Zellulärer Automat Aufgabe 1

April 23, 2023

[1]:
```
from pylab import *
import random


# print out each line in the input cell not only the last one
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# expand the cell width to 100% of t
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:70% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

Gruppe: Tobias Blümlhuber, Silvan Kron, Simon Gärtner

1 Kleines Projekt 1: Zellulärer Automat

1.1 Simulierung eines Waldbrandes

Ein mögliches Modell zur Simulation von Waldbränden basiert auf der Unterteilung einer Fläche in Zellen. Jede der Zellen kann 3 Zustände annehmen: Baum, Feuer, leer. Nach bestimmten Update-Regeln verändert sich die Zellbelegung in jedem Zeitschritt. Ein solches Modell nennt man zellulären Automaten.

Für einen solchen zellulären Automaten gilt:

- das untersuchte Gebiet wird in Zellen unterteilt (zellulär)
- die Zellen können bestimmte Zustände annehmen (Zustand)
- die Zustände ändern sich nach gewissen Regeln pro Zeitschritt (Automat)

Zelluläre Automaten finden in vielen Simulationsbereichen Anwendung, z.B. auch in der Verkehrssimulation.

Updateregeln:

Die Schlagkraft eines solchen Automaten steckt in den Updateregeln. Zunächst wird eine Anfangskonfiguration gesetzt, z.B. indem das Gebiet mit einer vom Anwender gegebenen Besetzungsdichte b zufällig mit Bäumen bepflanzt wird.

Für jede Zelle wird während der Simulation in jedem Zeitschritt der Zustand nach den folgenden Regeln neu bestimmt:

1. Ist eine Zelle leer, bleibt sie im nächsten Zeitschritt leer.
2. Falls sich in den 4 Nachbarzellen eines Baumes mindestens ein brennender Baum befindet, fängt der Baum mit der Wahrscheinlichkeit p_e Feuer. Ein brennender Baum brennt innerhalb eines Zeitschrittes herunter, im nächsten Zeitschritt ist die Zelle auf jeden Fall leer.

Aufgaben:

1. Schreiben Sie ein SageMath-Programm, das das obige einfache Waldbrandmodell umsetzt und grafisch darstellt. Die Gittergröße soll variabel sein.
2. Testen Sie anhand der Wahrscheinlichkeiten $p_e \in \{0.2, 0.5, 0.8\}$, wie sich das Feuer ausbreitet.
3. Implementieren Sie eine Statistikfunktion, die (z.B. für ein 20 x 20 Gitter) den Anteil des abgebrannten Waldes nach n Zeitschritten in Abhängigkeit von p_e experimentell ermittelt (jeweils für 5 verschiedene Initialisierungen).

- Plotten Sie den Anteil des abgebrannten Waldes in Abhängigkeit von der Zeit.
- Stellen Sie den Anteil des abgebrannten Waldes in Abhängigkeit p_e nach 10 Zeitschritten als Plot dar.

4. Wie behandeln Sie den Rand des Gebietes? Implementieren Sie eine weitere Möglichkeit und vergleichen Sie die Ergebnisse.
5. Nun soll noch die Möglichkeit eines Blitzeinschlages in einem Baum mit der Wahrscheinlichkeit p_b in das Modell aufgenommen werden. Vergleichen Sie mit Ihren bisherigen Ergebnissen.
6. Implementieren Sie *mindestens eine* der folgenden Modellerweiterungen und vergleichen Sie experimentell mit Ihren bisherigen Ergebnissen. Versuchen Sie Schlussfolgerungen zu finden:

- Ein Baum benötigt zum Abbrennen 2 Zeitschritte.
- Die Wahrscheinlichkeit des Baumes, Feuer zu fangen, steigt mit der Anzahl der brennenden Bäume in der Umgebung.
- In einer leeren Zelle wächst mit der Wahrscheinlichkeit p_n ein neuer Baum.
- Die Größe eines Baumes ändert sich von Zeitschritt zu Zeitschritt, ein großer Baum benötigt mehr Zeit zum Abbrennen.
- Berücksichtigen Sie Wind-Richtung (N, O, S, W) und -Geschwindigkeit (klein, mittel, hoch)

1.2 # Dokumentation und SageMath Skript

Wir haben uns dazu entschlossen, die Dokumentation in das SageMath Skript einzubinden. Daher finden Sie hier eine ausführliche Erklärung des Problems, des Skriptes, der Ergebnisse und der Arbeitsaufteilung.

1.3 ## Problembeschreibung

Um den Waldbrand-Automaten umzusetzen, mussten wir uns mit folgenden Teilproblemstellungen auseinandersetzen: - **Visualisierung**
Zuerst war zu entscheiden, worauf wir unsere Simulation aufbauen und wie wir sie graphisch visualisieren wollten. Hierbei entscheidend ist auch, wie die zwischenzeitlichen Zustände gespeichert werden sollten. - **Ereignisse**
Ein wesentlicher Teil des Automaten sind Ereignisse. Bei der Umsetzung ist deren Zeitpunkt und die Wahrscheinlichkeit, mit der sie stattfinden, wichtig. - **Auswertung**
Um das Verhalten des Automaten mit unterschiedlichen Parametern besser vergleichen zu können, muss die Auswertung unter gleichen Startbedingungen gestartet und dargestellt werden.

1.4 ## Mathematisches Modell

- **Speichern der Zustände**
  Um die Waldbrandsimulation mathematisch zu lösen, wird eine Matrix mit der zuvor gewählten Anzahl an Zeilen und Spalten erstellt, die die Zustände der einzelnen Felder beschreibt. Zudem werden die einzelnen Felder in ihre Zustände unterteilt und in separate Listen gespeichert. Somit ist es möglich, effizient auf die Position aller Elemente eines Zustandes zuzugreifen.
- **Wahrscheinlichkeiten bestimmter Ereignisse**
  Wie bereits erwähnt, finden bestimmte Ereignisse, die die Zustände von Feldern verändern, zu bestimmten Wahrscheinlichkeiten statt. Die Anzahl der Felder, welche ihren Zustand ändern sollen, wird anteilsmäßig von der gesamten Anzahl von Feldern, welche sich in dem Ausgangszustand befinden, berechnet. Da die Felder bereits in ihre Kategorien eingeteilt sind, kann hier einfach die Länge der jeweiligen Liste verwendet werden. Im Folgenden werden dann zufällig Elemente in der jeweiligen Liste ausgewählt und deren Zustand verändert, bis die Anzahl der berechneten zu ändernden Felder erreicht ist.

1.5 ## Implementierung in SageMath & Analyse und Deutung der Ergebnisse

Im folgenden Codeabschnitt finden Sie die Implementierung und Deutung der Ergebnisse der Simulation.

Folgende Struktur verfolgten wir in der Implementierung: 1. Methoden für alle Bestandteile des Programms, die wiederkehrend verwendet werden 2. Aufbau in 3 Teile:

