E.Tutorial
www.et.ethz.ch

Programmieren mit Python oder MATLAB Modul 6

Matrizenrechnen, Monte-Carlo-Simulationen und Zufallsexperimente

Selbstständiger Teil

Inhaltsverzeichnis

1 Überblick **3**

2 Teil A: Modellierung eines stochastischen Systems **3**
2.1 Einleitung . . . 3
2.2 Aufgabenstellung . . . 3
2.3 Zwischenschritte . . . 4
2.3.1 Schritt 1: Entscheidung einer Kugel an einem Nagel . . . 4
2.3.2 Schritt 2: Höhe dynamisch gestalten . . . 6
2.3.3 Schritt 3: Position einer Kugel nach dem Nagelbrett-Durchlauf . . . 6
2.3.4 Schritt 4: Kugel-Behälter einbauen . . . 7
2.3.5 Schritt 5: Position in Behälternummer umrechnen . . . 8
2.3.6 Schritt 6: Zähler für viele Kugeln einbauen . . . 9
2.3.7 Schritt 7: Visualisierung des Resultats . . . 10
2.4 Erweiterung . . . 10

3 Teil B: Tic Tac Toe-Spiel **10**
3.1 Einleitung . . . 10
3.2 Aufgabenstellung . . . 10
3.3 Zwischenschritte . . . 11
3.4 Erweiterungen . . . 11

4 Teil C: Waldbrand-Simulation **11**
4.1 Einführung und Aufgabenstellung . . . 11

4.2 Beschreibung des Modells 12
4.3 Ausgangssituation 13
4.4 Mögliche Zwischenschritte 13
4.5 Erweiterungen 14

5 Bedingungen für die Präsentation **15**

Begriffe

Vektor	Zeile (Row)	Zufallsexperiment
Matrix/Matrizen	Spalte (Column)	Stochastisches System
Dimension	Index/Indizes	
Array	Diskretisierung	Monte-Carlo-Simulation

Autoren:

Lukas Fässler

E-Mail:

et@ethz.ch

Datum:

29 November 2024

Version: 1.1

Hash: 9f51da6

Trotz sorgfältiger Arbeit schleichen sich manchmal Fehler ein. Die Autoren sind Ihnen für Anregungen und Hinweise dankbar!

1 Überblick

Der selbstständige Teil dieses Kurses besteht aus folgenden Teilen:

- Teil A: Modellierung eines stochastischen Systems (Galton-Board)
- Teil B: Tic Tac Toe-Spiel
- Teil C: Waldbrand-Simulation

Hinweis zur Bearbeitung dieses Moduls

Die Aufgaben dieses Moduls können sowohl mit **Python** (mit den Bibliotheken *NumPy* und *matplotlib*) als auch mit **MATLAB®** bearbeitet werden. Falls Sie dieses Modul im Rahmen eines Kurses absolvieren, informieren Sie sich, in welcher Programmiersprache dieses Modul bearbeitet werden soll.

2 Teil A: Modellierung eines stochastischen Systems

2.1 Einleitung

Das *Galton-Board* (benannt nach *Francis Galton*) ist ein Modell eines „Nagelbretts“ zur Demonstration und Veranschaulichung der Binomialverteilung, einer Wahrscheinlichkeitsverteilung, die in vielen Zufallsexperimenten eine Rolle spielt.

In Abbildung 1 ist ein mögliches *Galton-Board* visualisiert. Von einer Startposition rollt eine Kugel eine schiefe Ebene hinunter, trifft auf eine Reihe von Nägeln und landet schlussendlich in einem Auffangbehälter. Nachdem die Kugel auf den ersten Nagel getroffen ist, wird ihr Weg mit gleicher Wahrscheinlichkeit rechts oder links vom Nagel fortgesetzt. Nach einer bestimmten Anzahl von Entscheidungen, d.h. wenn die Kugel durch das ganze Nagelbrett gerollt ist, landet sie mit einer gewissen Wahrscheinlichkeit in einem der Behälter.

