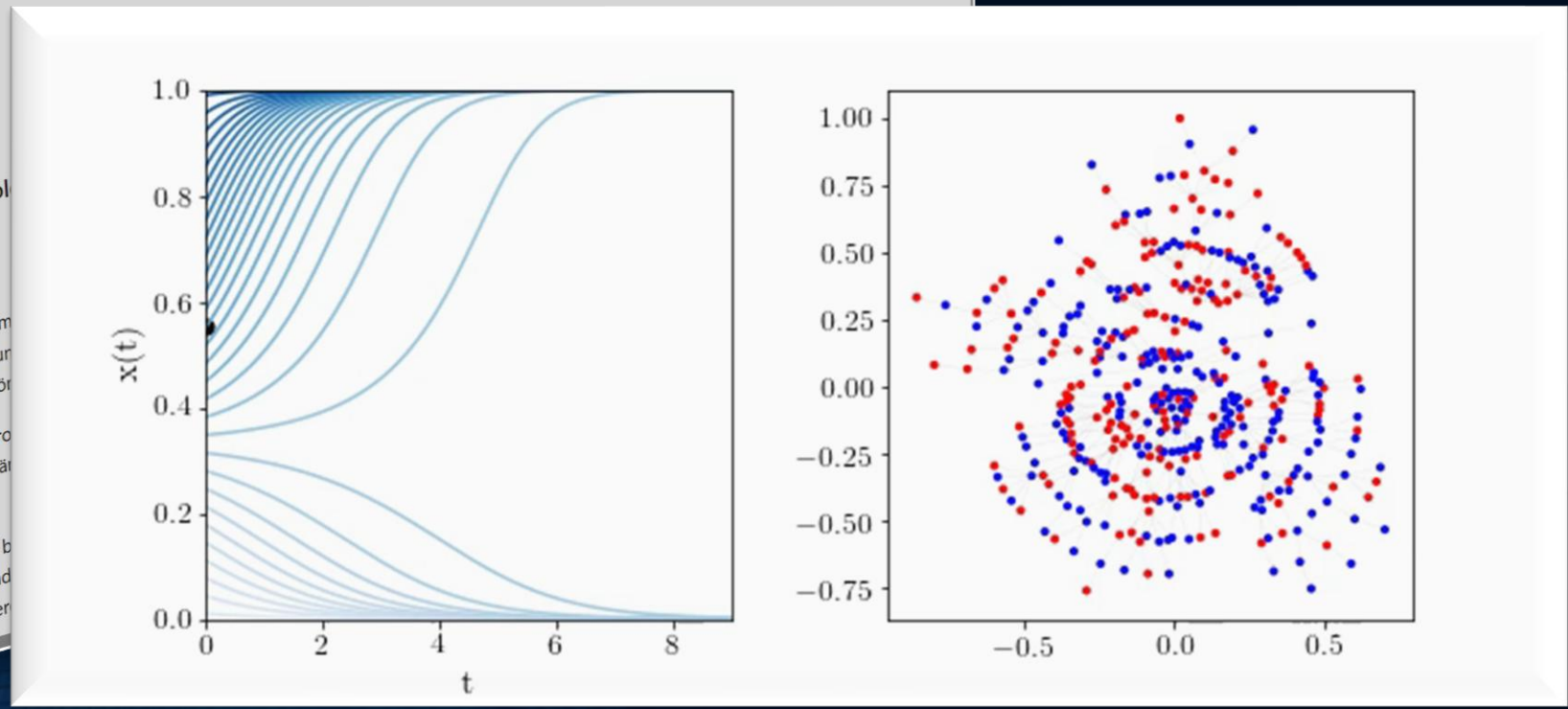
Einführung

Die Verknüpfung der Theorie komplexer Netzwerke (siehe Teil II der Vorlesung) m
Grundlagenforschung, als auch der angewandten, empirischen Netzwerkforschun
auf einem solchen komplexen Netzwerk ablaufenden Entscheidungsprozesse kön

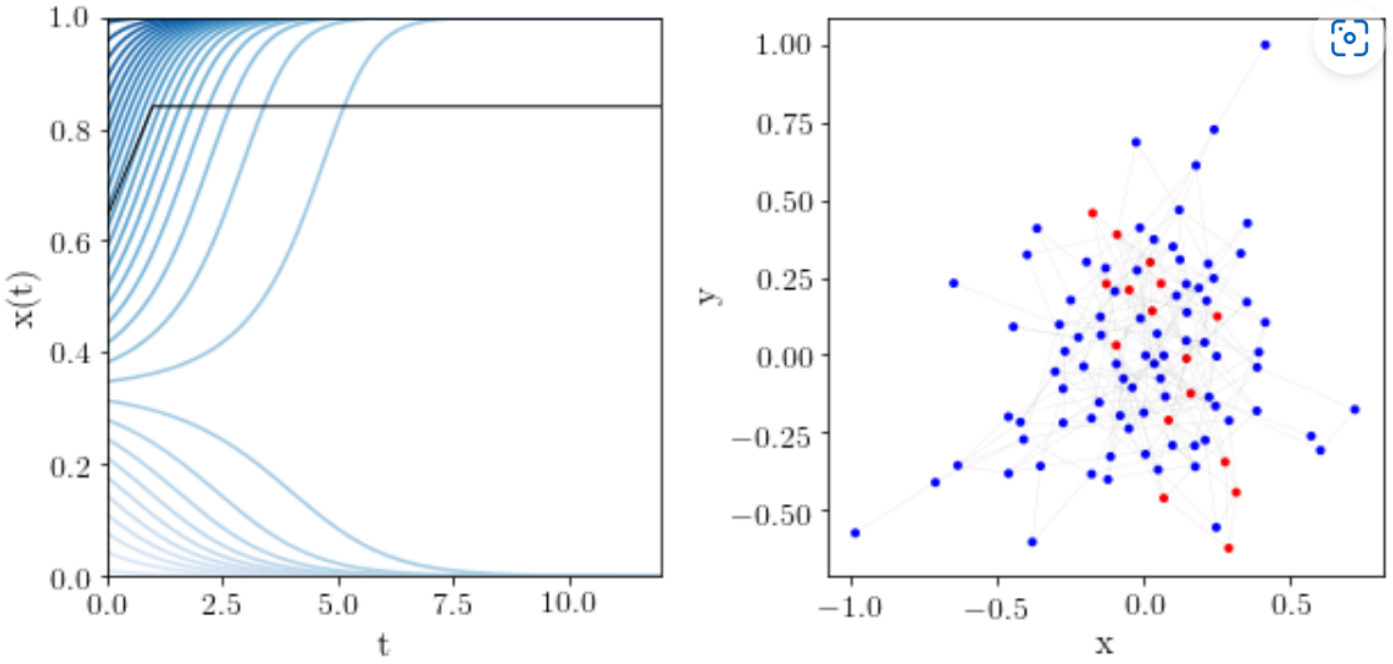
In den beiden letzten Jupyter Notebooks hatten wir die Spieler einer endlich gro
zugrundeliegende Netzwerk der Spielerknoten hatte somit eine einfache regulä
Notebook wollen wir das Spiel auf unterschiedlichen Netzwerken spielen.

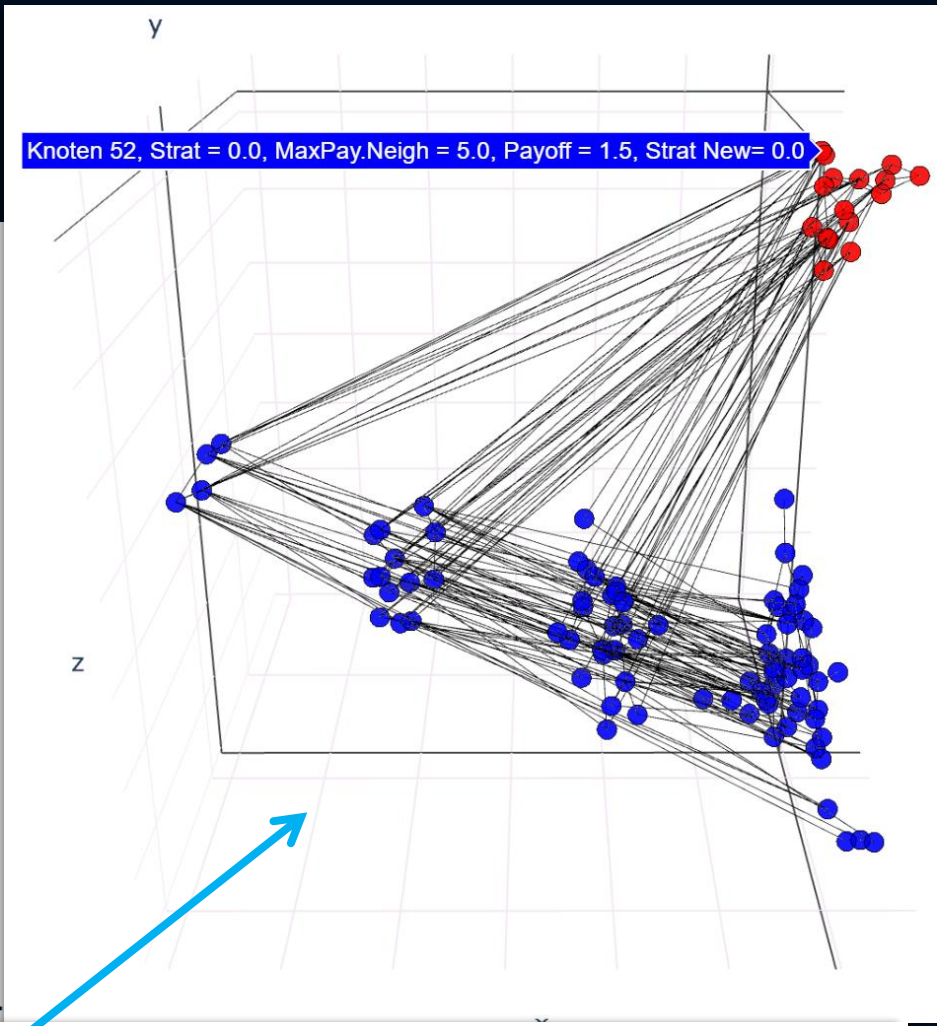
Wir beschränken uns im folgenden wieder auf symmetrische (2x2)-Spiele und b
mit ihren k-nächsten direkten Nachbarn im Netzwerk und am End
Auszahlung eines Spielers höher als der eigene mittler





Evolutionäre Spieltheorie auf zufälligen Netzwerken

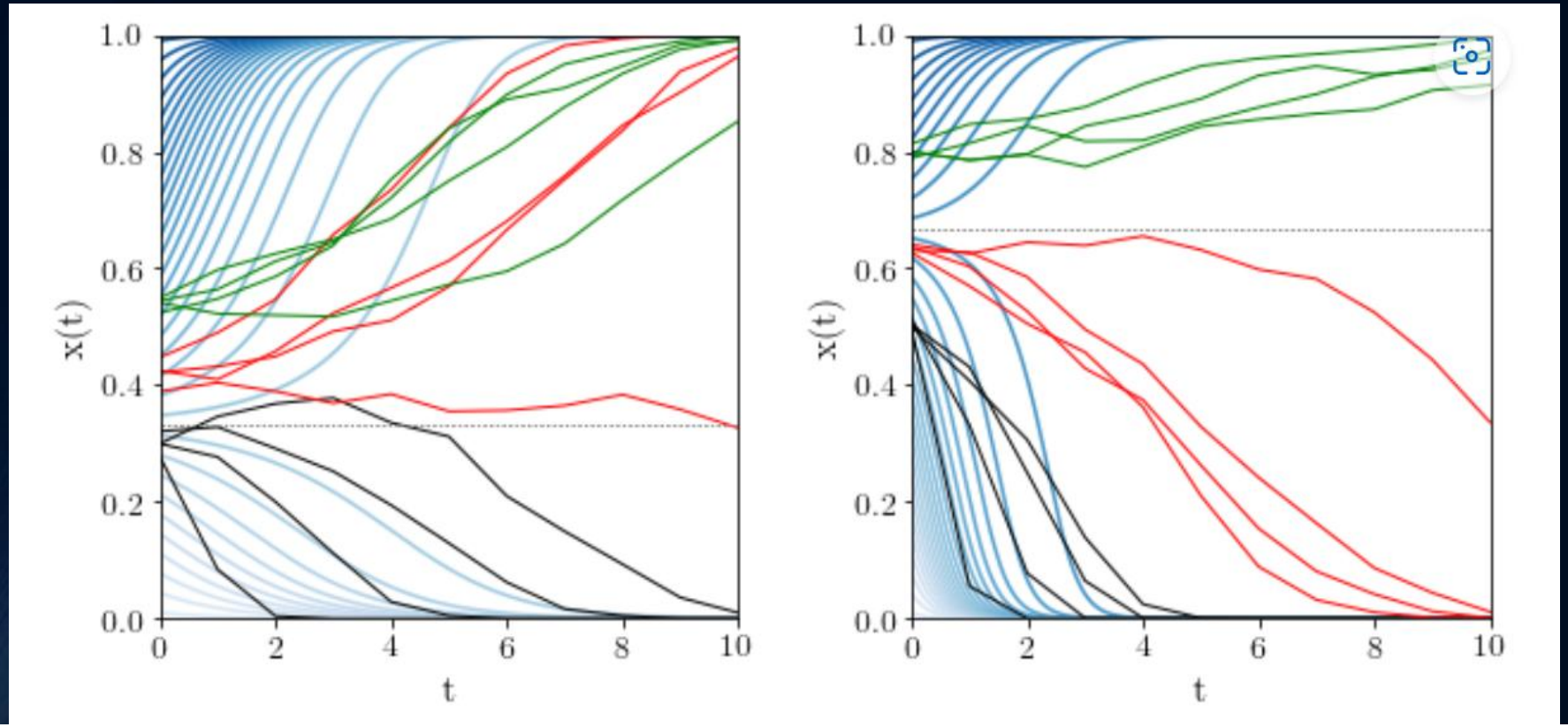




Die obige Abbildung stellt das letzte Bild des simulierten evolutionären Spiels auf dem zufälligen Netzwerk dar.

Die obere Abbildung zeigt das Netzwerk bei einer festgelegten Zeititeration. Die roten Spieler sind um einen gewissen Betrag in z-Richtung verschoben, die Nachbarn der roten Spieler um einen gewissen Betrag in xz-Richtung und Spieler, die ihre Strategie in der nächsten Spielperiode verändern, sind um einen gewissen Betrag in y-Richtung verschoben (in der oberen Abbildung tritt diese Situation aufgrund des statischen Endzustandes jedoch nicht auf). Zusätzlich kann man in der interaktiven Grafik die Knotennummer, die aktuelle und zukünftige Strategie, den erzielten Payoff und den maximalen Payoff der Nachbarn erkennen, wenn man mit der Maus in die Nähe eines Knotens gelangt. Die einzelnen Eigenschaften des Knotens und seiner Nachbarn können auch wie folgt ausgegeben werden:

Evolutionäre Spieltheorie auf vollständig verbundenen Netzwerken
Koordinationsspiele

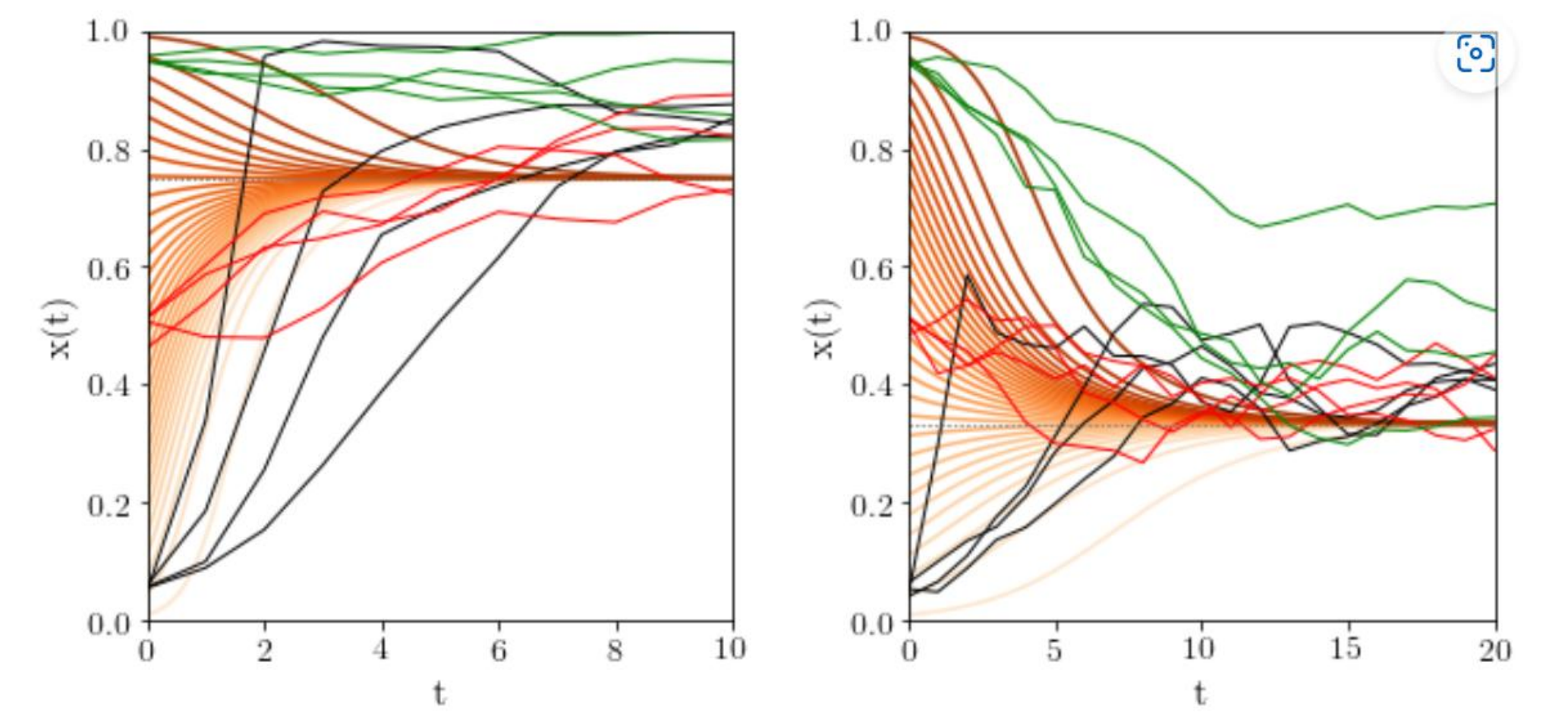


Die oberen Abbildungen zeigen die simulierten Ergebnisse für zwei Koordinationsspiele mit unterschiedlichem gemischten Nash-Gleichgewicht (siehe graue gepunktete Linien). Die simulierten Populationsvektoren (siehe schwarze, rote und grüne Kurven) stimmen gut mit den Vorhersagen der analytischen evolutionären Spieltheorie überein, wobei die Form der zeitlichen Entwicklung stark vom Zufall bestimmt ist.