```
a. Simulation des Automaten mit graphischer Ausgabe
b. Analytische Simulation mit verschiedenen Wahrscheinlichkeiten über ein Setup
c. Analytische Simulation mit allen Wahrscheinlichkeiten von 0 - 1 über verschiedene Setups
```

1.5.1 Arbeitsschritt 1:

Parameterfestlegung

In diesem Codeabschnitt werden verschiedenste Parameter für den Zellulären Automaten definiert:

- `number_of_columns` definiert die Spaltenanzahl des Gitternetzes
- `number_of_rows` definiert die Zeilenanzahl des Gitternetzes
- `density` definiert die Dichte der Bäume (in Prozent, z.B. 80% der Fläche sind mit Bäumen besetzt - 0.80)
- `pe` definiert die Wahrscheinlichkeit, dass ein Baum durch einen brennenden Nachbarbaum Feuer fängt
- `pb` definiert die Wahrscheinlichkeit, dass ein Baum durch einen Blitz getroffen wird
- `pw` definiert die Wahrscheinlichkeit, dass auf einem leeren Feld ein Baum nachwächst

Des Weiteren werden für jede Kategorie Listen erstellt, die ein Feld einnehmen kann (`forest_fields` - Felder mit Bäumen, `fire_fields` - Felder, auf denen ein Feuer ist, `free_fields` - Felder, auf denen weder ein Feuer noch ein Baum ist).

Ebenso wird die Matrix `forest` definiert. Diese dient als Grundlage für das Gitternetz des Automaten. Diese wird initial mit Nullen befüllt.

Anschließend wird die Liste `free_fields` mit allen vorhandenen Koordinaten befüllt. Hierbei handelt es sich immer um ein Tupel, das zuerst die Zeile beinhaltet und als zweites die Spalte.

[2]:
```
# define parameters for grid layout
number_of_columns = 20
number_of_rows = 20

# density of trees in the grid
density = 0.8

# possibility for fire
pe = 0.5

# possibility for lighting
pb = 0.0001

# possibility for a new tree
pw = 0.0001

# define lists for each category
forest_fields = []
fire_fields = []
free_fields = []


# create forest matrix
forest = random_matrix(ZZ, nrows = number_of_rows, ncols = number_of_columns, x␣
 ↪= 1)

# fill list with free fields with all possible values -> forest has no trees
for col in range(0, number_of_columns):
    for row in range(0, number_of_rows):
        free_fields.append((row, col))
```

1.5.2 Arbeitsschritt 2: Erstellen von Methoden und Durchführen der Simulation

printMatrix()

Die Methode `printMatrix(forest)` plottet die übergebene Matrix `forest`. Dabei gibt es folgende farbliche Unterteilung: - leeres Feld: weiß - Feld mit Baum: grün - brennendes Feld: rot

Die Einteilung, ob ein Feld leer ist, einen Baum beinhaltet oder brennt, beruht auf den Werten in der Matrix. Dafür wurde folgende Wertezuweisung vorgenommen: - -1: Feuer - 0: leeres Feld - 1: Feld mit Baum

[3]:
```
def printMatrix(forest):
    # define colors
    color_pool = ['#FF0000', '#FFFFFF', '#006400' ]
```

```
    # convert colors to color map
    color_map = matplotlib.colors.ListedColormap(color_pool[0:3])

    # return matrix plot
    return matrix_plot(forest, cmap = color_map, colorbar = True, vmin = -1,␣
 ↪vmax = 1)
```

createTree()

Die Methode `createTree(coordinates, forest, forest_fields, free_fields)` wird dazu benötigt, um an den übergebenen Koordinaten einen Baum zu erstellen. Im gleichen Zug wird auch dieses Feld aus den leeren Feldern entfernt.

[4]:
```
def createTree(coordinates, forest, forest_fields, free_fields):

    # set matrix value
    forest[coordinates[0], coordinates[1]] = 1

    # add to forest list
    forest_fields.append(coordinates)

    # remove from free fields list
    free_fields.remove(coordinates)
```

createFire()

Die Methode `createFire(coordinates, forest, forest_fields, fire_fields)` wird dazu benötigt, um an den übergebenen Koordinaten ein Feuer zu zünden. Ebenso wird auch hier gleich die Liste mit Bäumen angepasst.

[5]:
```
def createFire(coordinates, forest, forest_fields, fire_fields):

    # set matrix value
    forest[coordinates[0], coordinates[1]] = -1

    # remove from forest
    forest_fields.remove(coordinates)

    # add to fire list
    fire_fields.append(coordinates)
```

createEmptyField()

Die Methode `createEmptyField(coordinates, forest, free_fields)` wird dazu benötigt, um an den übergebenen Koordinaten ein leeres Feld zu erstellen.

[6]:
```
def createEmptyField(coordinates, forest, free_fields):