Der Weg einer einzelnen Kugel kann nicht vorausgesagt werden. Wird hingegen eine grosse Anzahl Kugeln nacheinander durch das System geschickt, findet man eine charakteristische Wahrscheinlichkeitsverteilung in den Behältern. Diese Wahrscheinlichkeiten heissen Bernoulli-Zahlen und stellen eine Binominalverteilung dar.

Bei **stochastischen Systemen** können wir bei einem Einzelereignis nicht voraussagen, wie es sich verhalten wird. Summieren wir hingegen die Einzelprozesse, kann ein im Voraus einschätzbares (deterministisches) System simuliert werden.

2.2 Aufgabenstellung

Sie haben die Aufgabe, eine *Galton-Board*-Simulation zu programmieren, welche folgende Anforderungen erfüllt:

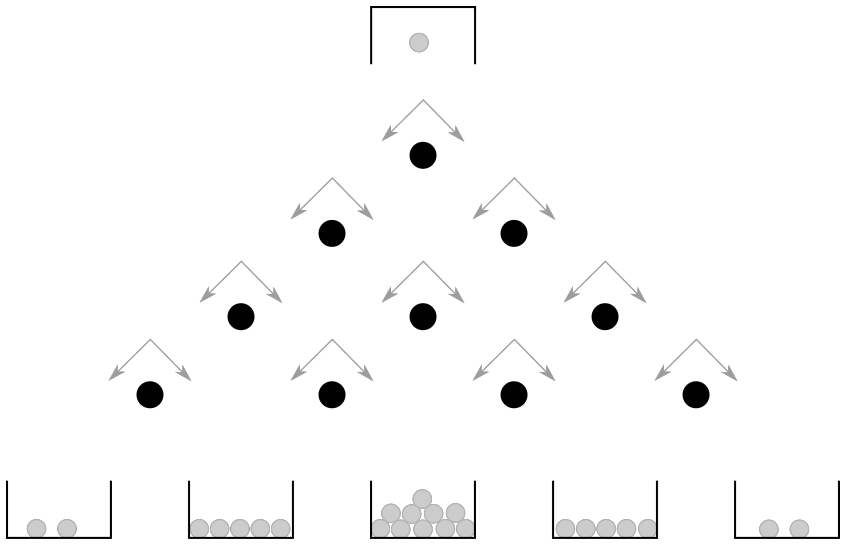

Abbildung 1: Nagelbrett der Höhe vier. Bei jedem Nagel (schwarzer Kreis) wird die Kugel (graue Kreise) zufällig nach links oder rechts abgelenkt.

- Es können verschiedene Werte für die **Höhe** und **Anzahl Kugeln** eingegeben werden.
- Als Resultat sollen die **Anzahlen der Kugeln** in den einzelnen Behältern ausgegeben und visualisiert werden (siehe Abbildung 2). Wir verzichten auf eine detaillierte Abbildung des Modells auf dem Bildschirm.

2.3 Zwischenschritte

Die Zwischenschritte der Aufgabe im Überblick:

- Schritt 1: Entscheidung einer Kugel an einem Nagel
- Schritt 2: Höhe dynamisch gestalten
- Schritt 3: Position einer Kugel nach einem Nagelbrett-Durchlauf
- Schritt 4: Kugel-Behälter einbauen
- Schritt 5: Position in Behälternummer umrechnen
- Schritt 6: Zähler für viele Kugeln einbauen
- Schritt 7: Visualisierung des Resultats

2.3.1 Schritt 1: Entscheidung einer Kugel an einem Nagel

- Generieren Sie eine Zufallszahl für eine 50:50-Entscheidung.