Das oben abgebildete evolutionäre Spiel eines Koordinationsspiels auf einem vollständig verbundenen Netzwerk, in dem jeder Spieler pro Spielperiode acht Spielpartner sucht und mit ihnen das Spiel spielt ($< k >_{\text{eff}} \approx 8$) wird nun durch ein Anti-Koordinationsspiel ersetzt.

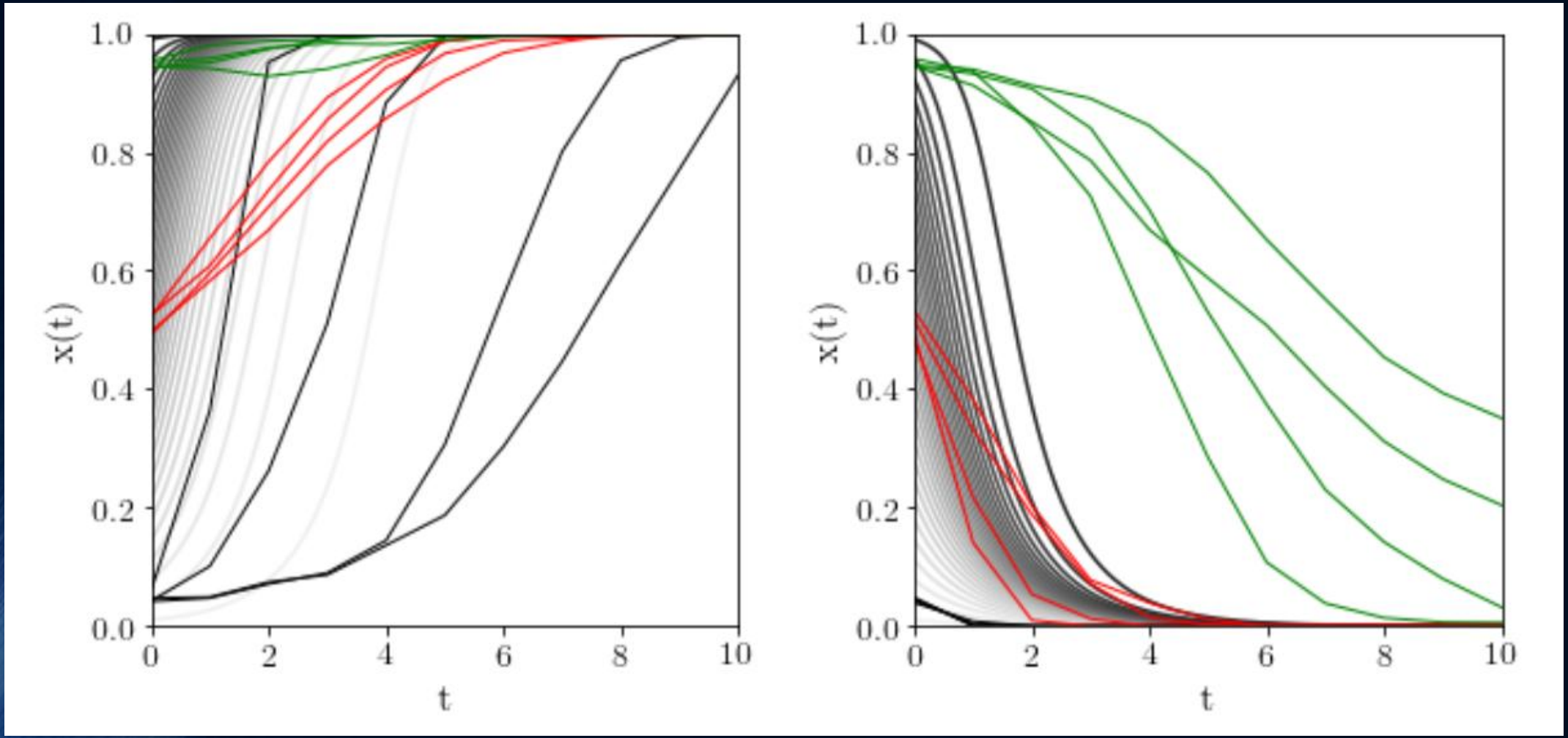
Evolutionäre Spieltheorie auf vollständig verbundenen Netzwerken

Anti-*Koordinationsspiele*



Evolutionäre Spieltheorie auf vollständig verbundenen Netzwerken

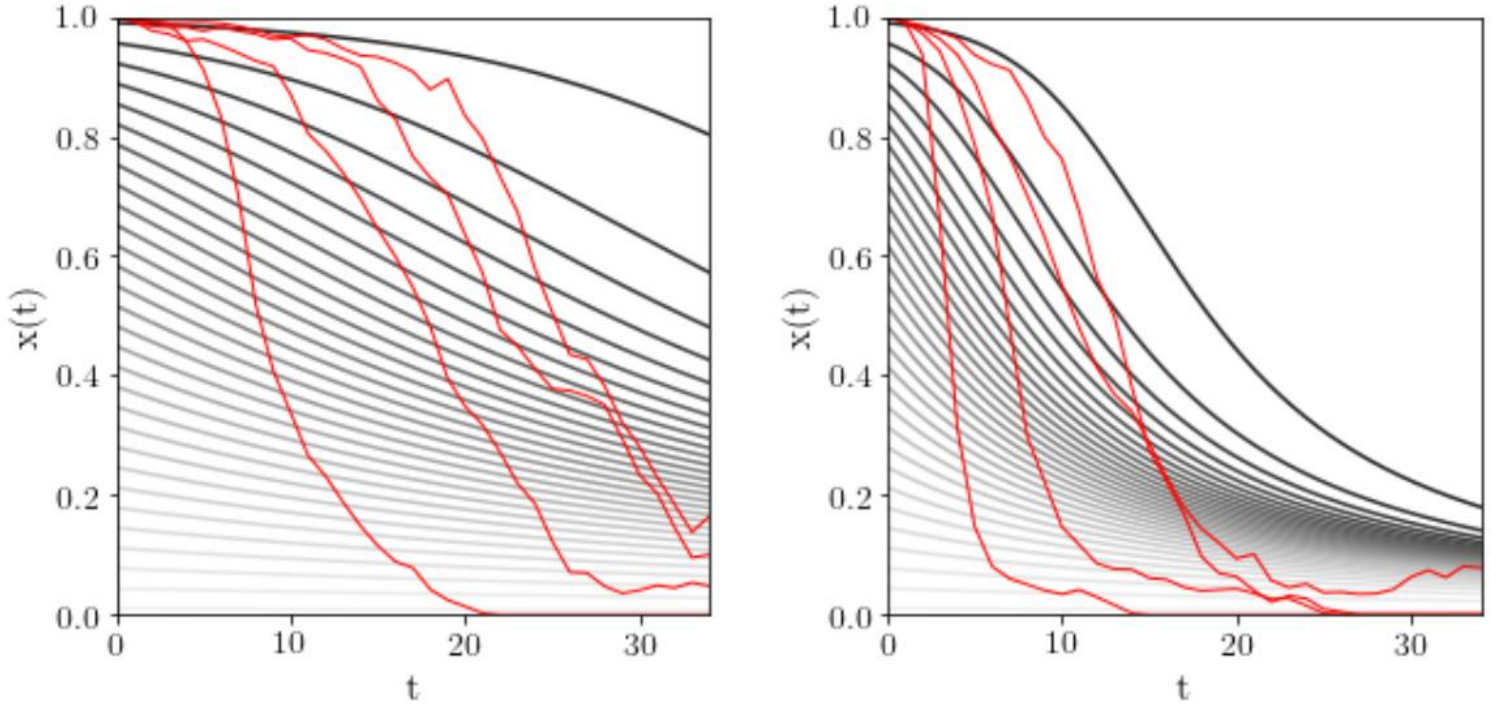
Dominante Spiele



Evolutionäre Spieltheorie auf vollständig verbundenen Netzwerken
Dominante Spiele

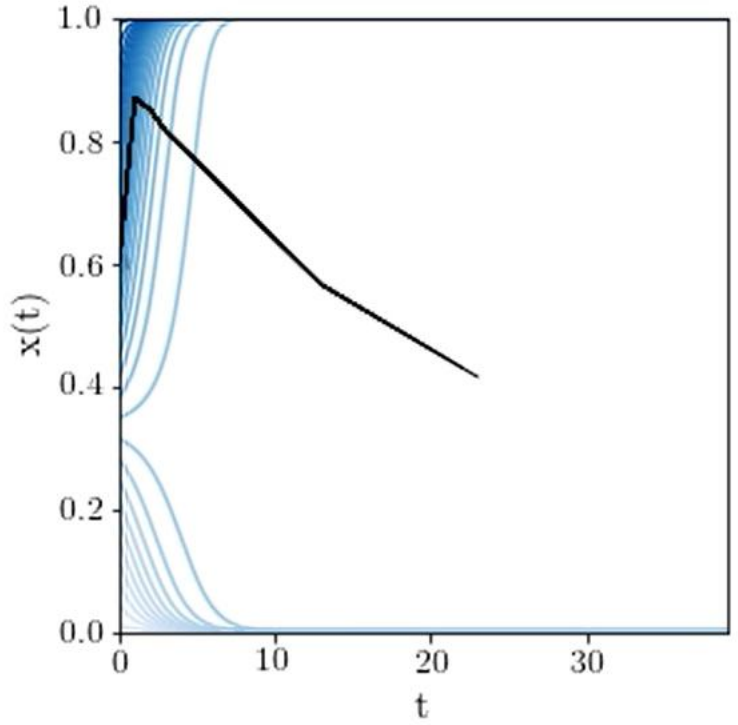
Bei den räumlichen dominanten Spielen hatten wir gesehen, dass sich die dominante Strategie nur ab einer gewissen Stärke der Dominanz durchsetzen kann. Dies wollen wir nun auf unserem Netzwerk untersuchen und setzen dafür die Auszahlungswerte der beiden dominanten Spiele wie folgt an:

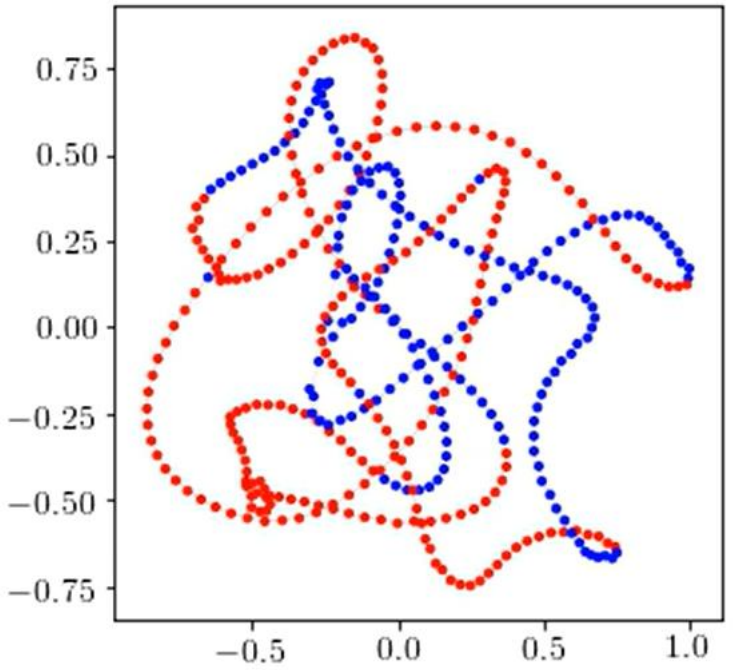
Linke Abbildung: $\hat{\$} = \begin{pmatrix} 1 & 0 \\ c = 1.1 & 0.01 \end{pmatrix}$, Rechte Abbildung: $\hat{\$} = \begin{pmatrix} 1 & 0 \\ c = 1.3 & 0.01 \end{pmatrix}$. Da sich aufgrund der schwachen Dominanz der Spiele (kleine c-Werte) die zeitliche Entwicklung verlangsamt, simulieren wir das Spiel nicht nur 11, sondern 35 Iterationen:



Man erkennt im Gegensatz zu den Ergebnissen der räumlichen Spiele, dass sich auch bei geringer Dominanz des Spiels (links $c = 1.1$, rechts $c = 1.3$) die zweite Strategie sich durchsetzt, wie es von der Replikatordynamik vorhergesagt wird.

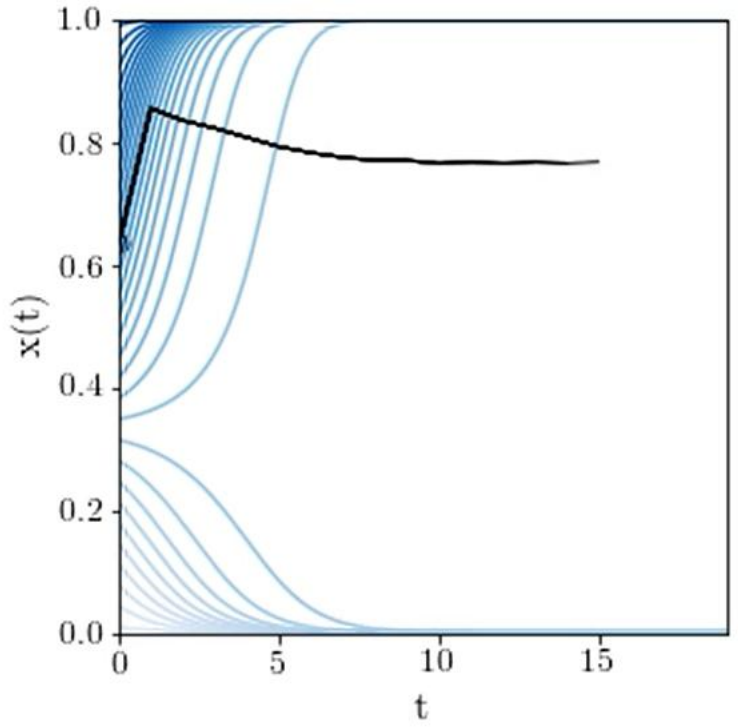
Evolutionäre Spieltheorie auf Ringgitter Netzwerken

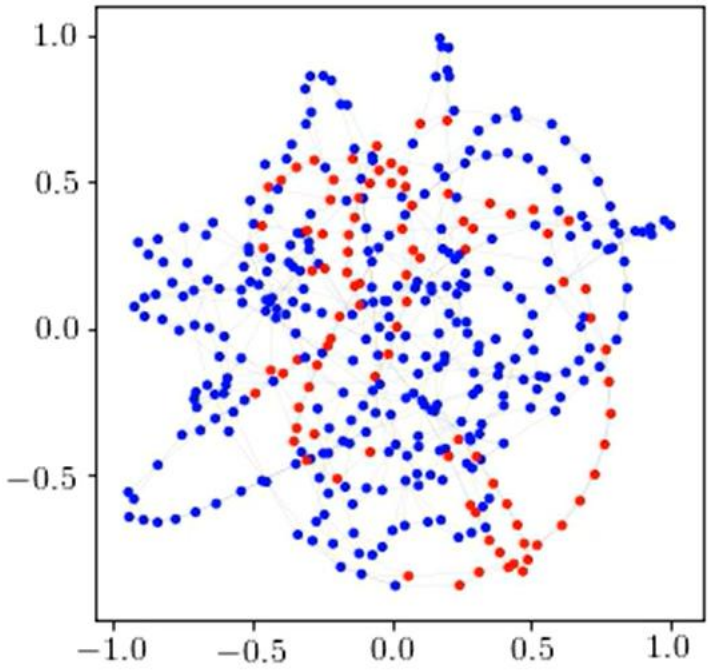




Man erkennt, dass sich die rote Strategie im Laufe der Zeit immer mehr auf dem Ringgitter durchsetzt, die zeitliche Ausbreitung jedoch sehr langsam ist. Wir betrachten uns nun die zeitliche Entwicklung auf einem "kleine Welt"-Netzwerk und erzeugen nach der Vorgehensweise von Watts und Strogatz 50 zufällige Spielerverbindungen:

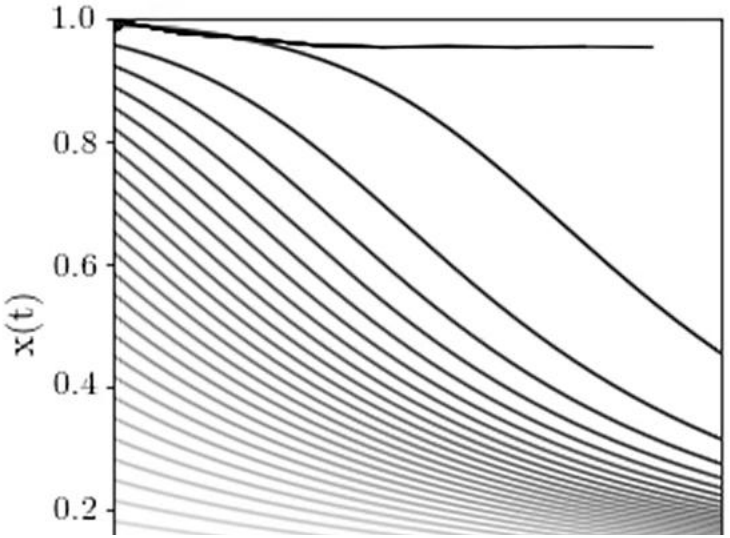
Evolutionäre Spieltheorie auf Kleine Welt Netzwerken