    # set matrix value
    forest[coordinates[0], coordinates[1]] = 0
```

```
    # add to free fields list
    free_fields.append(coordinates)
```

createTrees()

Die Methode `createTrees(density, free_fields, forest_fields, number_of_cols, number_of_rows, forest)` wird dazu benötigt um das Gitternetz initial zufällig mit Bäumen zu befüllen. Die Anzahl der Bäume richtet sich dabei nach dem in `density` festgelegten Prozentwert.

Zuerst wird hierfür die Anzahl der Bäume errechnet (`number_of_trees`).

Anschließend werden zufällig aus den leeren Feldern Felder entnommen und auf ihnen ein Baum gesetzt. Hierfür werden die bereits oben definierten Methoden verwendet.

[7]:
```
def createTrees(density, free_fields, forest_fields, number_of_cols,␣
↪number_of_rows, forest):

    # calculate amount of trees based on the density
    number_of_trees = number_of_cols * number_of_rows * density

    # create the amount of trees randomly
    while (number_of_trees > 0):

        # get a random index
        index = random.randint(0, len(free_fields)-1)

        # get the coordinates
        coordinates = free_fields[index]

        # create tree
        createTree(coordinates, forest, forest_fields, free_fields)
        number_of_trees -= 1
```

createInitialFire()

Die Methode `createInitialFire(forest_fields, fire_fields, forest)` erstellt das erste Feuer. Dieses wird zufällig aus den Feldern mit Bäumen ausgewählt und entzündet.

[8]:
```
def createInitialFire(forest_fields, fire_fields, forest):

    # get random index from forest list
    index = random.randint(0, len(forest_fields)-1)

    # get coordinates
    coordinates = forest_fields[index]

    # create fire
    createFire(coordinates, forest, forest_fields, fire_fields)
```

spreadFire()

Die folgende Methode `spreadFire(forest_fields, fire_fields, free_fields, pe, forest, oneSystem, amount_of_cols, amount_of_rows)` sorgt dafür, dass sich das Feuer ausbreiten kann. Hierfür werden zuerst alle bestehenden Brände durchgegangen und die umliegenden Felder ermittelt. Wenn auf diesen ein Baum steht, kommen sie in die engere Auswahl.

In einem weiteren Schritt wird dann aus den Bäumen, die in Gefahr sind, die Anzahl für neue Feuer ermittelt. Dies basiert auf der festgelegten Wahrscheinlichkeit für ein neues Feuer (`pe`).

Diese Felder werden dann wieder zufällig aus den möglichen neuen Bränden entnommen und mit der Hilfsmethode `createFire` entzündet.

1.5.3 Arbeitsschritt 4 - Weitere Implementierung für Randfelder

Wenn `oneSystem` wahr, ist, wird eine andere Behandlung der Randfelder vorgenommen.

Diese funktioniert wie folgt:

Ein Baum in Zeile 0 und Spalte 2 kann nicht nur den unteren, den linken und den rechten Baum entzünden, sondern auch den Baum auf der anderen Seite des Gitters. In diesem Fall den Baum in der maximalen Zeile und Spalte 2. Hierfür wird geschaut, ob eines der Felder, die das Feuer umgeben, außerhalb des Gitter liegt. Wenn dies zutrifft, wird das Feld so angepasst, dass dies auf ein Feld am gegenüberliegenden Gitterrand zeigt. Dies gilt nicht nur von oben nach unten und anderes herum, sondern auch für von links nach rechts und von rechts nach links.

Wenn `oneSystem` nicht wahr ist, dann werden die Ränder ganz normal behandelt.

[9]:
```
def spreadFire(forest_fields, fire_fields, free_fields, pe, forest, oneSystem,␣
↪amount_of_cols, amount_of_rows):

    # copy fire fields
    temp = fire_fields.copy()


    # reset fire list
    fire_fields.clear()

    # create list for possible fires
    possible_fire_fields = []

    # loop through all fires
    for fire in temp:

        # get fields around fire field
        top_field = (fire[0]-1, fire[1])
        bottom_field = (fire[0]+1, fire[1])
        right_field = (fire[0], fire[1]+1)
        left_field = (fire[0], fire[1]-1)

        # switch to enable a different implementation for border fields
```

```
        # -> function: a field in row 0 and column 2,
        # can create a fire in row max_row and column 2, also vice versa
        # and from the left side to the right side

        if oneSystem:

            # check if possible field is besides the border
            # if yes -> adjust coordinates to the other side
            # if no -> do nothing

            if left_field[1] < 0:
                left_field = (left_field[0], amount_of_cols-1)

            if right_field[1] == amount_of_cols:
                right_field = (right_field[0], 0)

            if top_field[0] < 0:
                top_field = (amount_of_rows-1, top_field[1])

            if bottom_field[0] == amount_of_rows:
                bottom_field = (0, bottom_field[1])

        # check if possible field is a fire field and is not allready in the
        # possible fire fields list
        # if yes -> add to possible fire fields list
        # if no -> do nothing

        if left_field in forest_fields and left_field not in␣
↪possible_fire_fields:
            possible_fire_fields.append(left_field)

        if right_field in forest_fields and right_field not in␣
↪possible_fire_fields:
            possible_fire_fields.append(right_field)

        if top_field in forest_fields and top_field not in possible_fire_fields:
            possible_fire_fields.append(top_field)

        if bottom_field in forest_fields and bottom_field not in␣
↪possible_fire_fields:
            possible_fire_fields.append(bottom_field)

        createEmptyField(fire, forest, free_fields)

    # calculate the amount of new fire fields based on the possiblity for a new␣
↪fire
    number_of_new_fires = len(possible_fire_fields) * pe
```

```
    # create the correct amount of fires
    while(number_of_new_fires > 0):

        # pick a random index out of the possible fire fields list
        index = random.randint(0, len(possible_fire_fields)-1)