Als Erstes schauen wir uns das Ereignis einer Kugel an einem Nagel an. Trifft eine Kugel auf dem Nagelbrett auf einen Nagel und rollt dann mit gleicher Wahrscheinlichkeit entweder links oder rechts weiter, stellt dies bezüglich der Wahrscheinlichkeit dasselbe

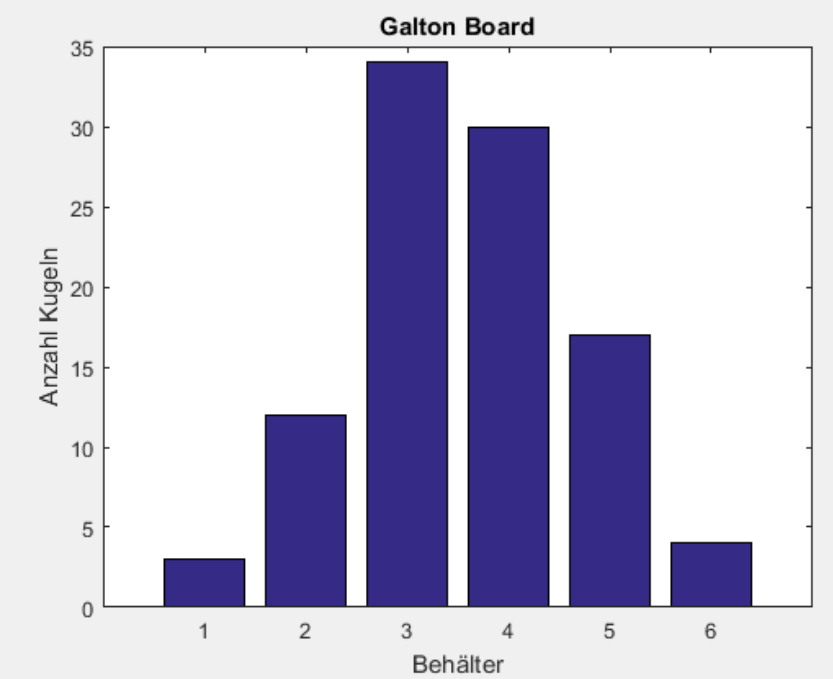

Abbildung 2: Mögliche Ausgabe des Programms nach Eingabe der Werte 5 und 100.

Problem dar, wie wenn wir bei einer Münze auf *Kopf* oder *Zahl* setzen können – eine 50:50-Wahrscheinlichkeitsentscheidung.

Da wir beim Programmieren weder eine Münze noch ein Nagelbrett und eine Kugel zur Verfügung haben, müssen wir versuchen, das Problem mathematisch zu definieren: Zunächst wird eine **Zufallszahl** generiert, welche dann anhand eines definierten Grenzwertes in zwei Bereiche „links“ oder „rechts“ bzw. 0 und 1 unterteilt werden kann.

- Weisen Sie der generierten Zufallszahl je nach Höhe ihres Wertes die Zahl 0 oder 1 zu.

Eine 50:50-Entscheidung sollte den gewählten Zahlenraum in zwei gleich grosse Teile unterteilen. Für jeden Fall weisen wir je ein Resultat zu. An Stelle von „links“ oder „rechts“ setzen wir hier eine neue Variable auf die beiden Werte 0 oder 1, weil wir diese rechnerisch weiter verwenden können.

Mögliche Ausgaben:

```
Zufallszahl:
0.3833
Variablenwert:
0

Zufallszahl:
0.7287
Variablenwert:
1
```

2.3.2 Schritt 2: Höhe dynamisch gestalten

- Wiederholen Sie die Entscheidungsprozesse für die Nagelbretthöhe h.

Auf einem Nagelbrett durchläuft eine Kugel eine Reihe solcher im letzten Schritt programmierter Entscheidungsprozesse. Wie viele Entscheidungsprozesse aufeinander folgen, ist abhängig von der **Anzahl der Nagelreihen**, also der **Höhe des Nagelbrettes**.

Mögliche Ausgaben:

```
Höhe? 5
1
0
0
1
0
```

2.3.3 Schritt 3: Position einer Kugel nach dem Nagelbrett-Durchlauf

- Bestimmen Sie die Position einer Kugel nach dem Nagelbrett-Durchlauf der Höhe h und geben Sie diese in der Konsole aus. Welche möglichen Werte gibt es?