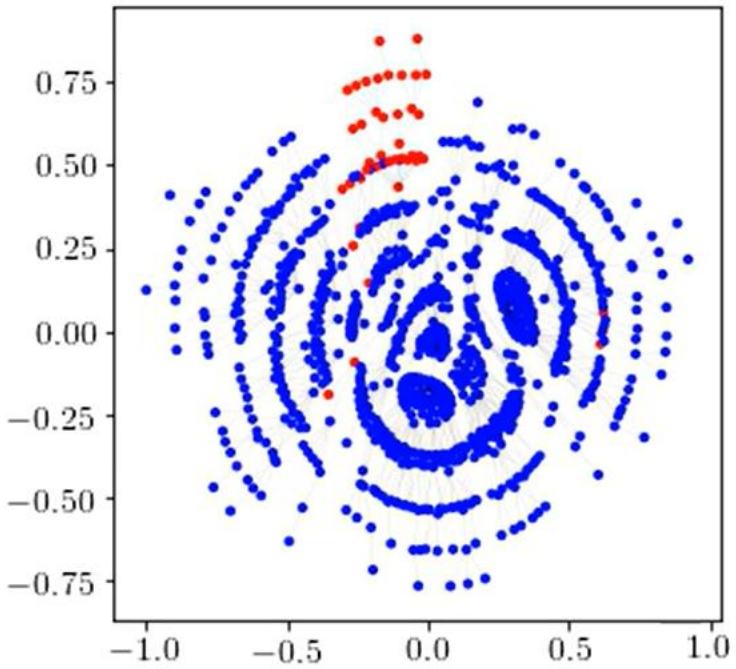


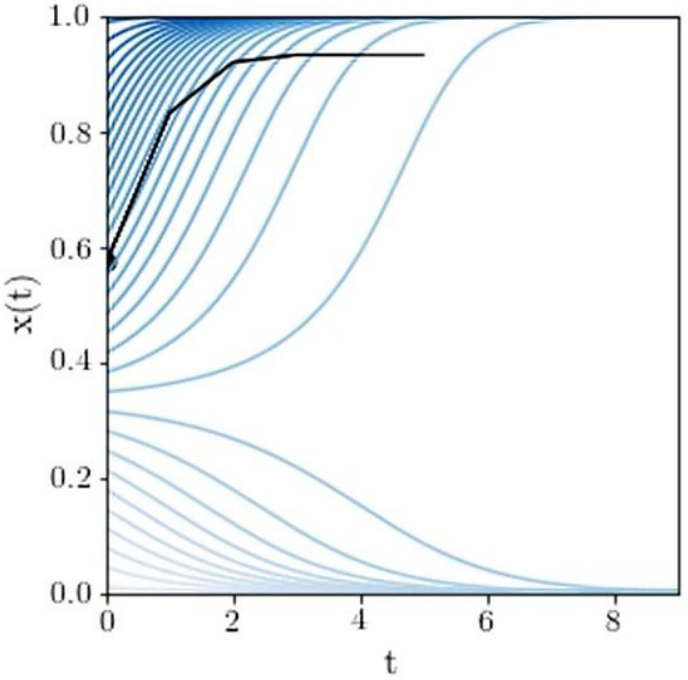


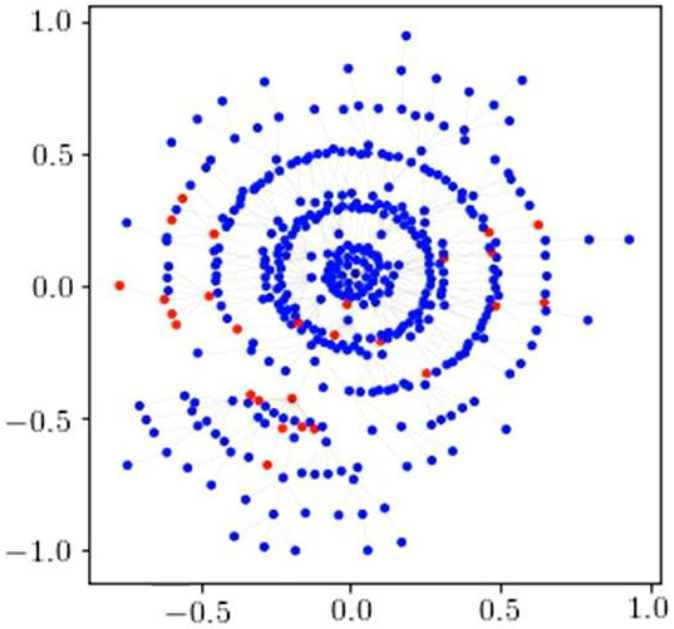
Bei dieser Simulation auf einem kleine Welt Netzwerk kann sich die rote Strategie nicht durchsetzen und der Endzustand der Population besteht lediglich aus einigen roten Gruppen-Clustern.

Evolutionäre Spieltheorie auf skalenfreien Netzwerken

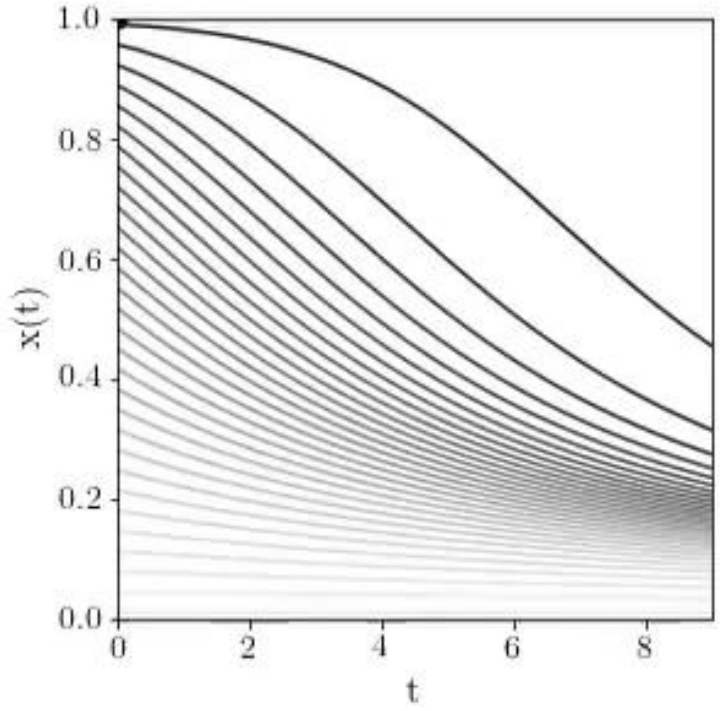


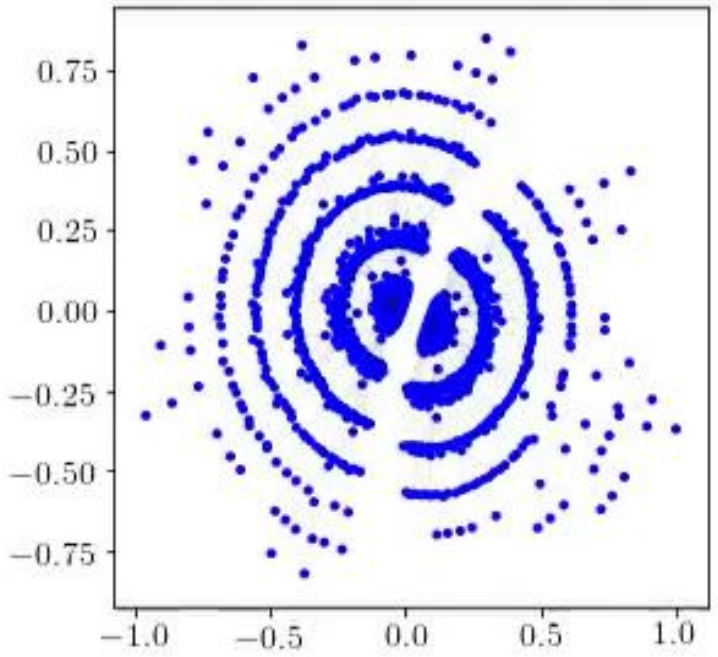






Evolutionäre Spieltheorie auf skalenfreien Netzwerken





Wir simulieren nochmals das gleiche Spiel, wobei wir nun mit einer Anfangskonfiguration starten, bei der lediglich der Spieler mit dem größten Knotengrad (der Hub des skalenfreien Netzwerkes) die rote Strategie wählt.

Die Simulation zeigt, dass sich nun die rote Strategie im Netzwerk ausbreiten kann, jedoch auch im Endzustand gewisse Spielergruppen existieren, die die blaue Strategie präferieren.

Wiederholung: Replikatordynamik

(für symmetrische (2x3)-Spiele)

Wir beschränken uns nun auf symmetrische (2x3)-Spiele , d.h. zwei Personen - 3 Strategien Spiele (M=3). Die Differentialgleichung der Replikatordynamik vereinfacht sich unter dieser Annahme wie folgt:

$$\frac{dx_j(t)}{dt} = x_j(t) \cdot \left[\sum_{k=1}^{3} \$_{jk} \cdot x_k(t) - \sum_{l=1}^{3} \sum_{k=1}^{3} \$_{kl} \cdot x_k(t) \cdot x_l(t) \right]$$

$$\frac{dx_j}{dt} = x_j \cdot \left[\begin{array}{l} \$_{j1} \cdot x_1 + \$_{j2} \cdot x_2 + \$_{j3} \cdot x_3 - \\ -\underbrace{\left(\begin{array}{l} \$_{11} \cdot x_1 \cdot x_1 + \$_{12} \cdot x_1 \cdot x_2 + \$_{13} \cdot x_1 \cdot x_3 + \\ +\$_{21} \cdot x_2 \cdot x_1 + \$_{22} \cdot x_2 \cdot x_2 + \$_{23} \cdot x_2 \cdot x_3 + \\ +\$_{31} \cdot x_3 \cdot x_1 + \$_{32} \cdot x_3 \cdot x_2 + \$_{33} \cdot x_3 \cdot x_3 \end{array} \right)}_{\overline{\$}} \end{array} \right]$$

$$j = 1,2,3$$

Replikatordynamik

(für symmetrische (2x3)-Spiele)

Man erhält ein System von drei gekoppelten Differentialgleichungen:

$$\frac{dx_1}{dt} = x_1 \cdot \left[\$_{11} \cdot x_1 + \$_{12} \cdot x_2 + \$_{13} \cdot x_3 - \bar{\$}\right]$$

$$\frac{dx_2}{dt} = x_2 \cdot \left[\$_{21} \cdot x_1 + \$_{22} \cdot x_2 + \$_{23} \cdot x_3 - \bar{\$}\right]$$

$$\frac{dx_3}{dt} = x_3 \cdot \left[\$_{31} \cdot x_1 + \$_{32} \cdot x_2 + \$_{33} \cdot x_3 - \bar{\$}\right]$$

Das System von Differentialgleichungen lässt sich bei gegebener Auszahlungsmatrix $\hat{\$}$ und Anfangsbedingung $(x_1(0), x_2(0), x_3(0))$ meist nur nummerisch (auf dem Computer) lösen. Die Lösungen bestehen dann aus den drei (zeitlich abhängigen) Populationsanteilen $(x_1(t), x_2(t), x_3(t))$.

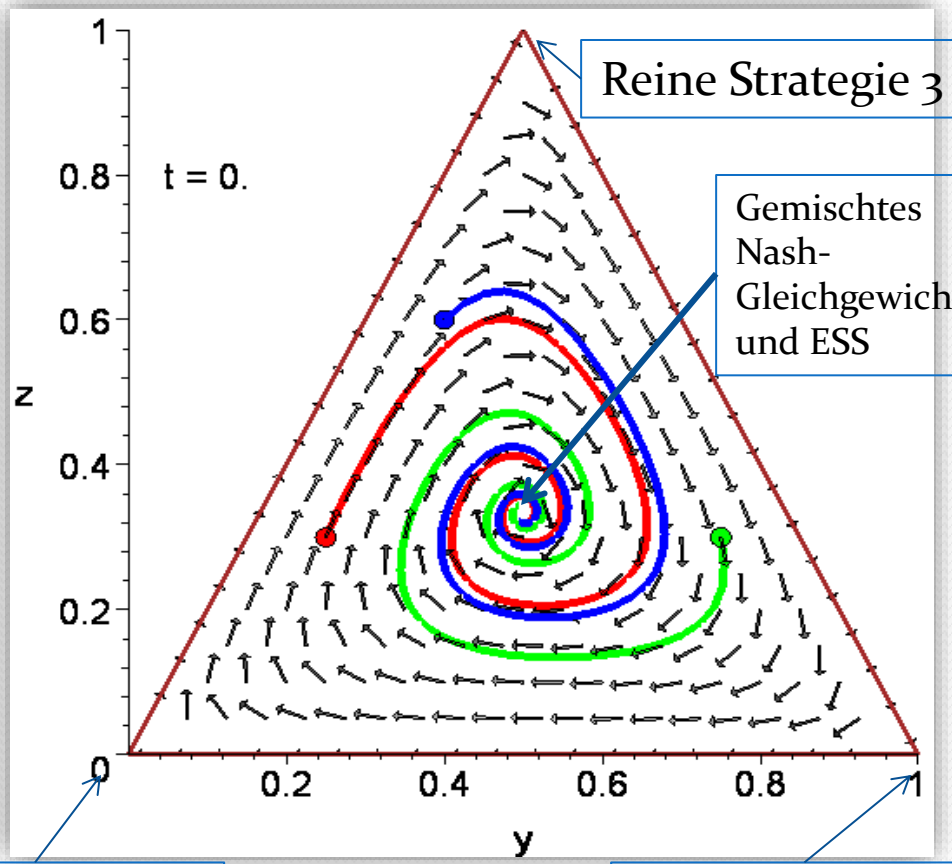
Replikatordynamik

(für symmetrische (2x3)-Spiele, **Beispiel 1**)

Wir betrachten im Folgenden ein Beispiel eines (2x3)-Spiels mit der rechts angegebenen Auszahlungsstruktur:

	Strategie 1	Strategie 2	Strategie 3
Strategie 1	(0, 0)	(2, -1)	(-1 , 2)
Strategie 2	(-1 , 2)	(0 , 0)	(2 , -1)
Strategie 3	(2 , -1)	(-1 , 2)	(0 , 0)

Die rechte Abbildung zeigt die zeitliche Entwicklung der relativen Populationsanteile der gewählten Strategien für drei mögliche Anfangsbedingungen. Die einzige evolutionär stabile Strategie dieses Beispiels befindet sich beim gemischten Nash-Gleichgewicht $\left(\frac{1}{3},\frac{1}{3},\frac{1}{3}\right)$
Die einzelnen Pfeile im Dreieck veranschaulichen den durch die Spielmatrix bestimmten Strategien-„Richtungswind“, dem die Population zeitlich folgen wird.



Zur Visualisierung der evolutionären Entwicklung benutzt man of die sogen. barycentric coordinates:

$$y := x_2 + \frac{x_3}{2}$$

$$z := x_3$$

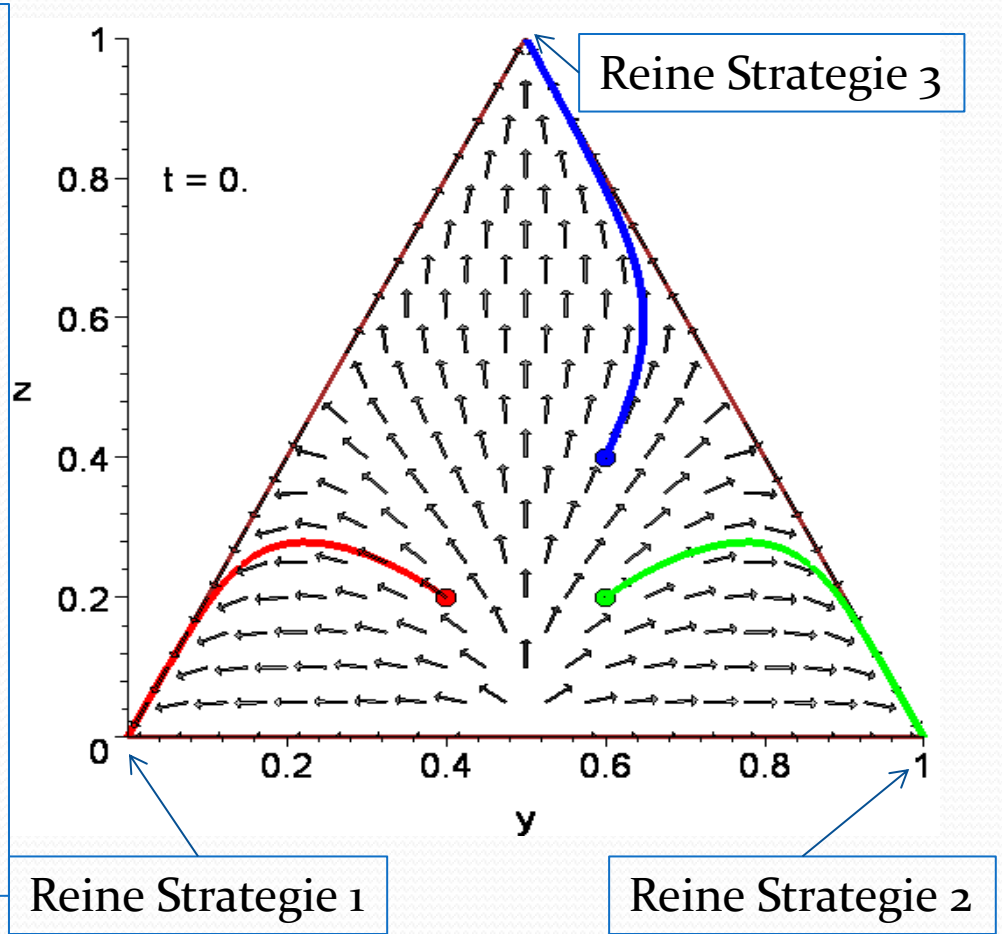
Replikatordynamik

(für symmetrische (2x3)-Spiele, **Beispiel 2**)

Wir betrachten im Folgenden ein Beispiel eines (2x3)-Spiels mit der rechts angegebenen Auszahlungsstruktur:

	Strategie 1	Strategie 2	Strategie 3
Strategie 1	(0, 0)	(-3, -3)	(-1 , -1)
Strategie 2	(-3 , -3)	(0 , 0)	(-1 , -1)
Strategie 3	(-1 , -1)	(-1 , -1)	(0 , 0)

Die rechte Abbildung zeigt die zeitliche Entwicklung der relativen Populationsanteile der gewählten Strategien für drei mögliche Anfangsbedingungen. Das Spiel besitzt drei Nash-Gleichgewichte in reinen Strategien, die ebenfalls evolutionär stabile Strategien darstellen. Welche der drei ESS die Population realisiert hängt von dem Anfangswert der Populationsanteile ab. Die zeitliche Entwicklung folgt wieder dem Strategien-„Richtungswind“ der zugrundeliegenden Auszahlungsmatrix.