        # get coordinates
        new_fire = possible_fire_fields[index]

        # create fire
        createFire(new_fire, forest, forest_fields, fire_fields)

        # remove from possible fire list
        possible_fire_fields.remove(new_fire)

        number_of_new_fires -= 1

    # clear possible fire list for the next run
    possible_fire_fields.clear()
```

1.5.4 Arbeitsschritt 5 - Blitzeinschlag

createLightningFire()

Die Methode `createLightningFire(forest_fields, fire_fields, free_fields, pb)` entzündet mit der Wahrscheinlichkeit `pb` einen Baum durch einen Blitz.

[10]:
```
def createLightningFire(forest_fields, fire_fields, free_fields, pb):

    # calculate the amount of fires through lighting based on the possibility␣
 ↪for a lighting
    number_lightning_strikes = len(forest_fields) * pb

    # create the right amount of fires
    while(number_lightning_strikes > 0):

        # pick a random index out of the forest list
        index = random.randint(0, len(forest_fields)-1)

        # get coordinates
        new_fire = forest_fields[index]

        # create a fire
        createFire(new_fire, forest, forest_fields, fire_fields)
        number_lightning_strikes -= 1
```

1.5.5 Arbeitsschritt 6 - Erweiterungen

growTree()

Die Methode `growTree(pw, free_fields, forest_fields, forest)` lässt mit einer gewählten Wahrscheinlichkeit Bäume auf leeren Feldern wachsen.

[11]:
```
def growTree(pw, free_fields, forest_fields, forest):

    # calculate the right amount of new trees based on the possibility for a␣
 ↪new tree
    number_of_trees = len(free_fields) * pw

    # create the right amount of new trees
    while(number_of_trees > 0):

        # pick a random index out of the free fields list
        index = random.randint(0, len(free_fields)-1)

        # get the coordinates
        coordinates = free_fields[index]

        # create a tree
        createTree(coordinates, forest, forest_fields, free_fields)
        number_of_trees -= 1
```

1.5.6 MAIN PROGRAMM

Dieser Codeblock lässt die Simulation einmal durchlaufen und gibt das Ergebnis als animierten Plot aus. Hierfür wird zuerst ein Setup erstellt und anschließend, bis keine Bäume mehr verfügbar sind oder die Obergrenze von 100 Zeiteinheiten erreicht ist, durchgeführt.

Für den animierten Plot werden die Plots nach jedem Zeitschritt abgespeichert und am Ende der Schleife einmal animiert ausgegeben.

Die Obergrenze von 100 Zeiteinheiten wurde festgelegt, da sonst die Möglichkeit für eine Endlosschleife besteht, wenn die Bäume im gleichen Tempo nachwachsen und verbrennen.

Die Animation wird in Form eines GIFs im Ordner dieser Datei abgelegt.

[12]:
```
current_time = 1

# inital setup -> create trees and a intial fire
createTrees(density, free_fields, forest_fields, number_of_columns,␣
 ↪number_of_rows, forest)
createInitialFire(forest_fields, fire_fields, forest)

# switch between border fields implementation
oneSystem = True
```

```
# collect all plots in a list
a = []

# run as long as we have forest fields, we have fires and the current time is␣
 ↪less than 100
# without the last check it could be possible that we run into an infinite␣
 ↪loop, because trees grow
# as fast as they durn down.

while(len(forest_fields) != 0 and current_time < 100):

    # let the fire spread
    spreadFire(forest_fields, fire_fields, free_fields, pe, forest, oneSystem,␣
 ↪number_of_columns, number_of_rows)

    # create a lightning
    createLightningFire(forest_fields, fire_fields, free_fields, pb)

    # create new trees
    growTree(pw, free_fields, forest_fields, forest)

    # add current matrix to plot list
    a.append(printMatrix(forest))

    current_time += 1

# animate all plots in the list and save it as file
b = animate(a)
b
b.gif(savefile='simulation.gif', delay=5, iterations=0)
```

[12]: Animation with 99 frames

1.6 ## Ergebnisse und Interpretation der graphischen Simulation

Unser Ergebnis hängt stark von den gewählten Werten der Wahrscheilichkeiten ab. Je höher die Wahrscheinlichkeit *pe* und *pb*, desto schneller brennt der Wald ab. Durch die Methode `growTree()` kann es sein, dass der Wald nie abbrennt oder sogar größer wird, was natürlich nicht realistisch ist. Hier müsste eine maximale Wahrscheinlichkeit für *pw* festgelegt werden.

Des Weiteren sind die Wahrscheinlichkeiten für den Blitz und die zweite Implementierung der Randfelder weitere "Brandbeschleuniger". Dies sieht man auch unten anhand der verschiedenen Graphen.

1.6.1 Aufgabenstellung 3 - Statistikfunktionen

setupForAnalytics()

`setupForAnalytics()` erstellt ein neues Setup für die Simulation. Hierfür werden vorerst alle Parameter zurückgesetzt. Anschließend werden wieder zufällig Bäume auf dem Gitternetz generiert und ein Startfeuer entzündet.

Diese Methode dient dazu, dass man für die Statistikfunktionen eine neues Setup erzeugen kann.

[13]:
```
def setupForAnalytics():

    # define lists for each category and clear them
    forest_fields.clear()
    fire_fields.clear()
    free_fields.clear()


    # create forest matrix
    forest = random_matrix(ZZ, nrows = number_of_rows, ncols =␣
 ↪number_of_columns, x = 1)

    # fill list with free fields with all possible values
    for col in range(0, number_of_columns):
        for row in range(0, number_of_rows):
            free_fields.append((row, col))