Mögliche Ausgaben:

```
Höhe? 4
1
1
1
0

Summe:
3

Höhe? 5
0
0
1
0
1

Summe:
2
```

2.3.4 Schritt 4: Kugel-Behälter einbauen

Damit wir mehrere Kugeln durch unser Nagelbrett schicken können, brauchen wir am Ende des Nagelbretts „Behälter“, in denen die Kugeln aufgefangen und dann später gezählt werden. Die Anzahl der Behälter hängt ebenfalls von der gewählten Höhe ab.

- Berechnen Sie die Anzahl der Behälter.
- Stellen Sie für jeden Behälter Speicherplatz als Zelle in einem dynamischen Vektor zur Verfügung.
- Geben Sie die entsprechenden Speicherplätze am Bildschirm aus (sie enthalten momentan noch jeweils den Wert 0).

Hinweis: Beachten Sie, dass die Anzahl der Behälter mit der Höhe des Nagelbretts zusammenhängt.

Mögliche Ausgabe:

```
Höhe? 5
0
1
1
0
1

Summe:
3

Kugeln in den Behaeltern:
0     0     0     0     0     0
```

2.3.5 Schritt 5: Position in Behälternummer umrechnen

- Berechnen Sie die Nummer des Behälters, in den eine Kugel nach dem Durchlauf durch das Nagelbrett mit einstellbarer Höhe gefallen ist.

> **Tipp:** Die Position nach dem Durchlauf einer Kugel durch das Nagelbrett ist in Ihrem momentanen Programm die Summe der Einzelentscheidungen abhängig von der Höhe (z.B. 0, 1, 2, 3 oder 4 für die Höhe 4). Die Behälter sind als Vektor repräsentiert, dessen Grösse mit der Höhe variiert. Um mit der Position den Indexwert des Vektors ansprechen zu können, muss der Wert der Position noch um 1 erhöht werden.

Mögliche Ausgabe:

```
Höhe? 5
0
1
1
0
1

Summe:
3

gelandet in Behälter:
4

Kugeln in den Behaeltern:
0     0     0     0     0     0
```

2.3.6 Schritt 6: Zähler für viele Kugeln einbauen

- Wiederholen Sie die Entscheidungsprozesse für **viele Kugeln** und das **Zählen der Anzahl Kugeln** im entsprechenden Behälter.

Zwischenschritte:

- Programmieren Sie eine Eingabemöglichkeit für die Anzahl Kugeln.
- Wiederholen Sie für jede Kugel den Weg durch das Nagelbrett.
- Erhöhen Sie in jenem Behälter die Anzahl Kugeln, in den sie gefallen ist.

> **Tipp:** Die Summe der Einzelentscheidungen einer Kugel muss für jede Kugel wieder bei 0 beginnen.

Mögliche Ausgabe:

```
Höhe? 5
Anzahl Kugeln? 100
...
...
...
Kugeln in den Behaeltern:
5     15     39     29     9     3
```

2.3.7 Schritt 7: Visualisierung des Resultats

- Stellen Sie die berechneten Resultate in einem Diagramm grafisch dar.

2.4 Erweiterung

- Erweitern Sie Ihr Galton-Board zu einer Animation.

3 Teil B: Tic Tac Toe-Spiel

3.1 Einleitung

Beim Spiel *Tic Tac Toe* setzen zwei spielende Personen auf einem quadratischen Spielfeld der Grösse 3×3 abwechselnd ihr Zeichen (z.B. Kreuze und Kreise) in ein freies Feld (siehe Beispiel in Abbildung 3). Die Person, die als erste drei Zeichen in eine Zeile, Spalte oder Diagonale setzen kann, gewinnt.

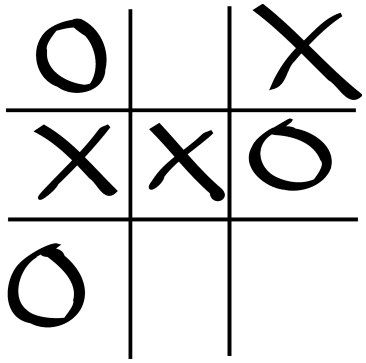

Abbildung 3: Spiel Tic Tac Toe (Details siehe Text).