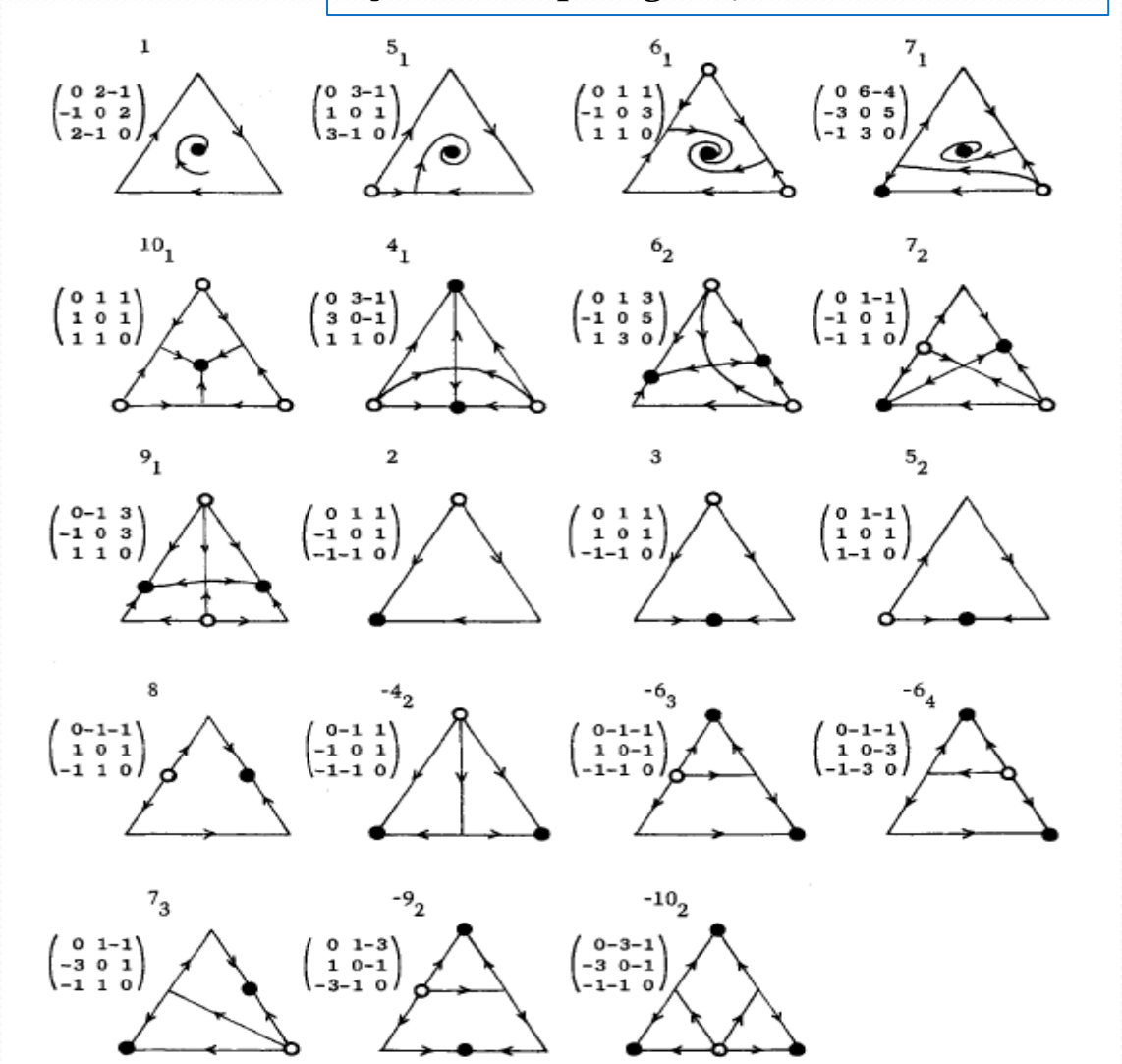


Replikatordynamik

(Klassifizierung symmetrische (2x3)-Spiele)

E. C. Zeeman, *POPULATION DYNAMICS FROM GAME THEORY, In:* Global Theory of Dynamical Systems, Springer 1980

E. C. Zeeman zeigt in seinem im Jahre 1980 veröffentlichten Artikel, dass man evolutionäre, symmetrische (2x3)-Spiele in 19 Klassen einteilen kann. Die Abbildung rechts zeigt das evolutionäre Verhalten dieser 19 Spieltypen. Die ausgefüllten schwarzen Punkte markieren die evolutionär stabilen Strategien der jeweiligen Spiele. Es gibt Spielklassen, die besitzen lediglich eine ESS und Klassen die sogar drei ESS besitzen.



Jupyter Notebook
Evolutionspiel4.ipynb

Physik der sozio-ökonomischen Systeme mit dem Computer

(Physics of Socio-Economic Systems with the Computer)

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Wintersemester 2025/26)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 22.08.2025

Erster Vorlesungsteil:

Die 19 Klassen der evolutionären symmetrischen (2×3)-Spiele

Einführung

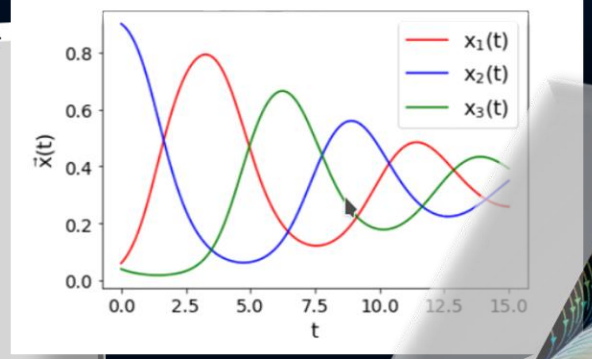
In diesem Unterkapitel werden die evolutionären symmetrischen (2×3)-Spiele analysiert. Symmetrische $(2 \times m)$-Spiele werden durch die folgende Differenzialgleichung beschrieben:

$$\frac{d\vec{x}}{dt} = \vec{x}\left(\hat{\$}\vec{x}\right) - \left(\left(\hat{\$}\vec{x}\right)^T \vec{x}\right)\vec{x}$$

Aufgrund der Symmetrie der Auszahlungsmatrix $\hat{\$} = \hat{\$}^A = \left(\hat{\$}^B\right)^T$ kann die zeitliche Entwicklung solcher Spiele durch nur einen Populationsvektor ($\vec{x}(t) = (x_1(t), x_2(t), \ldots, x_m(t))$ beschrieben werden.

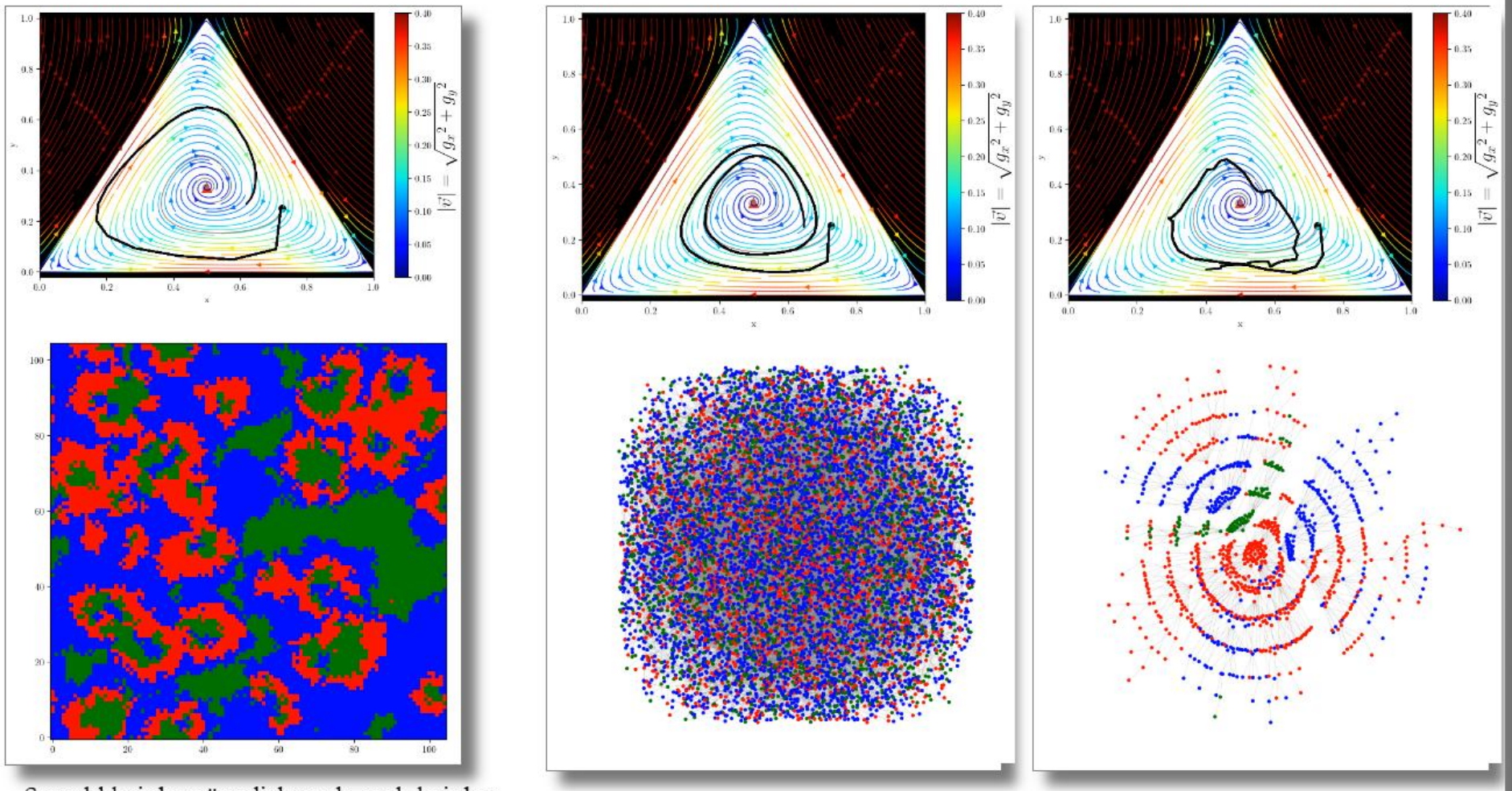
Wir beschränken uns im Folgenden auf den Fall $m = 3$ und setzen die Auszahlungsmatrix wie folgt an:

$$\hat{\$} = \begin{pmatrix} \$_{11} & \$_{12} & \$_{13} \\ \$_{21} & \$_{22} & \$_{23} \\ \$_{31} & \$_{32} & \$_{33} \end{pmatrix}$$

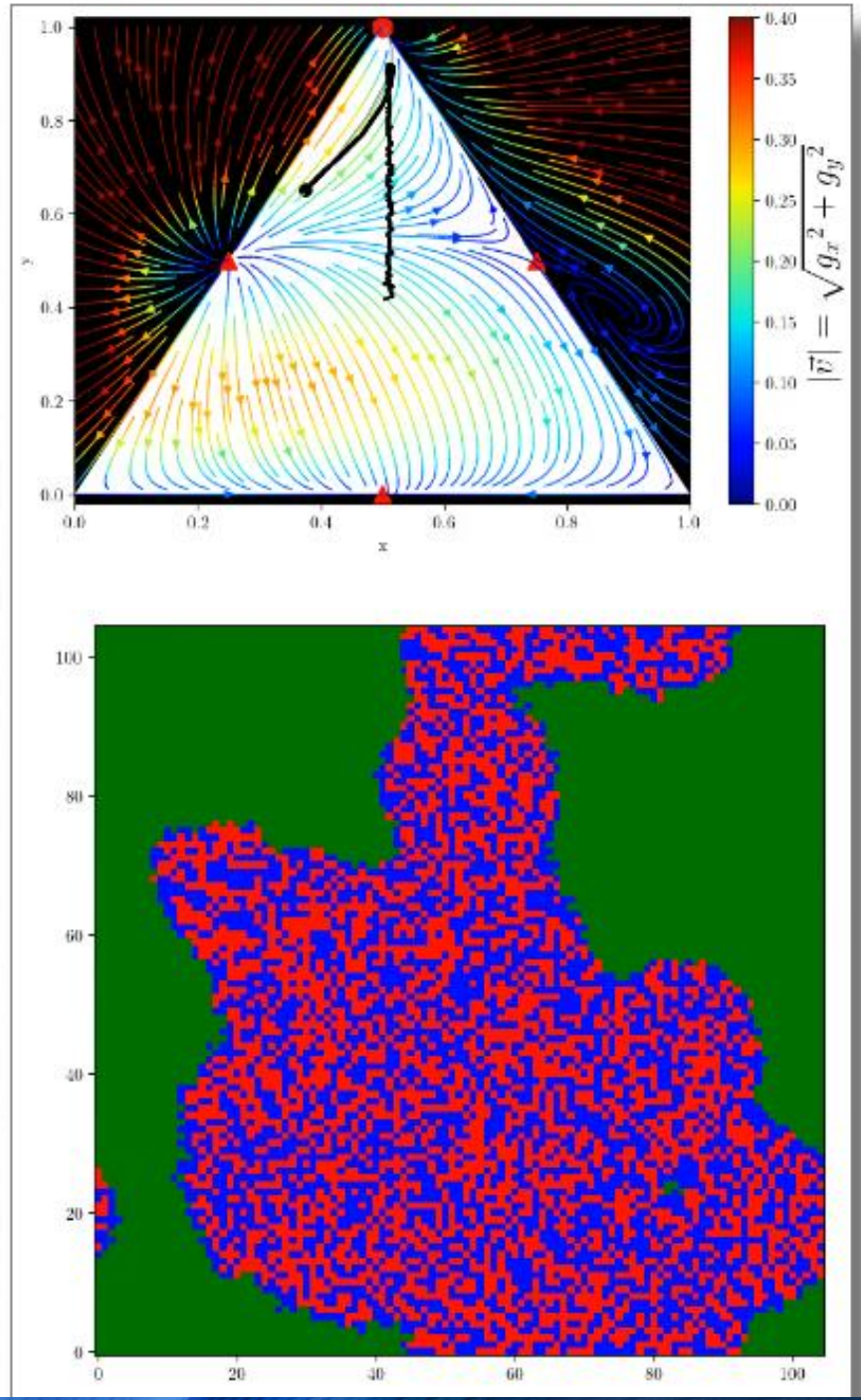


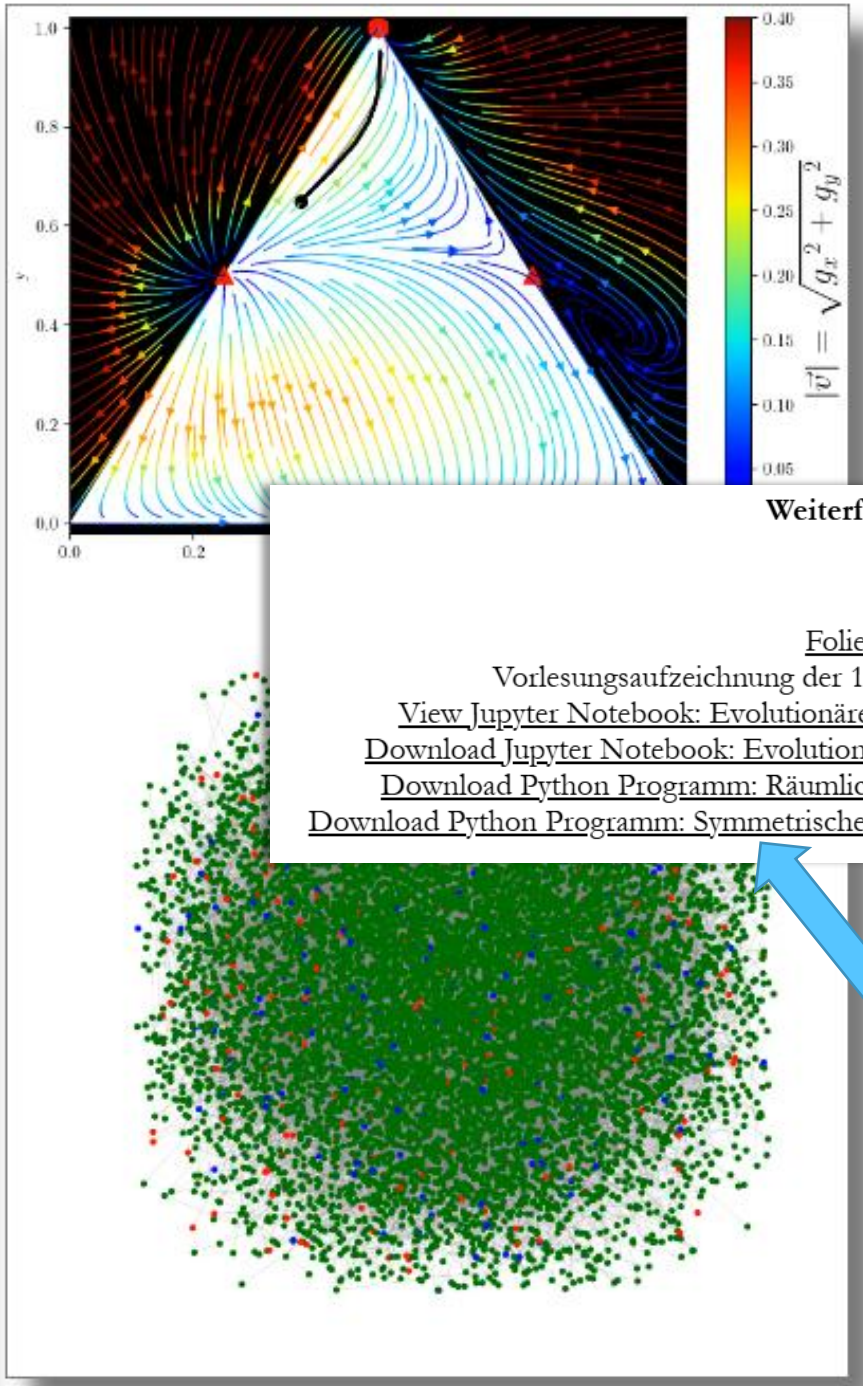
Evolutionäre symmetrische (2×3) Spiele auf unterschiedlichen Netzwerk-Topologien

In den Python Programmen SpatialGame_2x3.py und NetworkGame_2x3.py werden evolutionäre, symmetrische $(\mathbf{2} \times \mathbf{3})$ Spiele auf räumlichen Gitterstrukturen und anderen Netzwerk-Topologien simuliert. Da die deterministische Replikatordynamik symmetrischer $(\mathbf{2} \times \mathbf{3})$ Spiele bereits 19 unterschiedliche Spielklassen erlaubt (siehe Vorlesung 5), gibt es, mittels der stochastischen Simulationen, eine große Zahl von möglichen zeitlichen Entwicklungen der Population. Die unten dargestellten Animationen stellen drei Simulationen der Zeeman-Klasse 1 dar, wobei bei der linken Abbildung eine räumliche Gitterstruktur zugrunde liegt, die mittlere Animation ein zufälliges Netzwerk und die rechte Simulation ein skalenfreies Netzwerk verwenden.