    # create inital trees and fire
    createTrees(density, free_fields, forest_fields, number_of_columns,␣
 ↪number_of_rows, forest)
    createInitialFire(forest_fields, fire_fields, forest)
```

runAnalytics()

`runAnalytics(pe, pb, pw, forest_fields, fire_fields, free_fields, forest, number_of_columns, number_of_rows, oneSystem)` lässt die Simulation über **ein** Setup laufen. Hierbei wird das grundlegende Setup verändert, daher ist es wichtig, beim Aufruf dieser Methode nur Kopien der Listen und der Matrix zu übergeben. Somit ist sichergestellt, dass mehrere Analysevorgänge mit dem identischen Setup laufen können.

Des Weiteren wird eine `analytics` Liste geführt, welche die bis zu einem Zeitpunkt verbrannten Bäume (Anteil der Startmenge) aufsummiert. Dies wird dann anschließend an den Aufrufer zurückgegeben.

[14]:
```
def runAnalytics(pe, pb, pw, forest_fields, fire_fields, free_fields, forest,␣
 ↪number_of_columns, number_of_rows, oneSystem, density):
    current_time = 1

    amount_of_trees = number_of_columns * number_of_rows * density

    # time 0 - one tree burns
    analytics = [(0, 100 / amount_of_trees * 1)]
```

```
    # run as long as we have forest fields, we have fires and the current time␣
↪is less than 100
    # without the last check it could be possible that we run into an infinite␣
↪loop, because trees grow
    # as fast as they durn down.

    while(len(forest_fields) != 0 and current_time < 100):

        # spread fire, create a lightning and new trees
        spreadFire(forest_fields, fire_fields, free_fields, pe, forest,␣
↪oneSystem, number_of_columns, number_of_rows)
        createLightningFire(forest_fields, fire_fields, free_fields, pb)
        growTree(pw, free_fields, forest_fields, forest)

        # save current status of burned trees in analytics list
        current_status = (current_time, (100 / amount_of_trees *␣
↪len(fire_fields)) + analytics[current_time-1][1])
        analytics.append(current_status)

        current_time += 1

    return analytics
```

1.6.2 STATISTIKFUNKTIONEN I

Dieser Codeblock generiert die Analysen zur Simulation.

Hierbei wird die Simulation mit 5 verschiedenen Setups wiederholt. Das grundlegende Setup wird dabei nicht verändert, somit starten alle Simulationen auf der gleichen Datenbasis. Ziel ist es, zu vergleichen, wie sich der Waldbrand in diesem Setup mit unterschiedlichen Wahrscheinlichkeiten ausbreitet. Ebenso werden auch die Anteile des abgebrannten Waldes in Abhängigkeit zur Zeit als Plot ausgegeben.

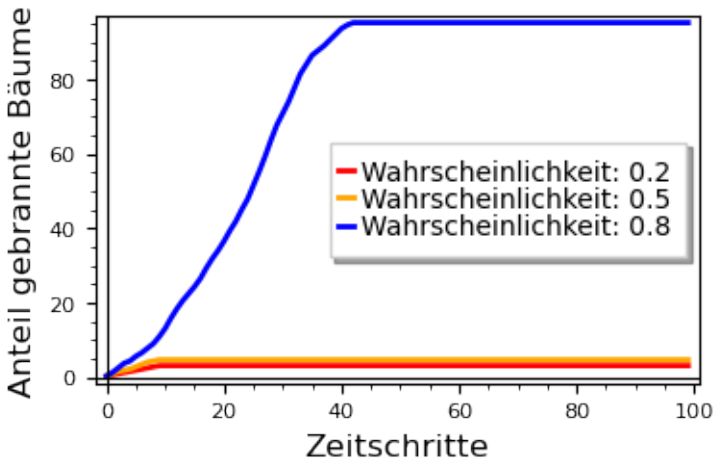
Anpassungen können an der Wahrscheinlichkeit für Feuer, für einen Blitz und für einen neuen Baum vorgenommen werden. Ebenso kann man entscheiden, welche Implementierung der Randfelder man benutzen möchte.

[15]:
```
# run analytics -> 3 possibilities on one setup for 5 different setups

# amount of different setups
run = 5

# define possibilities

# --- setup 1
pe1 = 0.20
pb1 = 0
pw1 = 0
```

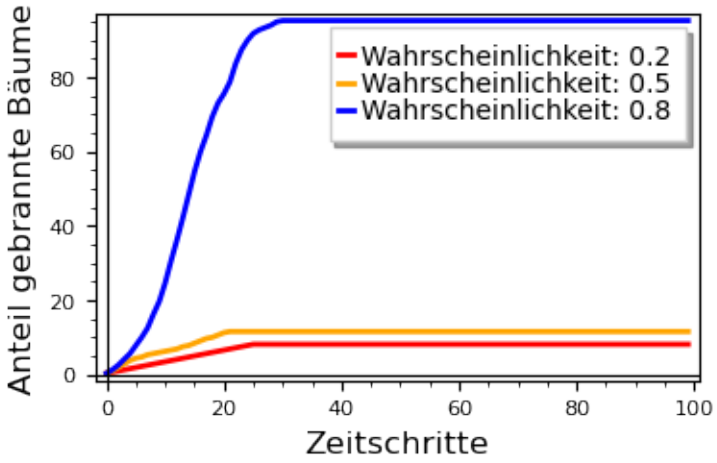
```
oneSystem1 = False

# --- setup 2
pe2 = 0.50
pb2 = 0
pw2 = 0
oneSystem2 = False

# --- setup 3
pe3 = 0.80
pb3 = 0
pw3 = 0
oneSystem3 = False

# save each pe category in one graphic
gpe1 = Graphics()
gpe2 = Graphics()
gpe3 = Graphics()


while run > 0:

    # create intial setup
    setupForAnalytics()

    # run 3 simulations with different possibilities -> transmit only copies so␣
 ↪that each possibiliy have the same initial setup
    analytics_20 = runAnalytics(pe1, pb1, pw1, forest_fields.copy(),␣
 ↪fire_fields.copy(), free_fields.copy(), copy(forest), number_of_columns,␣
 ↪number_of_rows, oneSystem1, density)
    analytics_50 = runAnalytics(pe2, pb2, pw2, forest_fields.copy(),␣
 ↪fire_fields.copy(), free_fields.copy(), copy(forest), number_of_columns,␣
 ↪number_of_rows, oneSystem2, density)
    analytics_80 = runAnalytics(pe3, pb3, pw3, forest_fields.copy(),␣
 ↪fire_fields.copy(), free_fields.copy(), copy(forest), number_of_columns,␣
 ↪number_of_rows, oneSystem3, density)