3.2 Aufgabenstellung

Bei Ihrem Tic Tac Toe-Spiel sollen abwechselnd die beiden Zeichen `X` oder `O` in ein 3×3-Spielbrett gesetzt werden. Sobald eine der beiden Zeichen eine Diagonale erreicht hat, soll das Spiel enden.

So könnte das fertige Tic Tac Toe-Spiel aussehen:

```
  S 1 2 3
Z  -------
1 | O . X
2 | O X X
3 | . . O

Zeile: 3
Spalte: 1

Game over. Spielerin 1 hat gewonnen!
```

3.3 Zwischenschritte

- Erstellen Sie ein 3×3-Spielfeld (siehe Boss Puzzle-Spiel im E.Tutorial®).
- Schreiben Sie eine Funktion `setup()`, die alle Elemente des Spielfeldes mit einem Einheitszeichen (z.B. `'.'`) beschreibt.
- Schreiben Sie eine Funktion `ausgabe()`, die das Spielfeld in der Konsole ausgibt.
- Schreiben Sie eine Benutzereingabe für die beiden Koordinaten des gewünschten Feldes.
- Setzen Sie das Zeichen `'X'` an der von der spielenden Person gewünschten Stelle.
- Ermöglichen Sie mehrere Spielzüge. Wechseln Sie das Zeichen so, dass bei der nächsten Eingabe das Zeichen `'O'` gesetzt wird.
- Schreiben Sie eine Funktion `hatGewonnen()`, die auswertet, ob jemand eine Diagonale erreicht hat. Das Spiel soll in diesem Fall enden.

3.4 Erweiterungen

- Werten Sie auch drei Zeichen in einer Spalte oder einer Zeile aus und sorgen Sie dafür, dass auch dann das Spiel endet.
- Der Spielgegner soll durch den Computer mittels Zufallsgenerator gespielt werden.

4 Teil C: Waldbrand-Simulation

4.1 Einführung und Aufgabenstellung

Bei dieser selbstständigen Aufgabe werden Sie einen Waldbrand unter Anwendung von **zellulären Automaten** modellieren und simulieren (Theorie dazu siehe früheres Modul). Ein 2-dimensionales Spielfeld repräsentiert eine Waldfläche. Unser Waldbrandmodell unterscheidet in seiner Grundversion die Zustände *Leer*, *Baum*, *Feuer*, *Asche*, wobei der erste und der letzte Zustand dieselben sind (siehe Abbildung 4). Ein Waldbrand kann durch ein zufälliges Ereignis (z.B. Blitzschlag, Brandstiftung, etc.) ausgelöst werden.

Ein Brand in einem Waldflächen-Kompartiment kann auf eine benachbarte Baumfläche übergreifen.

	leer
	Baum
	Feuer
	Asche

Abbildung 4: Zustände im Waldbrandmodell. Ein leeres Feld kann auf den Zustand Baum wechseln, ein Baum kann Feuer fangen und schliesslich zu Asche mit dem Zustand leer werden.

4.2 Beschreibung des Modells

Das einfach Waldbrandmodell soll folgende Anforderungen erfüllen (siehe Abbildung 5):