Sowohl bei den räumlichen als auch bei den Netzwerk-Simulationen zeigt sich oft eine qualitative Übereinstimmung mit den Vorhersagen der deterministischen evolutionären Spieltheorie.





Weiterführende Links

Folien der 10.Vorlesung

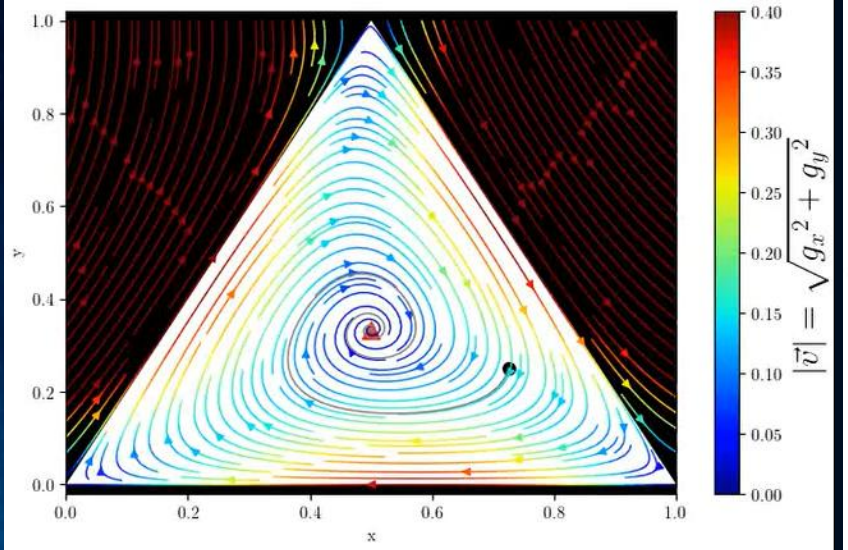
Vorlesungsaufzeichnung der 10.Vorlesung: WS 2022/23 bzw. WS 2021/22

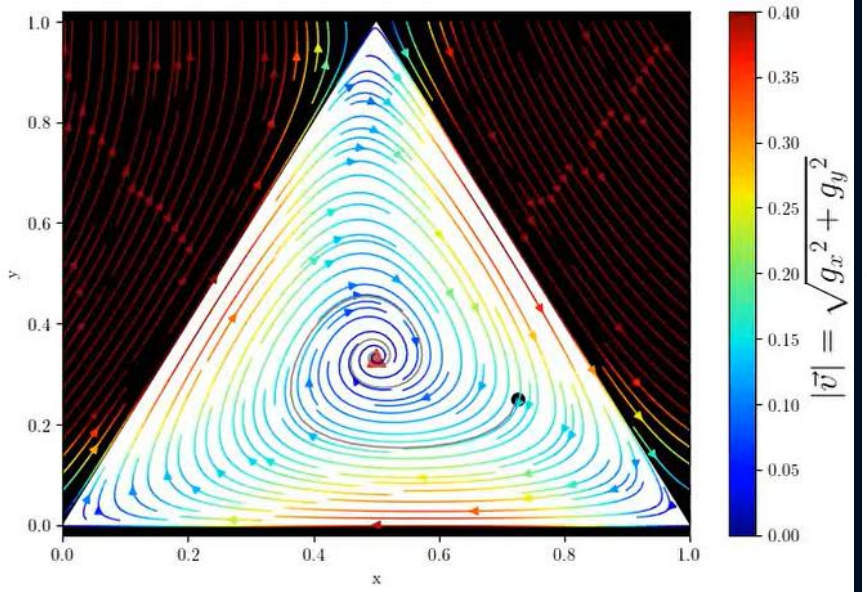
View Jupyter Notebook: Evolutionäre Spiele auf unterschiedlichen Netzwerk-Topologien

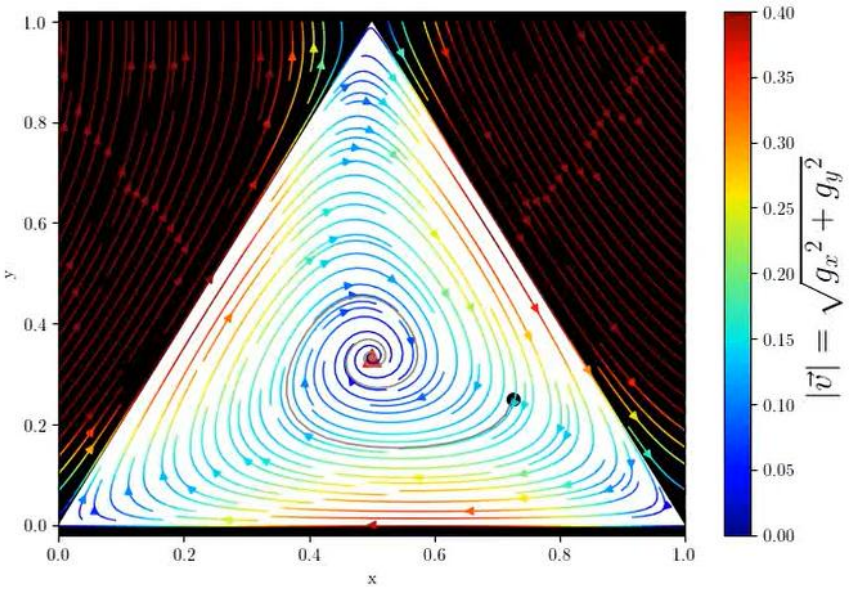
Download Jupyter Notebook: Evolutionäre Spiele auf unterschiedlichen Netzwerk-Topologien

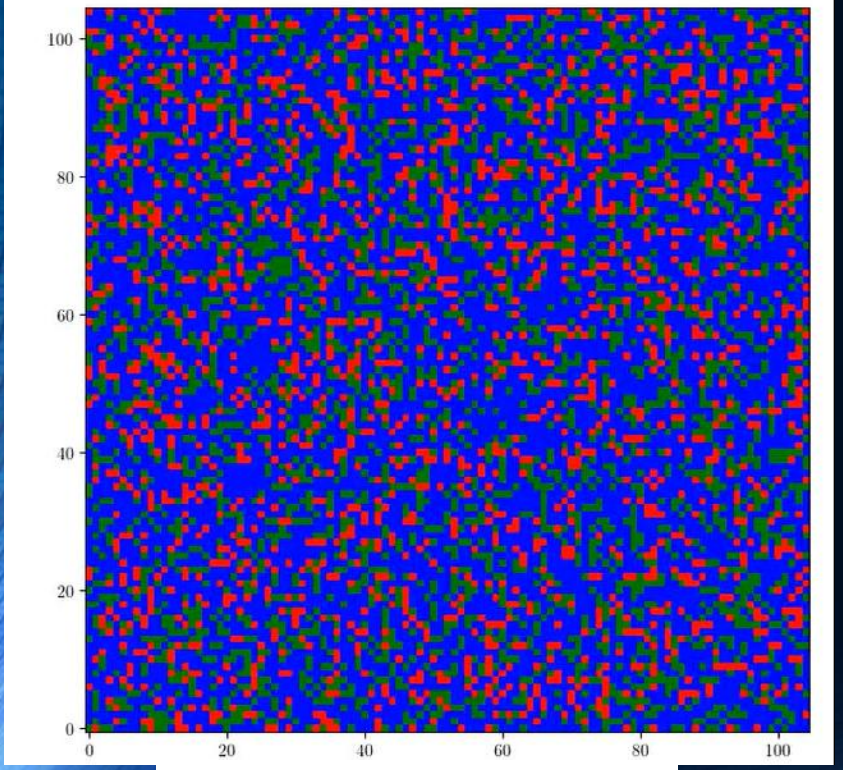
Download Python Programm: Räumliches symmetrisches (2×3)-Spiel: SpatialGame_2x3.py

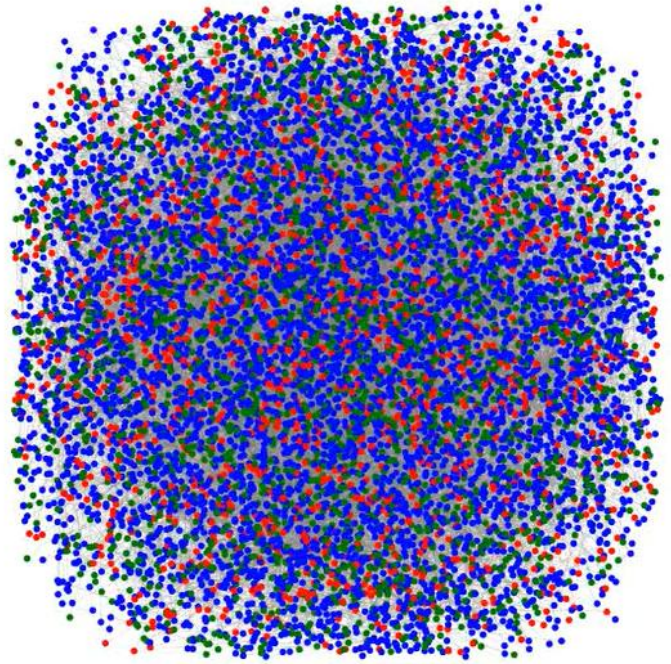
Download Python Programm: Symmetrisches (2×3)-Spiel auf einem Netzwerk: NetworkGame_2x3.py