    # collect all three lists in one plot
    g = Graphics()

    p20 = list_plot(analytics_20, plotjoined=True, frame=True,␣
 ↪axes_labels=["Zeitschritte","Anteil gebrannte Bäume"], fontsize=8,␣
 ↪color='red', thickness=2, legend_label=f'Wahrscheinlichkeit: {str(pe1)[0:
 ↪3]}')
```

```
    p50 = list_plot(analytics_50, plotjoined=True, frame=True,␣
 ↪axes_labels=["Zeitschritte","Anteil gebrannte Bäume"], fontsize=8,␣
 ↪color='orange', thickness=2, legend_label=f'Wahrscheinlichkeit: {str(pe2)[0:
 ↪3]}')
    p80 = list_plot(analytics_80, plotjoined=True, frame=True,␣
 ↪axes_labels=["Zeitschritte","Anteil gebrannte Bäume"], fontsize=8,␣
 ↪color='blue', thickness=2, legend_label=f'Wahrscheinlichkeit: {str(pe3)[0:
 ↪3]}')

    g = g + p20 + p50 + p80
    g.set_legend_options(loc='best')

    p20.legend(False)
    p50.legend(False)
    p80.legend(False)

    gpe1 += p20
    gpe2 += p50
    gpe3 += p80

    # print in which setup we are
    print('Setup: ' + str(6 - run))

    # print plot
    g.show(figsize=4)

    run -= 1

# print each possibility category
gpe1.show(figsize=4)
gpe2.show(figsize=4)
gpe3.show(figsize=4)
```

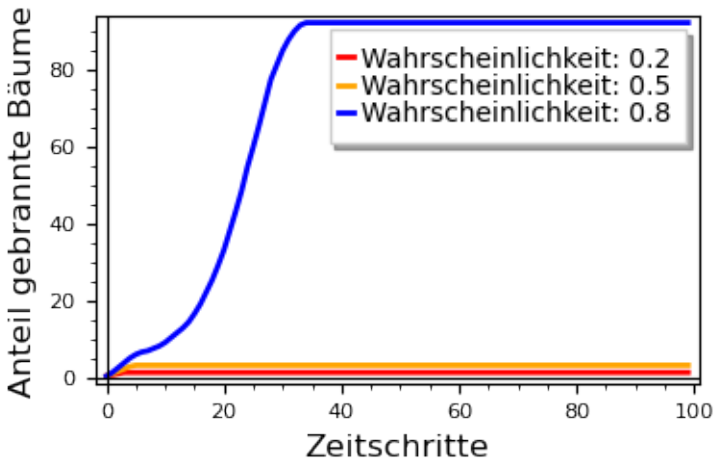
```
Setup: 1
```



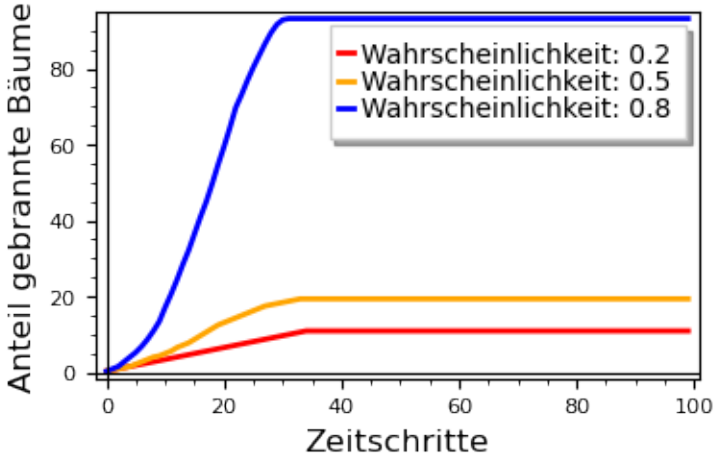
Setup: 2



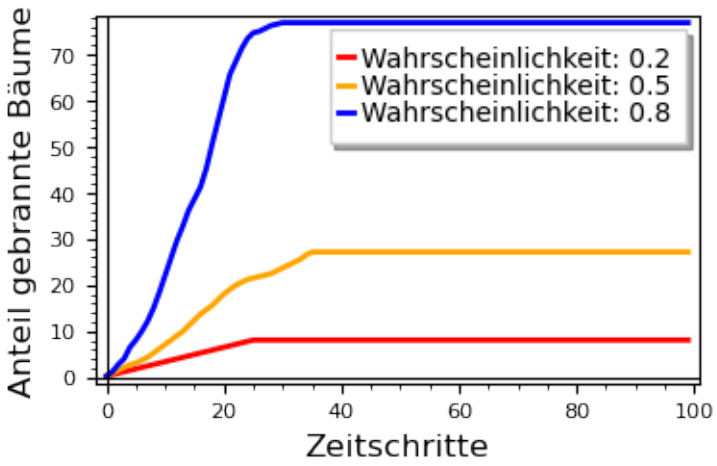
Setup: 3

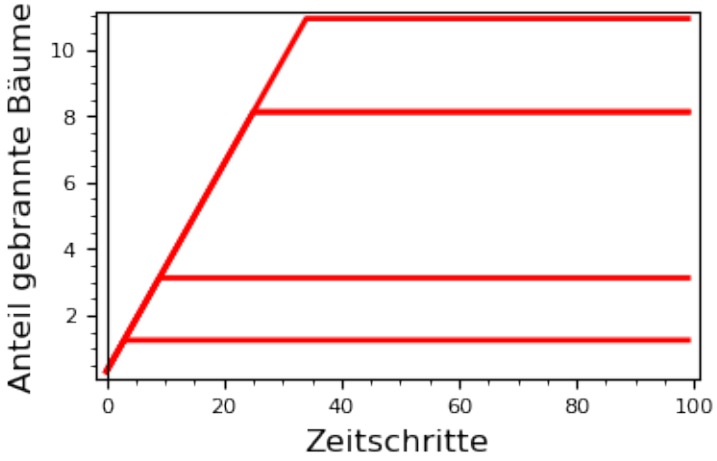


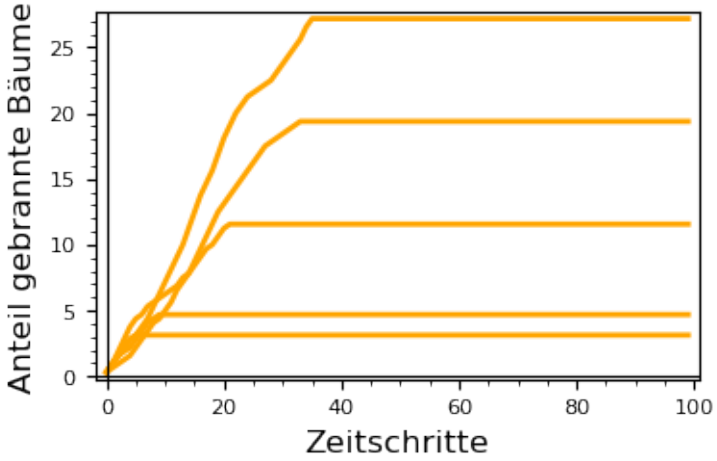
Setup: 4

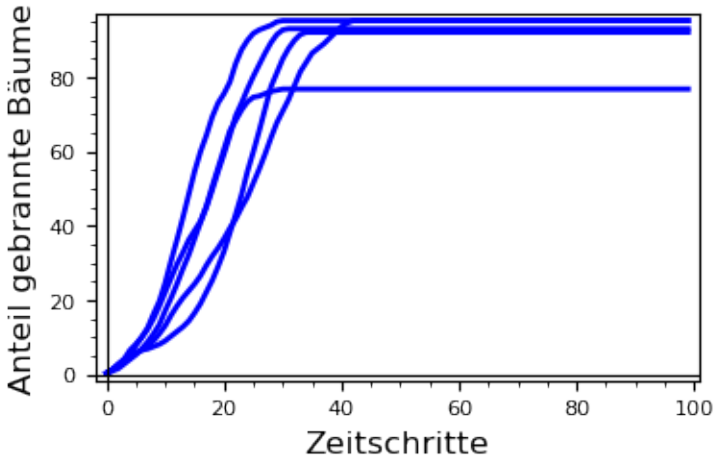


Setup: 5


Wahrscheinlichkeit: 0.2
Wahrscheinlichkeit: 0.5
Wahrscheinlichkeit: 0.8
Anteil gebrannte Bäume
Zeitschritte
0
10
20
30
40
50
60
70
0
20
40
60
80
100


Anteil gebrannte Bäume
Zeitschritte
2
4
6
8
10
0
20
40
60
80
100


Anteil gebrannte Bäume
Zeitschritte
0
5
10
15
20
25
0
20
40
60
80
100



1.7 ## Ergebnisse und Deutung STATISTIKFUNKTION I