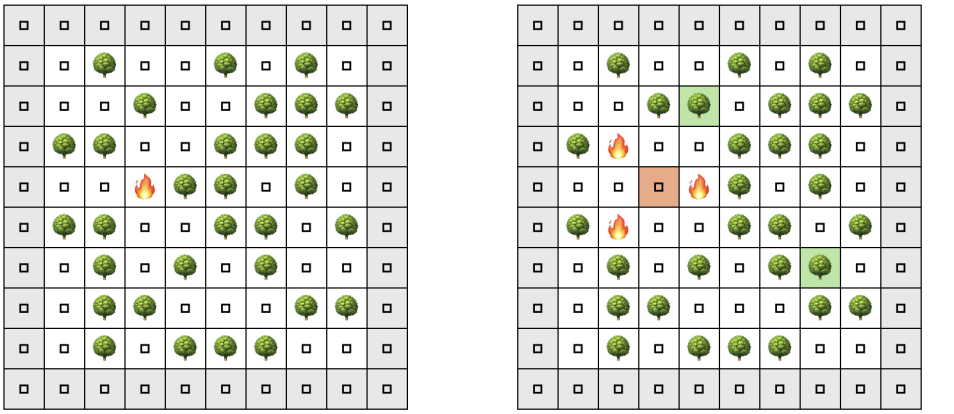

Abbildung 5: Beispiel der Regeln im einfachen Waldbrandmodell zu einem bestimmten Zeitpunkt (Gitter A) und einen Zeitpunkt später (Gitter B). Ein Baum im Gitter A hat spontan Feuer gefangen. In Gitter B haben eine Zeiteinheit später drei Nachbarzellen wegen ihrer Nachbarschaft zur brennenden Zelle ebenfalls Feuer gefangen (Wechsel von Baum zu Feuer). Gleichzeitig ist die zuvor brennende Zelle zu Asche geworden und zwei Bäume sind auf einem zuvor leeren oder auf einem Asche-Feld neu gewachsen.

- Die Waldfläche besteht aus einem Gitter der Grösse 8x8 plus Rand (total 10x10), umgesetzt als zwei NumPy-Arrays A und B.
- Die Simulation läuft über die Zeit t bis `tEnd`.
- Die Simulation startet mit einem bestimmten Prozentsatz an bewaldeten Kompartimenten (z.B. `grow_start`= 0.5).

- Auf den Wald-Kompartimenten wachsen Bäume mit einer Wahrscheinlichkeit p (z.B. 0.1) und können mit einer Wahrscheinlichkeit f (z.B. 0.005) Feuer fangen.
- Im Waldbrandmodell gibt es drei Zustände mit folgenden Werten: *Leer*, *Baum* und *Feuer*. Der Zustand *Asche* entspricht wieder dem Zustand *Leer*.
- Im Spielfeld werden bei jedem Durchgang folgende Regeln angewendet:
  - Befindet sich eine Zelle im Zustand *Leer*, dann wächst mit einer Wahrscheinlichkeit von p ein Baum. Der Zustand wechselt auf *Baum*.
  - Befindet sich eine Zelle im Zustand *Baum* und ist mindestens eine Nachbarzelle im Zustand *Feuer*, dann wechselt der Zustand der Zelle auf *Feuer*.
  - Falls ein *Baum* kein *Feuer* fängt, kann er mit einer Wahrscheinlichkeit von f Feuer fangen. Der Zustand wechselt in diesem Fall auf *Feuer*.
  - Falls ein *Baum* nicht Feuer fängt, bleibt er im Zustand *Baum*.
  - Befindet sich eine Zelle im Zustand *Feuer*, dann wechselt die Zelle den Zustand auf *Leer*.

4.3 Ausgangssituation

Bei dieser Aufgabe erhalten Sie als Grundgerüst ein Code-Skelett aus folgenden Elementen:

- Import-Anweisungen:
  - *NumPy* - ermöglicht die Verwendung von NumPy-Matrizen.
  - *matplotlib* - ermöglicht das Erstellen grafischer Darstellungen.
- Funktions-Definitionen:
  - `setup()` - kreiert die Ausgangssituation.
  - `print_grid()` - gibt die Matrix in der Konsole aus.
  - `burn()` - bestimmt, ob eine Nachbarzelle brennt.
  - `count()` - zählt die brennenden Kompartimente.
  - `update()` - berechnet die Zustände für eine nächste Zeiteinheit.
- Variablen im Hauptprogramm:
  - `EMPTY`, `TREE` und `FIRE` - speichert ein Emoji (*Bildschriftzeichen*) für die Zustände des Modells.
  - `size`, `grow_start`, `p`, `lightning` - setzt Simulationsparameter wie Gitter-Grösse, Wahrscheinlichkeiten für Baum beim Programmstart und Blitzeinschlag.
  - `Grid` - speichert das Gitter im Hauptprogramm.