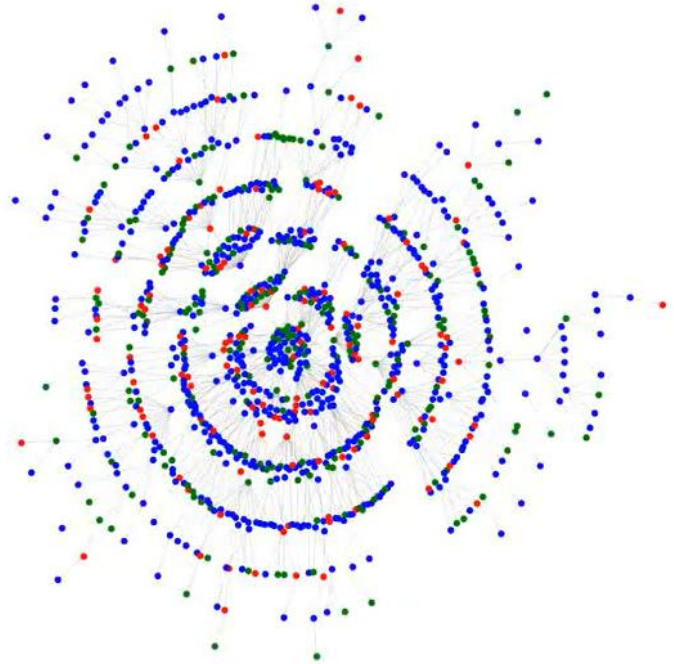








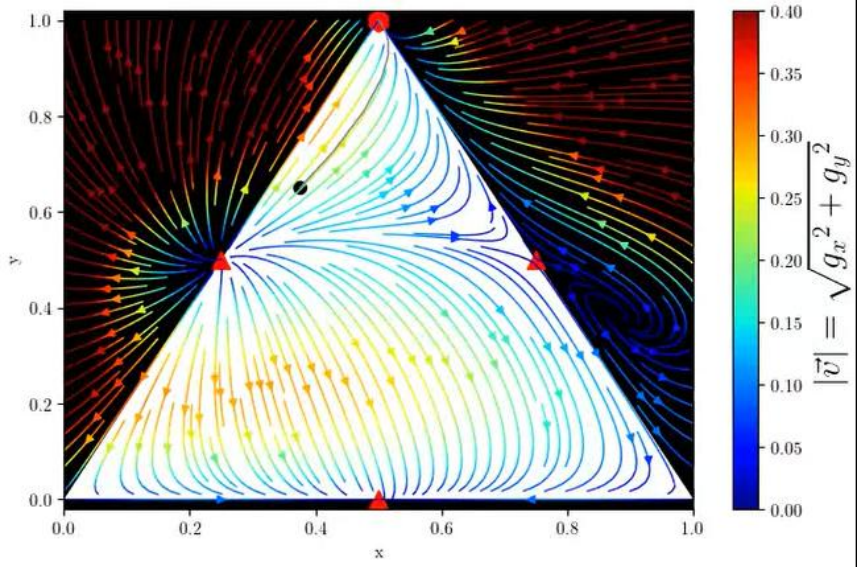


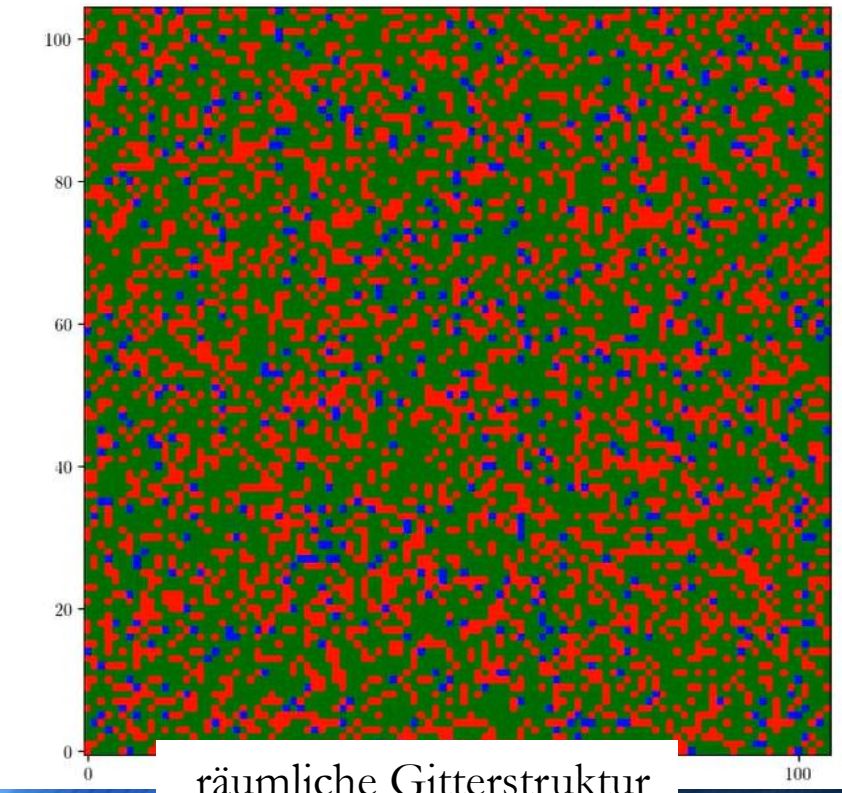


räumliche Gitterstruktur

zufälliges Netzwerk

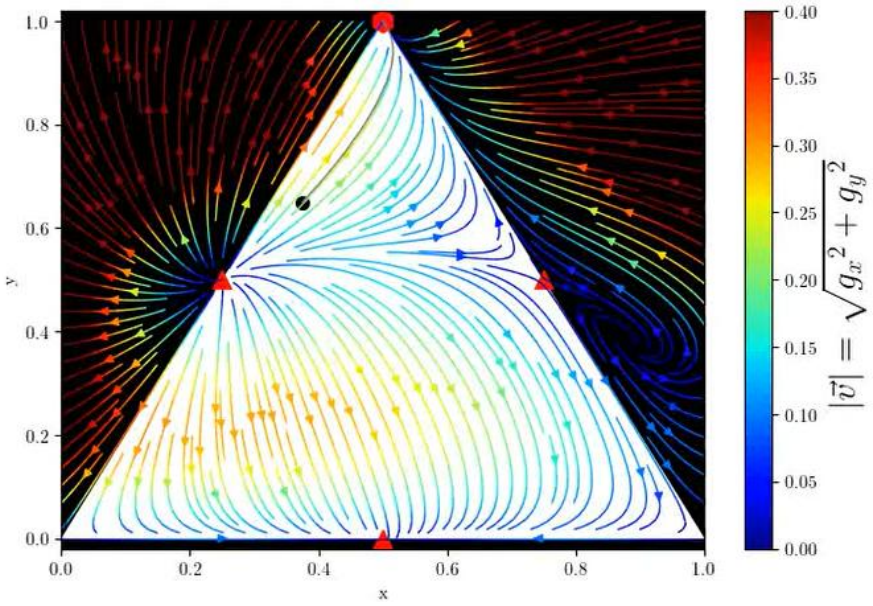
skalenfreies Netzwerk

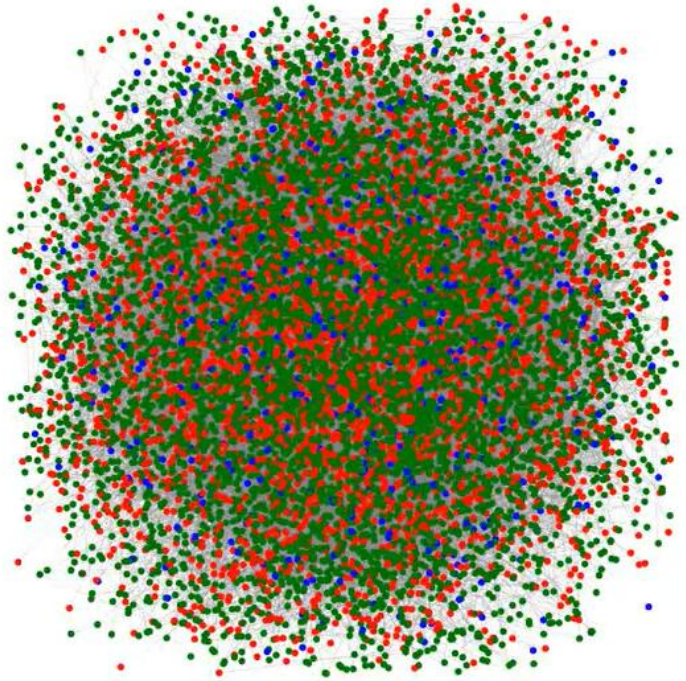




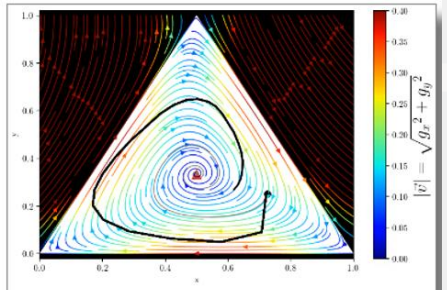
räumliche Gitterstruktur

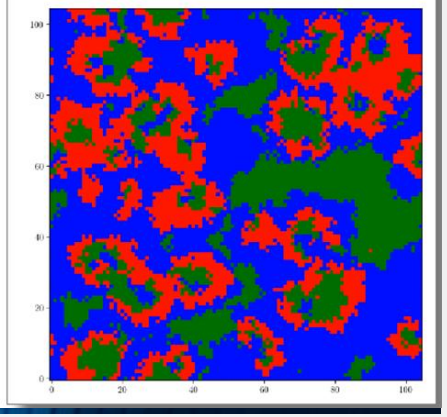
Die rechte Simulation ist in guter Übereinstimmung mit den Vorhersagen der deterministischen evolutionären Spieltheorie. Die simulierte Populations-entwicklung folgt relativ genau dem Populationswind der deterministischen Replikatordynamik und endet schließlich in dem reinen Nash-Gleichgewicht der oberen Ecke des baryzentrischen Dreiecks, bei dem die gesamte Population nur die grüne Strategie wählt.
Im Gegensatz dazu entwickelt sich die linke räumliche Simulation nach einiger Zeit entgegen den Vorhersagen der deterministischen Spieltheorie und folgt nicht mehr dem Populationswind der Replikatordynamik. Regionen, bestehend aus ausschließlich roten und blauen Strategien, dehnen sich langsam über die gesamte Population aus und schließlich endet die Population in dem dynamischen Zustand des gemischten Nash-Gleichgewichtes am mittleren unteren Rand des baryzentrischen Dreiecks.





zufälliges Netzwerk



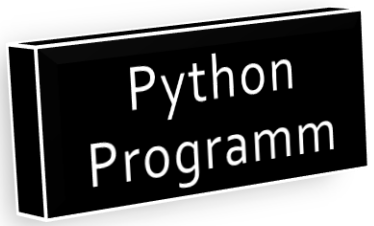


```python
# Erstellen eines raumlichen 2D-Gittergraph (8 Nachbarn)
def create_grid(width, height):
    g = nx.Graph()
    nkn = width * height
    for i in range(width):
        for j in range(height):
            k = i * height + j
            g.add_node(k)
    for i in range(width):
        for j in range(height):
            center = i * height + j
            neighbors = [
                ((i-1) % width) * height + (j-1) % height,  #
                (i % width) * height + (j-1) % height,      #
                ((i+1) % width) * height + (j-1) % height,  #
                ((i+1) % width) * height + (j % height),    #
                ((i+1) % width) * height + (j+1) % height,  #
                (i % width) * height + (j+1) % height,      #
                ((i-1) % width) * height + (j+1) % height,  #
                ((i-1) % width) * height + (j % height)     #
            ]
            for neigh in neighbors:
                if neigh != center:  # Vermeide Selbst-Edges
                    g.add_edge(center, neigh)
    return g
```

SpatialGame_2x3.py

home > hanauske > neu_2025 > VPSOC_2025 > Vorlesungen > 11 > SpatialGame_2x3.py

```python
###################################
# Python-Programm "Spatial (2x3)-Games"
###################################

import networkx as nx
import matplotlib.pyplot as plt
from random import randint, uniform
from math import isclose
import numpy as np
from matplotlib import rcParams
import matplotlib.gridspec as gridspec
from scipy.integrate import solve_ivp
import os
from sympy import symbols, Matrix, Eq, transpose, solve
import matplotlib.colors as colors

# baryzentrisches Dreiecks-Koordinatensystem
def xy(vx):
    return [vx[1]+vx[2]/2,vx[2]]
```




```python
# Funktion zur Berechnung der Nashgleichgewichte
def find_nash_equilibria(D):
    equilibria = []
    # Berechnung der reinen Nash-Gleichgewichte
    for i in range(D.shape[0]):
        for j in range(D.shape[0]):
            if D[i, j] == max(D[:, j]):
                if D[j, i] == max(D[:, i]):
                    equilibria.append({'type': 'pure', 's': (i+1,j+1)})
    #Berechnung der gemischten Nash-Gleichgewichte
    Loes_GN = []
    x1,x2,x3,y1,y2,y3 = symbols('x_1,x_2,x_3,y_1,y_2,y_3')
    xs = Matrix([x1,x2,x3])
    ys = Matrix([y1,y2,y3])
    Dollar_A = transpose(xs)*D*ys
    Dollar_As = Dollar_A.subs(x3,1-x1-x2).subs(y3,1-y1-y2)[0]
    Dollar_As_1 = Dollar_A.subs(x1,0).subs(x3,1-x2).subs(y3,1-y1-y2)[0]
    Dollar_As_2 = Dollar_A.subs(x2,0).subs(x3,1-x1).subs(y3,1-y1-y2)[0]
    Dollar_As_3 = Dollar_A.subs(x3,0).subs(x2,1-x1).subs(y3,1-y1-y2)[0]
    GemNash_Eq1 = Eq(Dollar_As.diff(x1),0)
    GemNash_Eq2 = Eq(Dollar_As.diff(x2),0)
    GemNash_Eq_1 = Eq(Dollar_As_1.diff(x2),0)
    GemNash_Eq_2 = Eq(Dollar_As_2.diff(x1),0)
    GemNash_Eq_3 = Eq(Dollar_As_3.diff(x1),0)
    Bed=Eq(1,y1+y2+y3)
    Loes_GN.append(solve([GemNash_Eq1,GemNash_Eq2,Bed]))
    Bed_a=Eq(0,y1)
    Bed_b=Eq(1,y2+y3)
    Loes_GN.append(solve([GemNash_Eq_1,Bed_a,Bed_b]))
    Bed_a=Eq(0,y2)
    Bed_b=Eq(1,y1+y3)
    Loes_GN.append(solve([GemNash_Eq_2,Bed_a,Bed_b]))
    Bed_a=Eq(0,y3)
    Bed_b=Eq(1,y1+y2)
    Loes_GN.append(solve([GemNash_Eq_3,Bed_a,Bed_b]))
    for l in Loes_GN:
        if l and 0 <= l[y1] < 1 and 0 <= l[y2] < 1 and 0 <= l[y3] < 1 :
            equilibria.append({'type': 'mixed', 's*': (l[y1],l[y2],l[y3])})
    return equilibria
```

```python
# Initialisiert Spieler-Array P mit Positionen und Strategien
def initialize_players(width, height, x_init):
    nkn = width * height
    p = np.zeros((nkn, 6))  # Spalten-Inhalt: [id, x, y, current_strat, payoff, next_strat]
    k = 0
    for i in range(width):
        for j in range(height):
            p[k, 0] = k
            p[k, 1] = i
            p[k, 2] = j
            zufall = uniform(0, 1)
            if zufall <= x_init[0]:
                p[k, 3] = 0 #->Strategie s_1
            elif zufall > (x_init[0]+x_init[1]):
                p[k, 3] = 2 #->Strategie s_3
            else:
                p[k, 3] = 1 #->Strategie s_2
            p[k, 5] = p[k, 3]
            k += 1
    return p

# Berechnet Auszahlungen für alle Spieler
def compute_payoffs(g, p, D):
    for u, v in g.edges():
        p[u, 4] += D[int(p[u,3]),int(p[v,3])]
        p[v, 4] += D[int(p[v,3]),int(p[u,3])]
```

```python
# Neue Strategien-Wahl
def update_strategy(p, g, rule):
    nkn = len(p)
    for k in range(nkn):
        neighbors = list(g.neighbors(k))
        if not neighbors:
            continue
        # Rule 0: Imitiere besten Nachbarn
        if rule == 0:
            neigh_payoffs = p[neighbors, 4]
            max_payoff = np.max(neigh_payoffs)
            if max_payoff > p[k, 4] and not isclose(max_payoff, p[k, 4]):
                best_idx = np.argmax(neigh_payoffs)
                p[k, 5] = p[neighbors[best_idx], 3]
        # Rule 1: Imitiere zufälligen besseren Nachbarn
        elif rule == 1:
            j = randint(0, len(neighbors) - 1)
            neigh_payoff = p[neighbors[j], 4]
            if neigh_payoff > p[k, 4] and not isclose(neigh_payoff, p[k, 4]):
                p[k, 5] = p[neighbors[j], 3]
        else:
            raise ValueError(f"Ungültige Regel: {rule}. Verwende 0 oder 1.")

# Wendet zukünftige Strategien an, berechnet Mittelwert des Populationsvektors und mittleren Payoffs und setzt Payoffs zurück
def apply_updates(p):
    nkn = len(p)
    mean_payoff = np.mean(p[:, 4]) / 8
    mean_strat_3 = [np.count_nonzero(p[:,3] == 0)/nkn, np.count_nonzero(p[:,3] == 1)/nkn, np.count_nonzero(p[:,3] == 2)/nkn]
    p[:, 4] = 0
    p[:, 3] = p[:, 5]
    return xy(mean_strat_3),  mean_payoff

# Definition der Funktionen g_x und g_y
def g_xy(xy,D):
    m = 3
    x = np.array([1-xy[0]-xy[1]/2,xy[0]-xy[1]/2,xy[1]])
    dx_dt = []
    for i in range(m):
        dx_dt.append(sum(D[i,j]*x[i]*x[j] for j in range(m)) - sum(sum(D[k,j]*x[k]*x[j] for j in range(m)) for k in range(m))*x[i])
    return [dx_dt[1]+dx_dt[2]/2,dx_dt[2]]
```

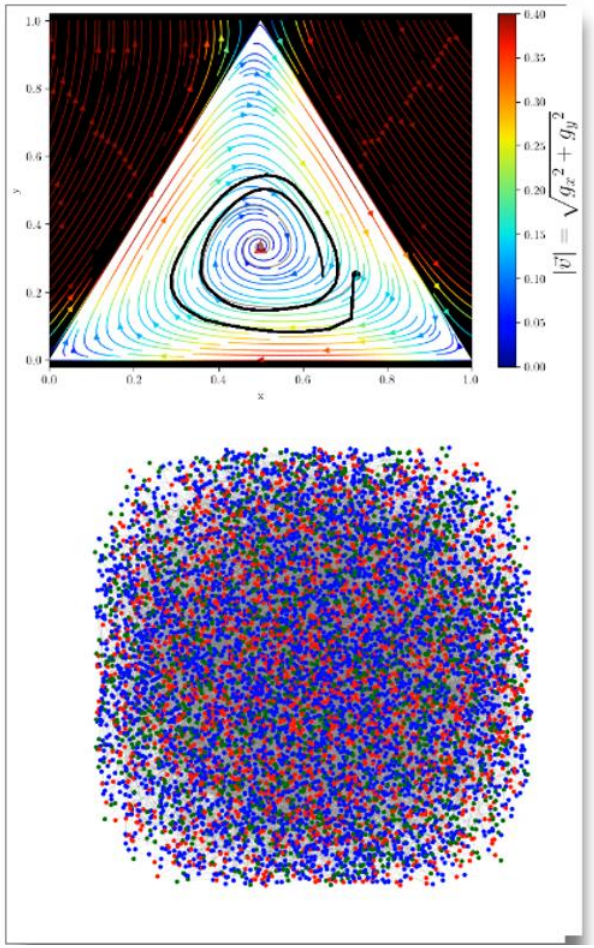
```python
# Führt die Simulation auf dem 2D-Gitter aus
def run_simulation(width=105, height=105, nit=15, rule=1, D = np.array([[0,-1,-1],[1,0,-3],[-1,-3,0]]), x_init=[0.25,0.5,0.25], output_dir="./pics"):
    os.makedirs(output_dir, exist_ok=True)

    # Netzwerk und Spieler initialisieren
    g = create_grid(width, height)
    p = initialize_players(width, height, x_init)
    av_strat_x = [xy(x_init)[0]]
    av_strat_y = [xy(x_init)[1]]
    av_dollar = []

    # Analytische Lösung
    t_span = [0, nit]
    t_ana, x_ana = analytical_solution(D, t_span, x_init)

    # Plot-Setup
    rcParams.update({
        'figure.figsize': [8,14],
        'text.usetex': True,
        'legend.fontsize': 12
    })
    plt.figure(0)
    gs = gridspec.GridSpec(2, 1, height_ratios=[1, 1.5], hspace=0.2)
    ax1 = plt.subplot(gs[0])
    ax2 = plt.subplot(gs[1])

    # Fuer die Darstellung des Feldliniendiagramms streamplot
    Y, X = np.mgrid[0:1:100j, 0:1:100j]
    gXY = g_xy([X,Y],D)
    # Die Farbe wird die Geschwindigkeit der Aenderung des Populationsvektors anze
    colorspeed = np.sqrt(gXY[0]**2 + gXY[1]**2)

    ax1.set_xlabel(r"$\rm x$")
    ax1.set_ylabel(r"$\rm y$")
    figure = ax1.streamplot(X, Y, gXY[0], gXY[1], linewidth=1,density=[2, 2],norm=colors.Normalize(vmin=0.,vmax=0.4), color=colorspeed, cmap=plt.cm.jet)
    ax1.fill([0,0.5,0], [0,1,1], facecolor='black')
    ax1.fill([1,0.5,1], [0,1,1], facecolor='black')
    ax1.fill([0,1,1,0], [0,0,-0.02,-0.02], facecolor='black')
    ax1.fill([0,1,1,0], [1,1,1.02,1.02], facecolor='black')
    ax1.scatter(xy(x_init)[0],xy(x_init)[1], s=50, marker='o', c="black")
    ax1.plot(xy(x_ana[0:3])[0], xy(x_ana[0:3])[1],c="grey",linewidth=1)
    # Erzeugung der nebenstehenden Farblegende colorbar
    cbar=plt.colorbar(figure.lines, aspect=20, ax=ax1)
    cbar.set_label(r'$\left| \vec{v} \right| =\sqrt{{g_x}^2 + {g_y}^2}$',size=20)
    ax1.set_xlim(0, 1)
    ax1.set_ylim(-0.02, 1.02)
```