Durch die Implementierung von mehreren Graphen kann man feststellen, dass die Ergebnisse stark variieren, was aufgrund der unterschiedlichen Ausgangspositionen des Feuers und unterschiedlichen Wäldern passiert. Was besonders auffällt, ist der große Abstand und die starke Steigung des Graphens mit der Wahrscheinlichkeit $pe = 0.8$. Die Graphen weisen generell darauf hin, dass je höher die Wahrscheinlichkeit pe ist, desto schneller und höher steigt der Anteil der abgebrannten Bäume.

Aber generell kann man sagen, dass sich die Graphen entsprechend ihrer Wahrscheinlichkeit ordnen. Ebenso ist es sehr interessant zu sehen, dass bei 20% und 50% die Startposition des Feuers einen großen Unterschied macht.

Schaltet man Stück für Stück die Zusatzoptionen ein (Blitz, Nachwachsen von Bäumen und die zweite Implementierung der Randfelder) wird das Ergebnis immer genauer, da der Ausgang kaum noch von der gewählten Wahrscheinlichkeit, dass ein Baum anfängt zu brennen, bestimmt wird. So rücken mit den Zusatzoptionen die Graphen immer näher zusammen. Aber man kann sagen, dass alle drei Optionen den Brand eher beschleunigen, anstatt ihn auszubremsen. Dies ist vor allem bei der Option Bäume nachwachsen zu lassen, sehr interessant.

1.7.1 STATISTIKFUNKTION II

In diesem Abschnitt wird eine weitere Analyse durchgeführt. Hier wird nämlich der Anteil abgebrannter Bäume zum Zeitpunkt 10 im Bezug auf die Wahrscheinlichkeit dargestellt.

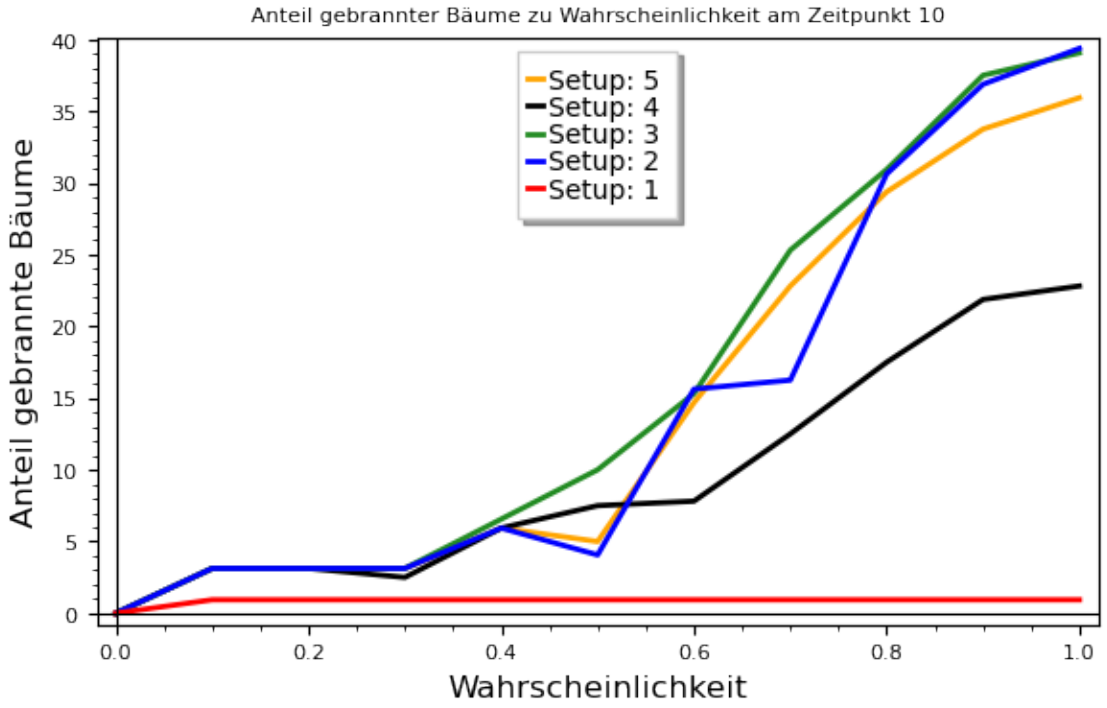
Hierfür werden 5 verschiedene Setups generiert und jedes mit jeder Wahrscheinlichkeit von 0 - 1 für ein Feuer an einem Nachbarfeld simuliert. Anschließend wird die Anzahl an abgebrannten Bäumen am Zeitpunkt 10 ermittelt und gespeichert.

Im Plot sieht man dann den Anteil, wie viele Bäume in diesem Setup mit der entsprechend gewählten Wahrscheinlichkeit am Zeitpunkt 10 verbrannt sind. Dabei ist jedes Setup in einer anderen Farbe

dargestellt.

[16]:
```
# amount of runs
run = 5