4.4 Mögliche Zwischenschritte

Eine Grundversion der Waldbrand-Simulation kann in folgenden Zwischenschritten erstellt werden:

- Ausgangssituation in der Funktion `setup()` erstellen, im Hauptprogramm aufrufen und in `Grid` speichern.
- `Grid` durch die Funktion `print_grid()` in der Konsole ausgeben.
- Neue Variable `tEnd` und eine Schleife für mehrere Generationen einführen.
- Regeln zur Veränderung der Zellzustände in `update()` definieren, ins Hauptprogramm zurückgeben und `Grid` aktualisieren.
- In `burn()` prüfen, ob eine Nachbarzelle brennt.
- In `count()` zählen, wie viele Zellen im Zustand *Feuer* sind und in einer neue Liste abspeichern.
- Grafische Darstellung der Resultate über die Zeit (Beispiel siehe Abbildung 6).

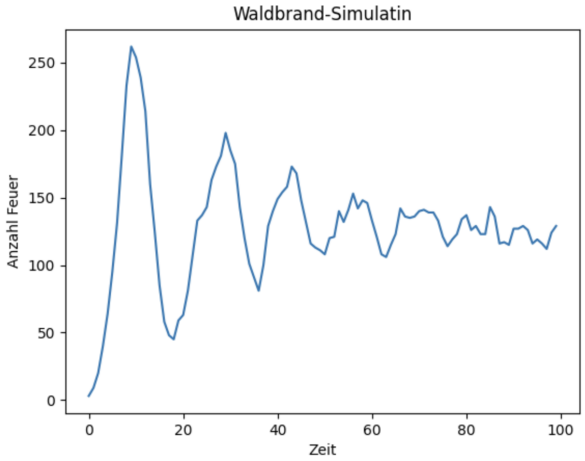

Abbildung 6: Beispiel einer Auswertung der Waldbrand-Simulation (size=50, grow_start=0.5, p=0.1, lightning=0.005, t=100).

4.5 Erweiterungen

Hier finden Sie einige Ideen, wie Sie Ihre Simulation erweitern können:

- Vergrössern Sie das Spielfeld (z.B. auf 100x100) und lassen Sie die Simulation mit unterschiedlichen Parametern laufen und werten Sie die Resultate aus.
- Erweitern Sie das Modell mit weiteren Zuständen, z.B. Bereiche, die kein Feuer fangen können (z.B. Brandschneise oder Strasse) oder verschiedene Wachstumsstufen der Bäume (z.B. junger Baum, wachsender Baum, ausgewachsener Baum). Definieren Sie neue Regeln und passen Sie die Regeln der Grundversion entsprechend an.
- In der Literatur wird beschrieben, dass in dynamischen Modellen nach einer bestimmten Zeit ein stabiler Zustand (so genannte *selbstorganisierte Kritikalität* oder *self-organized criticality*) erreicht werden kann. Das heisst, die Anzahl der Bäume im Spielfeld bleiben nach vielen Generationen mehr oder weniger konstant. Können Sie mit Ihrer Simulation einen solchen Zustand erreichen? Bei welcher Anzahl Bäumen wird der Zustand erreicht? Variieren Sie die Parameter p und f.

- Erstellen Sie eine Live-Animation mit `FuncAnimation` der Klasse `animation` der library `matplotlib` (für Leute, die gerne eine Herausforderung haben). Details finden Sie unter (https://matplotlib.org/stable/api/animation_api.html).

5 Bedingungen für die Präsentation

Führen Sie einer Assistenzperson Ihre erstellten Programme vor. Überlegen Sie sich, wie Sie einem Laien folgende Fragen erklären würden:

- Wie werden Vektoren und Matrizen gespeichert?
- Wie werden Elemente von Vektoren und Matrizen adressiert?
- Wie interpretieren Sie die Resultate Ihrer Simulationen?

Die Begriffe dieses Kursmoduls sollten Sie mit einfachen Worten erklären können.