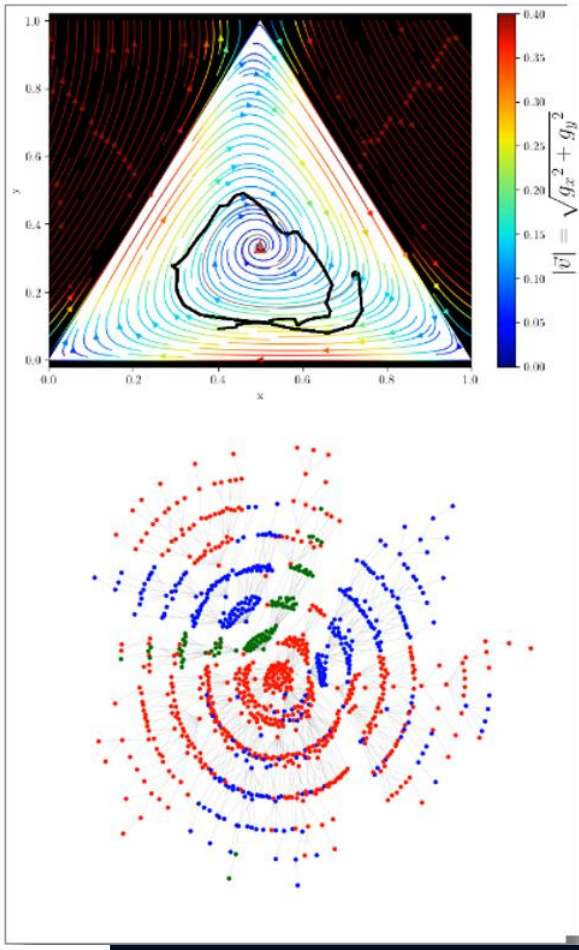
```python
# Analytische Lösung der Replikator-DGL
def analytical_solution(D, t_span, x_init, num_points=500):
    # Definition des DGL-Systems
    def DGLsys(t, x):
        x = np.asarray(x)
        Dx = D @ x
        u = np.dot(x, Dx)
        return x * (Dx - u)

    t_eval = np.linspace(t_span[0], t_span[1], num_points)
    sol = solve_ivp(DGLsys, t_span, x_init, t_eval=t_eval, rtol=10**(-13), atol=10**(-13))
    return sol.t, sol.y

# Plotted den aktuellen Zustand der Simulation
def plot_simulation(ax1, ax2, av_strat, p, width, height):
    nkn = len(p)
    sgross = np.sqrt( 2822400 / nkn ) # Groesse der Spieler Kaestchen
    col = ['r' if s == 0 else 'b' if s == 1 else 'g' for s in p[:, 3]]
    col_new = ['r' if s == 0 else 'b' if s == 1 else 'g' for s in p[:, 5]]
    alpha = [1 if p[k, 3] == p[k, 5] else 0.5 for k in range(nkn)]

    #Mittelwert des Populationsvektors
    ax1.plot(av_strat[0], av_strat[1],c="black",linewidth=2)

    # Grid-Plot
    x_pos, y_pos = p[:, 1], p[:, 2]
    #ax2.scatter(x_pos, y_pos, s=sgross, c=col, marker="s", alpha=alpha, edgecolor='none')
    ax2.scatter(x_pos, y_pos, s=sgross, c=col, marker="s", edgecolor='none')

    ax2.set_xlim(-0.5, width - 0.5)
    ax2.set_ylim(-0.5, height - 0.5)
    ax2.set_aspect('equal', adjustable='box')
```

```
    # Berechnung der Nashgleichgewichte
    equilibria = find_nash_equilibria(D)
    print("Nash-Gleichgewichte:")
    for eq in equilibria:
        if eq['type'] == 'pure':
            print(f"Reines: Spieler 1 Strategie {eq["s"][0]}, Spieler 2 Strategie {eq["s"][1]}")
            # Kennzeichnung der reinen Nashgleichgewichte im Bild
            if eq["s"][0] == 1 and eq["s"][1] == 1:
                ax1.scatter(0,0, s=150, marker='h', c="red")
            if eq["s"][0] == 2 and eq["s"][1] == 2:
                ax1.scatter(1,0, s=150, marker='h', c="red")
            if eq["s"][0] == 3 and eq["s"][1] == 3:
                ax1.scatter(0.5,1, s=150, marker='h', c="red")
        else:
            print(f"Gemischtes: s*={eq["s*"]}")
            ax1.scatter(xy(eq["s*"])[0],xy(eq["s*"])[1], s=100, marker='^', c="red")

    # Simulation-Schleife
    for it in range(0, nit):
        print(f"Iteration {it} ------------------------------------------------------")
        compute_payoffs(g, p, D)
        update_strategy(p, g, rule)
        plot_simulation(ax1, ax2, [av_strat_x,av_strat_y], p, width, height)
        mean_strat, mean_dollar = apply_updates(p)
        av_strat_x.append(mean_strat[0])
        av_strat_y.append(mean_strat[1])
        av_dollar.append(mean_dollar)

        # Speichern (PNG und PDF)
        filename = f"img-{'{:0>3d}'.format(it)}"
        plt.savefig(os.path.join(output_dir, f"{filename}.png"), dpi=100, bbox_inches="tight", pad_inches=0.05)
#        plt.savefig(os.path.join(output_dir, f"{filename}.pdf"), bbox_inches="tight", pad_inches=0.05)
        ax2.clear()

    plt.close()
    print("Mittelwerte der Auszahlungen: ", [round(x, 3) for x in av_dollar])
    print("Simulation abgeschlossen. Plots in", output_dir, "gespeichert.")
```

```
# Auszahlungsmatrix des symmetrischen (2x3)-Spiel (19 Zeeman Klassen)
# Zeeman_Klasse_1: 0,2,-1,-1,0,2,2,-1,0
# Zeeman_Klasse_2: 0,3,-1,1,0,1,3,-1,0
# Zeeman_Klasse_3: 0,1,1,-1,0,3,1,1,0
# Zeeman_Klasse_4: 0,6,-4,-3,0,5,-1,3,0
# Zeeman_Klasse_5: 0,1,1,1,0,1,1,1,0
# Zeeman_Klasse_6: 0,3,-1,3,0,-1,1,1,0
# Zeeman_Klasse_7: 0,1,3,-1,0,5,1,3,0
# Zeeman_Klasse_8: 0,1,-1,-1,0,1,-1,1,0
# Zeeman_Klasse_9: 0,-1,3,-1,0,3,1,1,0
# Zeeman_Klasse_10: 0,1,1,-1,0,1,-1,-1,0
# Zeeman_Klasse_11: 0,1,1,1,0,1,-1,-1,0
# Zeeman_Klasse_12: 0,1,-1,1,0,1,1,-1,0
# Zeeman_Klasse_13: 0,-1,-1,1,0,1,-1,1,0
# Zeeman_Klasse_14: 0,-1,1,-1,0,1,-1,-1,0
# Zeeman_Klasse_15: 0,-1,-1,1,0,-1,-1,-1,0
# Zeeman_Klasse_16: 0,-1,-1,1,0,-3,-1,-3,0
# Zeeman_Klasse_17: 0,1,-1,-3,0,1,-1,1,0
# Zeeman_Klasse_18: 0,1,-3,1,0,-1,-3,-1,0
# Zeeman_Klasse_19: 0,-3,-1,-3,0,-1,-1,-1,0

# Festlegung der Simulationsparameter
D11,D12,D13,D21,D22,D23,D31,D32,D33 = 0,2,-1,-1,0,2,2,-1,0
set_D = np.array([[D11,D12,D13],[D21,D22,D23],[D31,D32,D33]])
run_simulation(105, 105, 80, 1, set_D, [0.15,0.6,0.25],"./pics_1")
```

```python
# Erstellen eines zufälligen Graphen
def create_random_graph(nkn, prob):
    g = nx.erdos_renyi_graph(nkn, prob)
    i = 0
    while i < nkn:
        if g.degree(i) == 0:
            Kn = randint(0, nkn-1)
            if Kn != i:
                g.add_edge(i,Kn)
                i += 1
        else:
            i += 1
    return g

# Erstellen des Netzwerks
def create_network(net, nkn):
    if net == "random":
        g = create_random_graph(nkn, 2*nkn/nkn**2)
    elif net == "scale_free":
        g = nx.barabasi_albert_graph(nkn, 1)
    elif net == "small_world":
        g = nx.watts_strogatz_graph(nkn, 4, 0.01)
    else:
        raise ValueError(f"Ungültiges Netzwerk: {rule}. Verwende random, scale_free oder small_world")

    if net == "scale_free" and nkn <= 1000:
        pos = nx.kamada_kawai_layout(g)
    elif net == "small_world" and nkn <= 1000:
        pos = nx.circular_layout(g)
    else:
        pos_random = nx.random_layout(g)
        pos = nx.spring_layout(g, pos=pos_random, k=0.04, iterations=5)

    return g, pos
```





```python
# Plotted den aktuellen Zustand der Simulation
def plot_simulation(ax1, ax2, av_strat, p, g, pos):
    nkn = len(p)
    sgross = 15
    col = ['r' if s == 0 else 'b' if s == 1 else 'g' for s in p[:, 3]]
    col_new = ['r' if s == 0 else 'b' if s == 1 else 'g' for s in p[:, 5]]
    alpha = [1 if p[k, 3] == p[k, 5] else 0.5 for k in range(nkn)]

    #Mittelwert des Populationsvektors
    ax1.plot(av_strat[0], av_strat[1],c="black",linewidth=2)

    # Netzwerk-Plot
    #nx.draw_networkx_nodes(g, pos, node_size=sgross, node_color=col, alpha=alpha, edgecolors="none")
    nx.draw_networkx_nodes(g, pos, node_size=sgross, node_color=col, edgecolors="none")
    nx.draw_networkx_edges(g, pos, alpha=0.3, width=0.4, edge_color="grey")
    ax2.axis("off")

# Führt die Simulation auf dem 2D-Gitter aus
def run_simulation(net="small_world", nkn=500, nit=80, rule=1, D = np.array([[0,2,-1],[-1,0,2],[2,-1,0]]), x_init=[0.15,0.6,0.25], output_dir="./pics"):
    os.makedirs(output_dir, exist_ok=True)

    # Netzwerk und Spieler initialisieren
    g, pos = create_network(net, nkn)
    p = initialize_players(nkn, x_init)
    av_strat_x = [xy(x_init)[0]]
    av_strat_y = [xy(x_init)[1]]
    av_dollar = []
```

```python
# Festlegung der Simulationsparameter
D11,D12,D13,D21,D22,D23,D31,D32,D33 = 0,2,-1,-1,0,2,2,-1,0
set_D = np.array([[D11,D12,D13],[D21,D22,D23],[D31,D32,D33]])
run_simulation("random", 10000, 120, 1, set_D, [0.15,0.6,0.25],"./pics_1_random")
#run_simulation("scale_free", 1000, 80, 1, set_D, [0.15,0.6,0.25],"./pics_1_scale_free")
```

Einführung in die Objekt-orientierte Programmierung

Die meisten Programmiertechniken, die wir bis jetzt kennengelernt haben, verwendeten den Programmentwurfstil der *prozeduralen Programmierung* und wir benutzten meist die Programmiersprache Python bzw. verwendeten Python Jupyter Notebooks. Wir werden nun einerseits den Fokus immer mehr auf die Strukturierung von Programmen legen (das Programmierparadigma der objektorientierten Programmierung) und dies zunächst am Beispiel des in C++ integrierten Klassenkonzept beschreiben. Das Konzept der objektorientierten Programmierung beruht auf der alltäglichen Erfahrung, dass man Objekte nach zwei Maßstäben beurteilt: Ein Objekt besitzt einerseits messbare Eigenschaften und ist aber auch andererseits über seine Verhaltensweisen definiert. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird und die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf diesem Konzept der *Klasse*. In einer C++ Klasse werden die messbaren Eigenschaften des Objektes in Instanzvariablen (Daten-Member) gespeichert und durch Konstruktoren werden diese Daten-Member dann initialisiert. Die Verhaltensweisen des Objektes werden durch klasseninterne Funktionen, die sogenannten Member-Funktionen beschrieben. In dem folgenden Link werden die Grundlagen der Objekt-orientierten Programmierung und C++ Klassen allgemein vorgestellt und die dort besprochenen Konzepte werden in den nächsten Vorlesungen benutzt, um das Verhalten der Spieler auf einem komplexen Netzwerk in Objekt-orientierter Weise in den C++ und Python Programmen zu implementieren.

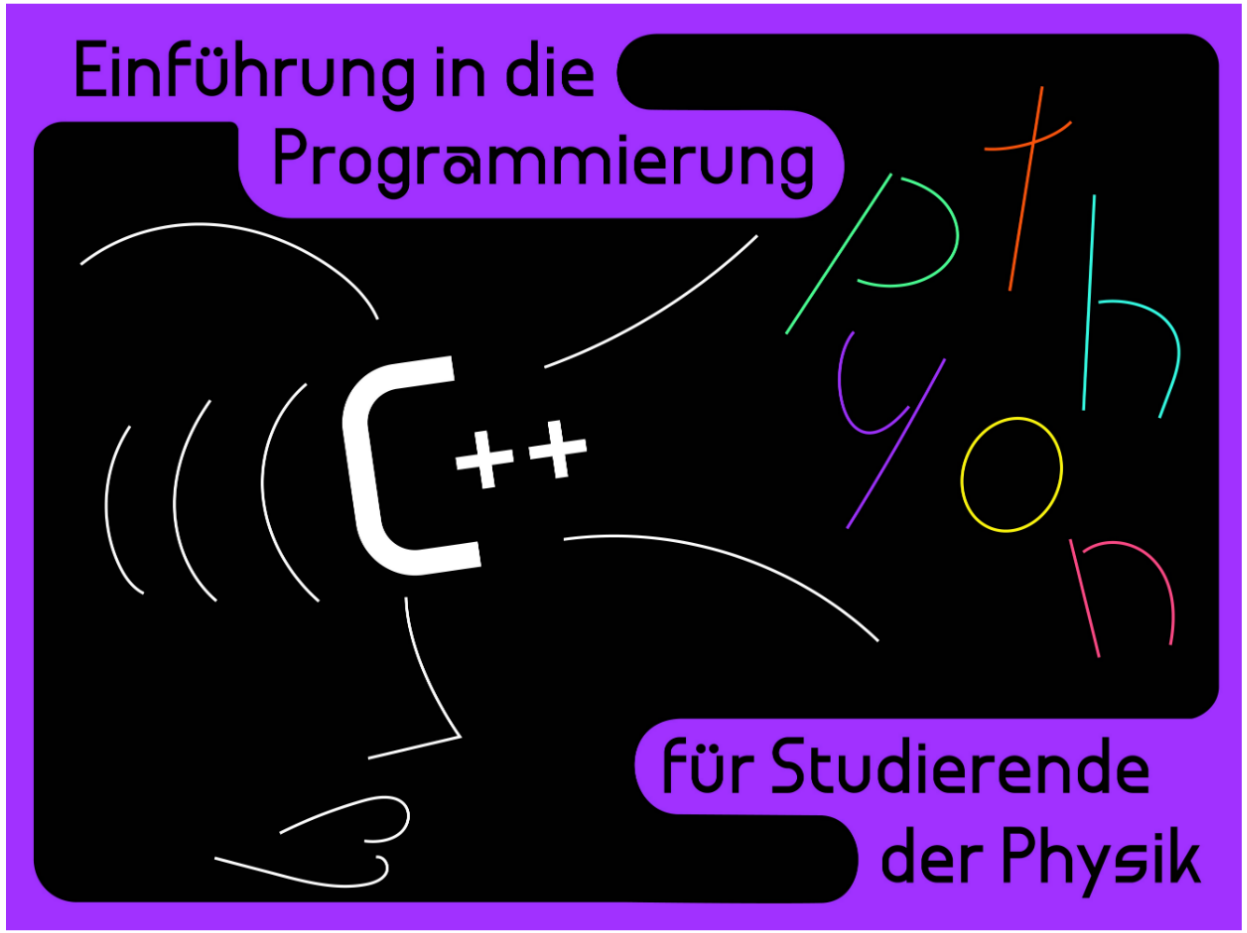


Illustration: Deborah Moldawski

Nächster Zoom Link am 14.07.2022, 14:00-16:00 Uhr: ID: 794 847 5614, PWD: 785453

Die Vorlesung *Einführung in die Programmierung für Studierende der Physik* stellt ein Pflichtmodul im Bachelor Studium Physik der Goethe-Universität Frankfurt dar. Bei regelmäßiger und erfolgreicher Teilnahme an den Übungen/Praktika erhalten Sie eine Zulassung zur Klausur. Den benoteten Schein und sechs Creditpoints erhält man schließlich bei bestandener Klausur. Falls Sie bereits in einem vergangenen Semester (nach der alten Studienordnung) die Zulassung zur Klausur erhalten haben, können Sie direkt an der abschließenden Klausur teilnehmen. Jedoch rate ich Ihnen, die Vorlesung und die Übungen/Praktika trotzdem nochmals zu belegen, da sich die Inhalte und Schwerpunkte zu den vergangenen Vorlesungen unterscheiden könnten.

Objekt-orientierte Programmierung und C++ Klassen

Einführung

Die C++ Typen, die wir bisher kennengelernt hatten (z.B. int i, double a, int v[3], double A[4][5]), die sogenannten *integrierten Typen*, werden wir nun mittels eines *Abstraktionsmechanismus* erweitern, um eigene, benutzerdefinierte Typen zu erstellen. Ein *benutzerdefinierter Typ*, wie z.B. die C++ Struktur '**struct**' oder die C++ Klasse '**class**', ist ein Abstraktionskonzept, das den Quelltext eines C+ Programms übersichtlicher macht, indem es das Programm in voneinander separierbare Teilbereiche aufteilt. Große Programme bestehen oft aus einzelnen Teilaufgaben, die man mittels einer sinnvollen Klassenstruktur voneinander trennen und ordnen kann. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird und die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf diesem Konzept der *Klasse*.

Benutzerdefinierte Typen und Abstraktionsmechanismen in C++

Das Konzept der *objektorientierten Programmierung* beruht auf der alltäglichen Erfahrung, dass man Objekte nach zwei Maßstäben beurteilt: Ein Objekt besitzt einerseits messbare Eigenschaften (z.B. Farbe, Gewicht, ...) und ist aber auch andererseits über seine Verhaltensweisen (z.B. zeitliches Verhalten, Interaktionsverhalten, Bewegungsverhalten, ...) definiert. Eine Klasse ist ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von realen/fiktiven Objekten (Klassifizierung). Mittels des Konzeptes der Klasse lassen sich solche Objekte im Programm realisieren. Eine Klasse stellt dabei den Bauplan für das zu beschreibende Objekt bereit und die wirkliche Realisierung des Objektes (die Instanzbildung) findet dann im Hauptprogramm zur Laufzeit statt. Die formale Beschreibung wie das Objekt beschaffen ist, d.h. welche Merkmale (Instanzvariablen bzw. Daten-Member der Klasse) und Verhaltensweisen (Methoden der Klasse bzw. Member-Funktionen) das zu beschreibende Objekt hat, werden innerhalb der Klasse definiert. Eine Klasse ist also eine Vorlage, eine abstrakte Idee, die ein Grundgerüst von Eigenschaften und Methoden vorgibt. Die Erzeugung eines Objektes dieser Klasse entspricht der Materialisierung dieser Idee im Programm. Bei der Erzeugung des Objektes wird der sogenannte *Konstruktor* der Klasse aufgerufen, und verlässt das Objekt den Gültigkeitsbereich seines Teilbereiches des Programms, wird es durch den sogenannten *Destruktor* wieder zerstört. Das Grundgerüst einer Klasse besitzt die folgende Form, wobei im Anweisungsblock der Klasse nicht alle der aufgezählten Größen definiert werden müssen.

class Klassenname { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

C++ Klassen: Zugriffskontrolle und die öffentlich zugänglichen Bereichen eines Objektes

```
class Klassenname {
    // Private Instanzvariablen (Daten-Member) der Klasse
    ...

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Standard-Konstruktor und überladene Konstruktoren der Klasse
        ...

        // Member-Funktionen der Klasse
        ...
};
```

Eine weitere wichtige Klassen-Terminologie ist die Kennzeichnung von privaten und öffentlich zugänglichen Bereichen des Objektes. In einer Klasse werden die Daten-Member und Member-Funktionen nach außen gekapselt, sodass der Benutzer der Klasse sie nicht manipulieren kann (**private**-Bereiche der Klasse). Kennzeichnet man einen Bereich der Klasse jedoch als **public**, so kann man von außen auf die Daten und Methoden zugreifen und sie auch verändern. Neben diesen beiden Klassifizierungsbegriffen gibt es zusätzlich die Kennzeichnung **protected**, bei der man nur von Unterklassen heraus auf die Daten und Methoden zugreifen kann. Besitzt eine Klasse keine explizite Kennzeichnung von privaten und öffentlich zugänglichen Bereichen, so sind alle Merkmale der Klasse privat. Bei der Verwendung der C++ Struktur '**struct**' sind hingegen alle Merkmale öffentlich und man kann '**struct**' somit als eine öffentliche '**class**' ansehen. Die nebenstehende Abbildung veranschaulicht die Schreibweise einer C++ Klasse im Quellcode, wobei gewöhnlicherweise zunächst die privaten und dann die als öffentlich gekennzeichneten Definitionen und Anweisungen folgen.

Merkmale von C++ Klassen: Daten-Member und Member-Funktionen

Daten und Funktionen, die in einer Klassendefinition deklariert werden, bezeichnet man als *Daten-Member* (Instanzvariablen) und *Member-Funktionen* (Klassen-interne Funktionen). Durch die Bezeichner private, protected und public findet eine Kapselung der Klassen-internen Merkmale von den anderen Bereichen des C++ Programmes statt. Der

Objektorientiertes Programm: Die Klasse „Players“

SpatialGame_2x2_Class.py

home > hanauske > neu_2025 > VPSOC_2025 > Vorlesungen > 11 > SpatialGame_2x2_Class.py

```python
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from random import randint, uniform, random, choice
from math import isclose
import os
from matplotlib import rcParams
import matplotlib.gridspec as gridspec

# Klasse der relevanten Eigenschaften der Spieler
class Players:
    # Konstruktor zur Initialisierung der Spielereigenschaften
    def __init__(self, n_nodes, initial_coop_prob=0.3):
        # Instanzvariablen (Daten-Member) der Klasse:  [id, x, y, current_strat, payoff, next_strat, last_opponent_strat]
        # data:= [id, x, y, current_strat, payoff, next_strat, last_opponent_strat]
        self.data = np.zeros((n_nodes, 7))
        self.n_nodes = n_nodes

        # Initialisierung
        k = 0
        for i in range(int(np.sqrt(n_nodes))):
            for j in range(int(np.sqrt(n_nodes))):
                self.data[k, 0] = k
                self.data[k, 1] = i
                self.data[k, 2] = j
                self.data[k, 3] = 1 if uniform(0, 1) < initial_coop_prob else 0
                self.data[k, 5] = self.data[k, 3]
                self.data[k, 6] = 1 - self.data[k, 3]  # dummy
                k += 1
    # Öffentlichen Member-Funktionen der Klasse
    def reset_payoffs(self):
        self.data[:, 4] = 0.0

    def set_next_strategy(self, node_idx, strategy):
        self.data[node_idx, 5] = strategy

    def apply_next_strategies(self):
        self.data[:, 3] = self.data[:, 5]

    def get_current_strategy(self, node_idx):
        return self.data[node_idx, 3]

    def get_last_opponent_strategy(self, node_idx):
        return self.data[node_idx, 6]

    def update_last_opponent(self, node_idx, opponent_strategy):
        self.data[node_idx, 6] = opponent_strategy

    def mean_strategy(self):
        return np.mean(self.data[:, 3])
```

Python Programm

```python
    # Öffentliche Member-Funktion: Strategie-Update nach verschiedenen Regeln (rule)
    def update_strategies(self, graph, rule):
        for node in range(self.n_nodes):
            neighbors = list(graph.neighbors(node))
            if not neighbors:
                continue

            my_payoff = self.data[node, 4]
            my_current = self.get_current_strategy(node)
            # Standardmäßig: next = current (falls keine Änderung)
            self.set_next_strategy(node, my_current)

             # Imitate the Best
            if rule == 0:
                neigh_payoffs = self.data[neighbors, 4]
                max_p = np.max(neigh_payoffs)
                if max_p > my_payoff and not isclose(max_p, my_payoff):
                    best_idx = neighbors[np.argmax(neigh_payoffs)]
                    best_strategy = self.get_current_strategy(best_idx)
                    self.set_next_strategy(node, best_strategy)

            # Zufälliger besserer Nachbar
            elif rule == 1:
                neigh = neighbors[randint(0, len(neighbors)-1)]
                if self.data[neigh, 4] > my_payoff and not isclose(self.data[neigh, 4], my_payoff):
                    self.set_next_strategy(node, self.get_current_strategy(neigh))

            # Tit-for-Tat (Mehrheitsentscheidung)
            elif rule == 2:
                last_moves = [self.get_last_opponent_strategy(n) for n in neighbors]
                coop_count = sum(last_moves)
                new_strat = 1 if coop_count >= len(neighbors)/2 else 0
                self.set_next_strategy(node, new_strat)

            # Fermi-Funktion
            elif rule == 3:
                fermi_K = 0.5 # Temperatur für Fermi-Regel
                neigh = neighbors[randint(0, len(neighbors)-1)]
                neigh_payoff = self.data[neigh, 4]
                neigh_strat = self.get_current_strategy(neigh)
                if my_current != neigh_strat:  # Nur wenn unterschiedlich
                    delta = my_payoff - neigh_payoff
                    prob = 1 / (1 + np.exp(delta / fermi_K))
                    if random() < prob:
                        self.set_next_strategy(node, neigh_strat)

            # # Imitate the Best mit Fehlerrate μ
            elif rule == 4:
                mu = 0.01 # Fehlerrate μ
                neigh_payoffs = self.data[neighbors, 4]
                max_p = np.max(neigh_payoffs)
                if max_p > my_payoff and not isclose(max_p, my_payoff):
                    best_idx = neighbors[np.argmax(neigh_payoffs)]
                    best_strategy = self.get_current_strategy(best_idx)
                    self.set_next_strategy(node, best_strategy)
                if random() < mu:
                    self.set_next_strategy(node, choice([0, 1]))

            else:
                raise ValueError(f"Unbekannte Update-Regel: {rule}")
```

Weitere „update“-Regeln

Die Klasse „SpatialGameSimulation“

```python
# Klasse der räumlichen Simulation des Spiels
class SpatialGameSimulation:
    # Konstruktor zur Initialisierung des räumlichen Spiels
    def __init__(self, width=105, height=105, nit=30, rule=1,
                 payoff_params=(3,4,1,5), initial_coop=0.3, output_dir="./output"):
        self.width = width
        self.height = height
        self.nit = nit
        self.rule = rule
        self.a, self.b, self.c, self.d = payoff_params
        self.output_dir = output_dir

        os.makedirs(output_dir, exist_ok=True)

        self.graph = self._create_spatial_grid() # Erzeugung des Networkx Graphen des räumlichen Gitters
        self.players = Players(width * height, initial_coop) # Instanzbildung der Spieler

        self.av_strat_history = [self.players.mean_strategy()]
        self.av_payoff_history = []

        self._setup_plot()

    # Protected Member-Funktion: Erzeugung des räumlichen Gitters
    def _create_spatial_grid(self):
        g = nx.Graph()
        n = self.width * self.height

        for i in range(self.width):
            for j in range(self.height):
                k = i * self.height + j
                g.add_node(k)

                neighbors = [
                    ((i-1)%self.width, (j-1)%self.height),
                    ((i  )%self.width, (j-1)%self.height),
                    ((i+1)%self.width, (j-1)%self.height),
                    ((i+1)%self.width, (j  )%self.height),
                    ((i+1)%self.width, (j+1)%self.height),
                    ((i  )%self.width, (j+1)%self.height),
                    ((i-1)%self.width, (j+1)%self.height),
                    ((i-1)%self.width, (j  )%self.height),
                ]
                for ni, nj in neighbors:
                    neigh_id = ni * self.height + nj
                    if neigh_id != k:
                        g.add_edge(k, neigh_id)
        return g
```

```python
    def payoff(self, s_a, s_b):
        return (self.a * s_a * s_b +
                self.b * s_a * (1-s_b) +
                self.c * (1-s_a) * s_b +
                self.d * (1-s_a) * (1-s_b))


    def compute_all_payoffs(self):
        self.players.reset_payoffs()

        for u, v in self.graph.edges():
            su = self.players.get_current_strategy(u)
            sv = self.players.get_current_strategy(v)

            pu = self.payoff(su, sv)
            pv = self.payoff(sv, su)

            self.players.data[u, 4] += pu
            self.players.data[v, 4] += pv

            if self.rule == 2:  # Nur für TfT notwendig
                self.players.update_last_opponent(u, sv)
                self.players.update_last_opponent(v, su)


    def update_strategies(self):
        # Weiterleitung an die Klasse Players mit den Parametern
        self.players.update_strategies(self.graph, self.rule)


    def _setup_plot(self):
        rcParams.update({'figure.figsize': [7.5, 10], 'text.usetex': True})
        self.fig = plt.figure()
        gs = gridspec.GridSpec(2, 1, height_ratios=[1, 3.0], hspace=0.1)

        self.ax1 = plt.subplot(gs[0])
        self.ax2 = plt.subplot(gs[1])

        self.ax1.set_xlim(0, self.nit-1)
        self.ax1.set_ylim(0, 1)
        self.ax1.set_ylabel(r'$x(t)$')
```

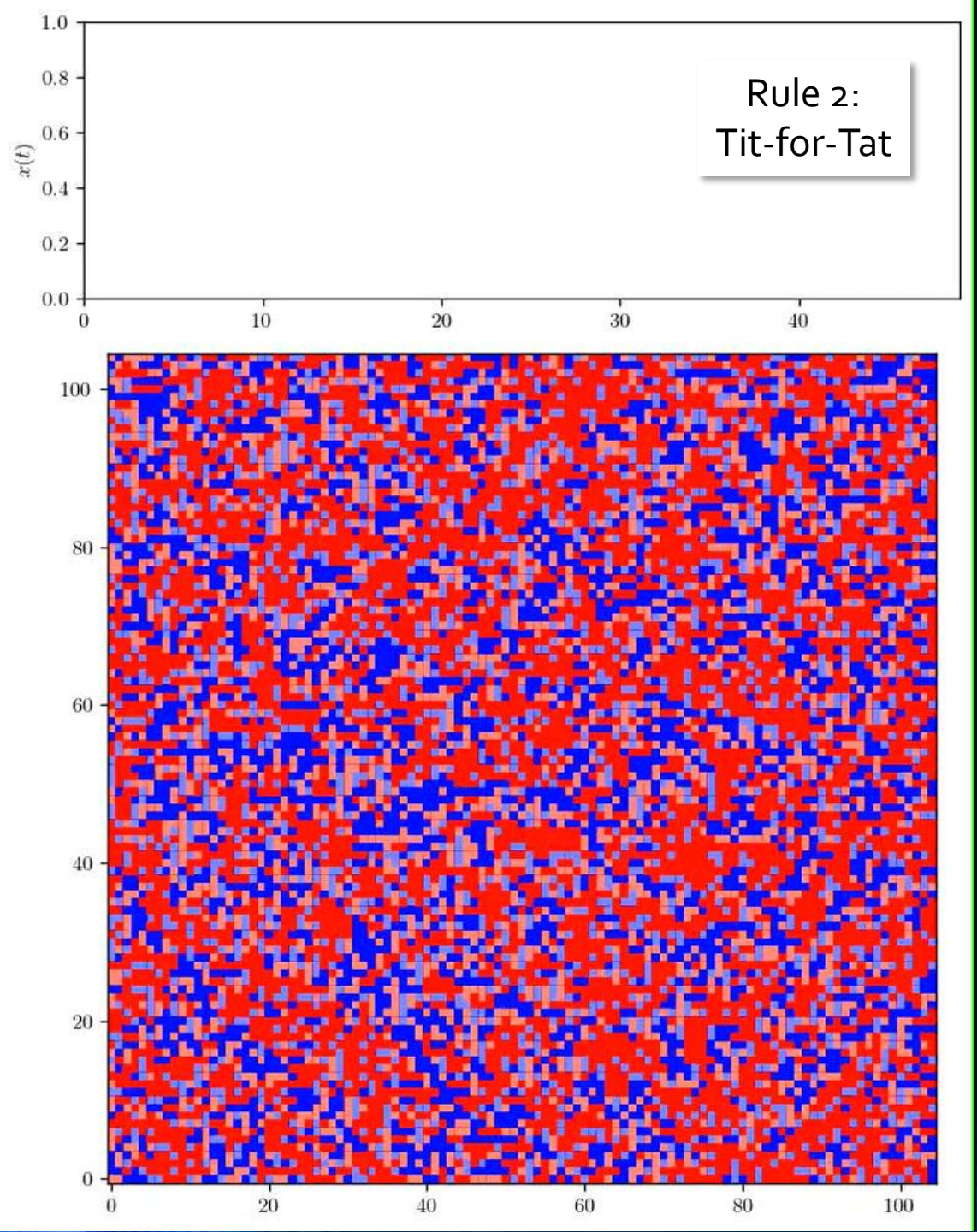
```
204
205     def plot_current_state(self, iteration):
206         p = self.players.data
207         sgross = np.sqrt(2822400 / self.players.n_nodes)
208
209         colors = ['red' if s == 0 else 'blue' for s in p[:, 3]]
210         alpha = [1 if curr == next_s else 0.5
211                  for curr, next_s in zip(p[:, 3], p[:, 5])]
212
213         self.ax1.plot(range(len(self.av_strat_history)), self.av_strat_history, 'k-')
214         self.ax2.scatter(p[:,1], p[:,2], s=sgross, c=colors, marker="s",
215                          alpha=alpha, edgecolor='none')
216
217         self.ax2.set_xlim(-0.5, self.width-0.5)
218         self.ax2.set_ylim(-0.5, self.height-0.5)
219         self.ax2.set_aspect('equal')
220
221     # Öffentlichen Member-Funktion: Simulation durchführen
222     def run(self):
223         for it in range(self.nit):
224             print(f"Iteration {it:3d}", end=" ... ")
225
226             self.compute_all_payoffs()
227             self.update_strategies()
228
229             self.plot_current_state(it)
230
231             mean_strat = self.players.mean_strategy()
232             mean_payoff = np.mean(self.players.data[:, 4]) / 8
233
234             self.av_strat_history.append(mean_strat)
235             self.av_payoff_history.append(mean_payoff)
236
237             self.players.apply_next_strategies()
238
239             fname = f"img-{it:03d}"
240             self.fig.savefig(os.path.join(self.output_dir, f"{fname}.png"),
241                              dpi=120, bbox_inches="tight")
242             self.ax2.clear()
243
244             print(f"x = {mean_strat:.4f}")
245
246         print("\nSimulation abgeschlossen.")
247         print("Mittlere Auszahlungen:", [round(x,3) for x in self.av_payoff_history])
248
```

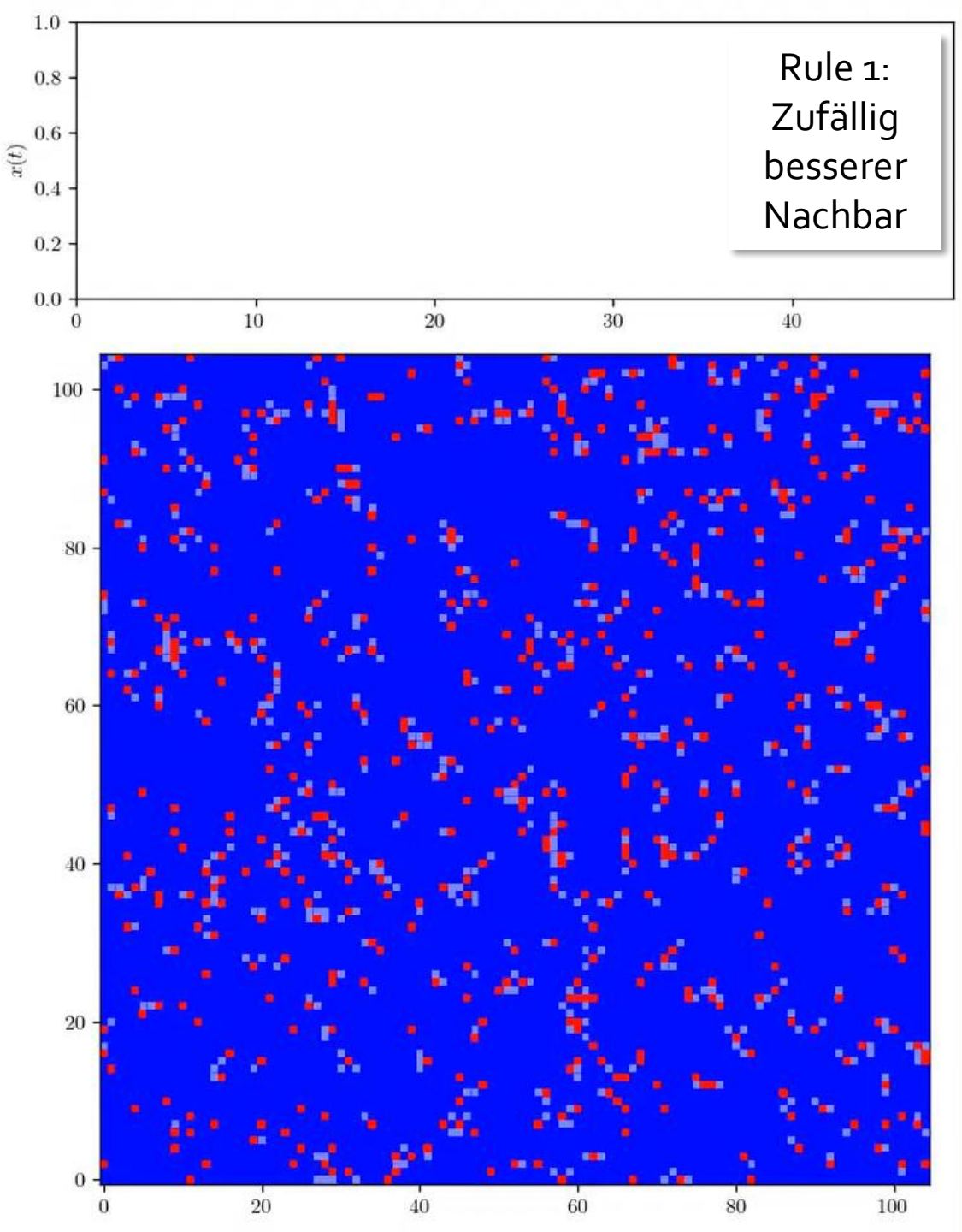
Ende der Klasse „SpatialGameSimulation“ und Instanzbildung der Klasse

Öffentliche Member-Funktion der Klasse „SpatialGameSimulation“

Starten der Simulation

```
250 if __name__ == "__main__":
251     # Intanzbildung der Simulationsklasse
252     sim = SpatialGameSimulation(
253         width=105,
254         height=105,
255         nit=40,
256         rule=2,
257         payoff_params=(-3, -9, -1, -7),
258         initial_coop=0.4,
259         output_dir="./output"
260     )
261
262     # Durchführen der Simulation
263     sim.run()
264
```


Rule 2:
Tit-for-Tat
x(t)


Rule 1:
Zufällig
besserer
Nachbar
x(t)