# graphic for plot
g = Graphics()

# different colors
colors = ['red', 'blue', 'forestgreen', 'black', 'orange']

# define possiblities
pb = 0
pw = 0
oneSystem = False

while run > 0:

    # create a initial setup
    setupForAnalytics()

    # with possibility 0, 0 trees burn down at timestamp 10
    amount_to_pe = [(0,0)]

    # loop through all possibilites -> starting with 0.1 - ends with 1 - steps␣
 ↪0.1
    for i in range(1, 11):

        # calculate possibility
        pe = i / 10

        # run analytics
        analytics = runAnalytics(pe, pb, pw, forest_fields.copy(), fire_fields.
 ↪copy(), free_fields.copy(), copy(forest), number_of_columns, number_of_rows,␣
 ↪oneSystem, density)

        # check if run takes longer than 10 time units
        if(len(analytics) < 10):

            # if not -> take the last amount of burned trees
            amount_to_pe.append((pe, analytics[len(analytics)-1][1]))

        else:

            # if yes -> take amount of burned trees at timestamp 10
            amount_to_pe.append((pe, analytics[9][1]))
```

```
    # add list to plot
    plot = list_plot(amount_to_pe, plotjoined=True, title="Anteil gebrannter␣
 ↪Bäume zu Wahrscheinlichkeit am Zeitpunkt 10", frame=True,␣
 ↪axes_labels=["Wahrscheinlichkeit","Anteil gebrannte Bäume"], fontsize=8,␣
 ↪color=colors[run - 1], thickness=2, legend_label=f'Setup: {str(run)}')
    plot.set_legend_options(loc='upper center')
    g = g + plot

    run -= 1

# show plot
g.show()
```



1.8 ## Ergebnisse und Deutung STATISTIKFUNKTION II

Der Graph zeigt, dass je höher die Wahrscheinlichkeit *pe* ist, desto höher ist der Anteil der abgebrannten Bäume. Da es sich um Wahrscheinlichkeiten handelt, können die Werte stark variieren. Was auffällt ist, dass sich der Anteil der abgebrannten Bäume pro Setup an den höheren Wahrscheinlichkeiten am meisten unterscheidet.

Auch hier hat die Implementierung der Randfelder und die Beschleunigung durch Blitze starke Auswirkungen. So sorgt z.B. eine sehr hohe Wahrscheinlichkeit, dass Bäume durch Blitze getroffen

werden dazu, dass die Graphen sehr nahe, fast zu einem Graphen zusammenkommen.

Die Wahrscheinlichkeit neue Bäume zu erzeugen, wirkt sich ebenso auf die Graphen aus. Denn, anders als erwartet, erhöht sie sogar den Anteil abgebrannter Bäume am Zeitpunkt 10. Die Erhöhung zeigt sich um ungefähr 2-3% (bei *pw* von 5%). Dies hat damit zu tun, dass das Feuer sich nicht von alleine ausbremst, sondern durch die neuen Bäume mehr Rohstoffe bekommt, um weiter zu brennen.

Die weitere Implementierung der Randfelder wirkt sich zugunsten des Feuers aus. Sie steigert den Anteil um weitere 5%.

1.9 ## Arbeitsaufteilung

Für dieses Projekt haben wir folgende Arbeitsauflteilung vorgenommen:

- Aufgabe 1 - 4: Implementierung des SageMath Skriptes: Simon Gärtner
- Aufgabe 5 - 6: Implementierung des SageMath Skriptes: Tobias Blümlhuber

Dokumentation - gleiche Einteilung auch für das Referat: - Problembeschreibung und mathematisches Modell: Tobias Blümlhuber - Implementierung im SageMath Skript: Simon Gärtner / Silvan Kron - Deutung und Analyse der Ergebnisse und Arbeitsaufteilung: Silvan Kron

[ ]: