

## BERICHT ZUR GLÜHWÜRMCHEN-SIMULATION

Narek Grigoryan

Matrikelnummer: 1547616

Kurs: Software Architecture for Enterprises (SA4E) – Übung 1

### Einleitung

Dieser Bericht dokumentiert die Entwicklung einer Simulation synchronisierter Glühwürmchen, die auf Basis der Aufgabenstellung im Kurs "Software Architecture for Enterprises" erstellt wurde.

Die Simulation basiert auf dem Kuramoto-Modell und umfasst zwei zentrale Aufgaben:

1. Aufgabe 1: Monolithische Implementierung einer Simulation in einer Torus-Struktur, die Threads verwendet.
2. Aufgabe 2: Erweiterung zur verteilten Simulation mittels gRPC.

### Torus-Struktur

Die Glühwürmchen-Simulation verwendet eine Torus-Struktur, in der die Glühwürmchen auf einem 2D-Gitter angeordnet sind.

Ein Torus verbindet die Ränder des Gitters zyklisch, sodass jedes Glühwürmchen vier Nachbarn hat (oben, unten, links, rechts), auch an den Rändern des Gitters. Diese Struktur ermöglicht eine kontinuierliche Nachbarschaftsinteraktion ohne Begrenzungen.

### Kuramoto-Modell

Das Kuramoto-Modell dient als Grundlage für die Synchronisation der Glühwürmchen. Es beschreibt die Dynamik gekoppelter Oszillatoren, die durch folgende Gleichung dargestellt wird:

$$d\theta/dt = \omega + (K / N) * \sum \sin(\theta_j - \theta_i)$$

Hierbei sind:

- $\theta$ : Phase des Oszillators (Glühwürmchen),
- $\omega$ : Natürliche Frequenz,
- $K$ : Kopplungskonstante, die die Stärke der Interaktion bestimmt,
- $N$ : Anzahl der Nachbarn.

In der Simulation wird jede Phase in eine Farbe umgewandelt, sodass Synchronisation visuell als Angleichung der Helligkeiten sichtbar wird.

## Aufgabe 1: Monolithische Simulation

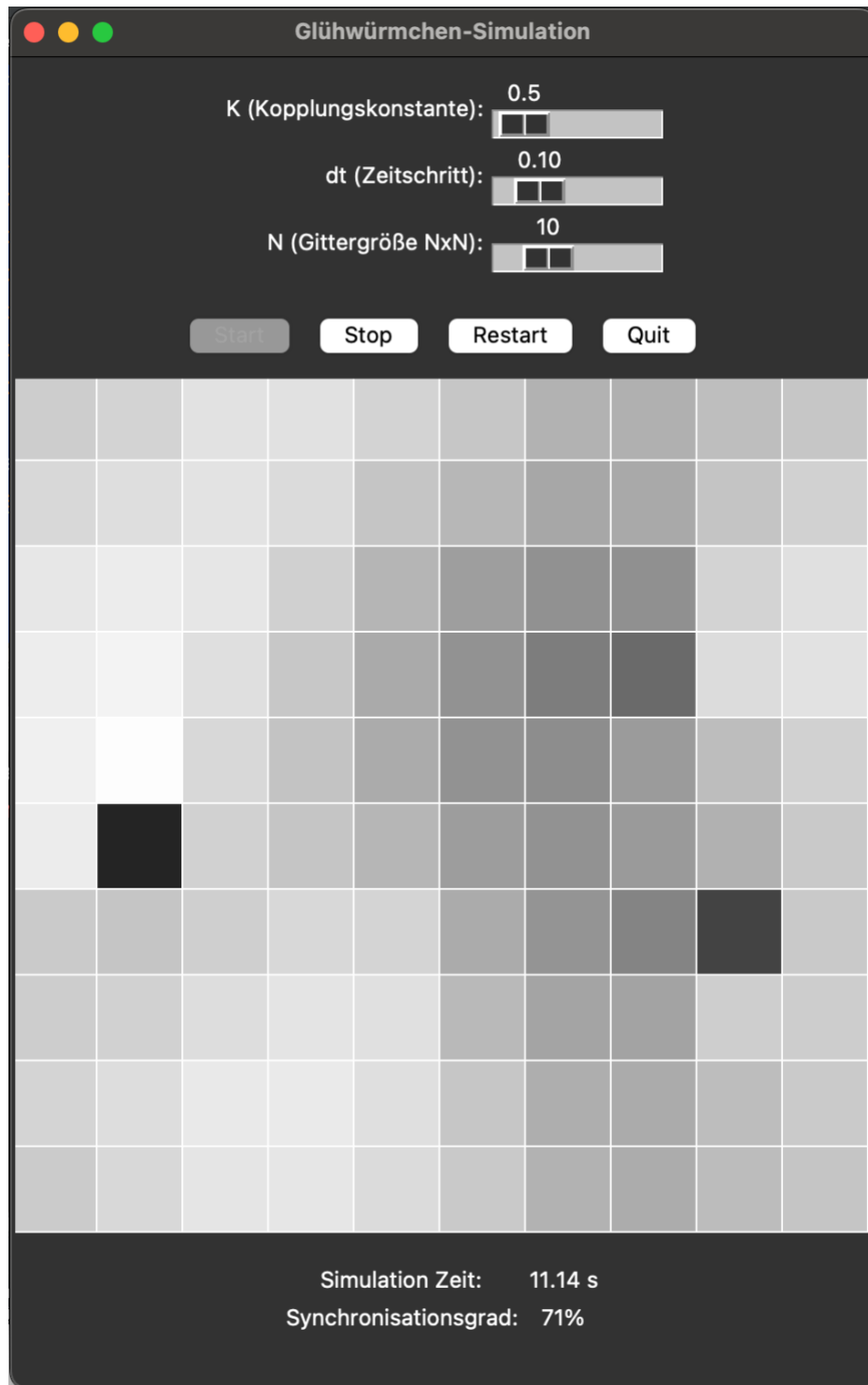
In Aufgabe 1 wurde eine monolithische Simulation mit mehreren Threads implementiert, wobei jedes Glühwürmchen als Thread modelliert ist. Die Glühwürmchen beeinflussen sich gegenseitig basierend auf dem Kuramoto-Modell. Die GUI zeigt die Simulation in Echtzeit.

### Beschreibung der GUI

Die grafische Benutzeroberfläche (GUI) wurde mit `Tkinter` entwickelt und zeigt die folgenden Elemente:

- Parametersteuerung: Benutzer können die Kopplungskonstante `K`, den Zeitschritt `dt` und die Gittergröße `N` anpassen.
- Simulationsfenster: Ein Gitter von Rechtecken repräsentiert die Glühwürmchen. Die Helligkeit jedes Rechtecks zeigt die aktuelle Phase.
- Statusanzeigen: Die GUI zeigt die Simulationszeit und den Synchronisationsgrad in Prozent.

Die folgende Abbildung zeigt die GUI während einer Simulation:



## Implementierung

Die Implementierung der Simulation basiert auf zwei zentralen Modulen:

1. Firefly-Klasse (Firefly\_1.py): Repräsentiert ein einzelnes Glühwürmchen und implementiert die Logik zur Phasenberechnung.
2. FireflySimulation-Klasse (Firefly\_1Simulation.py): Verwaltert die Simulation, stellt die GUI bereit und steuert die Interaktion zwischen den Glühwürmchen.

### Firefly-Klasse

Die Firefly-Klasse implementiert die Logik eines Glühwürmchens als Thread. Jede Instanz hat eine Phase, natürliche Frequenz und Methoden zur Interaktion mit ihren Nachbarn. Wichtige Methoden:

- ``run``: Aktualisiert die Phase basierend auf dem Kuramoto-Modell.
- ``get_neighbors``: Berechnet die Nachbarn im Torus.
- ``phase_to_color``: Wandelt die Phase in eine Helligkeit um.

### FireflySimulation-Klasse

Die FireflySimulation-Klasse verwaltet die gesamte Simulation. Sie erstellt das Gitter der Glühwürmchen, startet die Threads und aktualisiert die GUI.

Wichtige Funktionen:

- ``initialize_fireflies``: Initialisiert das Gitter der Glühwürmchen und verknüpft sie mit der GUI.
- ``calculate_sync_degree``: Berechnet den Synchronisationsgrad basierend auf den Phasen.

***Disclaimer: Der Synchronisationsgrad bleibt oft bei 99%, da minimale numerische Rundungsfehler in den Phasenberechnungen auftreten. Diese Abweichungen verhindern oft dass exakt 100% erreicht wird. In der Praxis bedeutet 99% nahezu vollständige Synchronisation.***

- ``update_status``: Aktualisiert kontinuierlich die Simulationszeit und den Synchronisationsgrad in der GUI.

## Zusammenfassung Aufgabe 1

Die monolithische Simulation zeigt erfolgreich, wie die Glühwürmchen ihre Phasen synchronisieren. Die GUI ermöglicht es, den Prozess intuitiv zu beobachten und die Auswirkungen von Parametern wie ``K`` und ``dt`` zu untersuchen.

## Aufgabe 2: Verteilte Simulation

### Einleitung

In Aufgabe 2 wurde die monolithische Simulation aus Aufgabe 1 erweitert, um die Glühwürmchen als unabhängige, verteilte Prozesse zu implementieren, die über ein Netzwerk miteinander kommunizieren. Hierbei wird das Kommunikationsprotokoll gRPC verwendet, um den Synchronisationsprozess zwischen den Glühwürmchen zu steuern.

Die Simulation verdeutlicht die dezentrale Natur der Synchronisation, bei der jedes Glühwürmchen als eigenständiger Server fungiert und seine Phase regelmäßig mit seinen Nachbarn austauscht.

### Kommunikationsprotokolle: Apache Thrift und gRPC

Apache Thrift und gRPC sind Kommunikationsprotokolle, die es ermöglichen, verteilte Systeme zu entwickeln, bei denen verschiedene Prozesse über Netzwerke miteinander kommunizieren.

- Beide Protokolle verwenden Remote Procedure Calls (RPC), um Funktionen über das Netzwerk aufzurufen, als ob sie lokal wären.
- Sie arbeiten mit Interface Definition Language (IDL), mit der man das Kommunikationsschema definiert, das später automatisch in Code für Server und Client übersetzt wird.

Warum wurde gRPC gewählt?

- Mein Code für Aufgabe 1 war bereits in Python implementiert, und gRPC bietet eine ausgezeichnete Unterstützung für Python.
- gRPC verwendet HTTP/2 für bidirektionale Kommunikation und ist besonders effizient in der Verarbeitung von Nachrichten.
- Mit gRPC kann die Kommunikation über eine `.proto`-Datei einfach definiert und die notwendigen Stubs für Client und Server automatisch generiert werden.

## Schrittweise Umsetzung der Aufgabe

### 1. Vorbereitung der Umgebung

Um mit gRPC zu arbeiten, mussten zunächst die benötigten Python-Bibliotheken installiert werden: ***pip install grpcio grpcio-tools***

Diese Pakete ermöglichen die Nutzung und Generierung der gRPC-Code-Stubs in Python.

### 2. Definition des Kommunikationsprotokolls

Die Kommunikation zwischen den Glühwürmchen wurde in der Datei `firefly.proto` definiert. Der Inhalt dieser Datei lautet:

```
syntax = "proto3";

service FireflyService {
  rpc SendPhase (PhaseMessage) returns (Empty);
  rpc RequestPhase (Empty) returns (PhaseMessage);
}

message PhaseMessage {
  string id = 1;
  double phase = 2;
}

message Empty {}
```

Erklärung der `firefly.proto`-Datei:

- `service FireflyService`: Definiert die verfügbaren Methoden für die Kommunikation zwischen Glühwürmchen.
- `SendPhase`: Ein Nachbar sendet seine Phase an ein Glühwürmchen.
- `RequestPhase`: Ein Nachbar fragt die Phase eines Glühwürmchens ab.
- `message PhaseMessage`: Strukturiert die Nachricht, die eine Glühwürmchen-ID und eine Phase enthält.

### 3. Generierung der gRPC-Code-Stubs

Um die `.proto`-Datei in Python-Code zu übersetzen, wurde folgender Befehl ausgeführt:

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. firefly.proto
```

Warum ist die Generierung der Stubs notwendig?

Die Stubs dienen als Schnittstelle zwischen dem Python-Code und dem gRPC-Framework. Sie abstrahieren die Details der Netzwerkkommunikation und ermöglichen es, die Methoden direkt im Python-Code zu verwenden.

### 4. Anpassung der `firefly.py`

Die `firefly.py` wurde angepasst, um:

1. Einen gRPC-Server zu starten, der Nachrichten von Nachbarn empfangen kann.
2. Die Phase an Nachbarn zu senden, indem eine Client-Verbindung zu den anderen Glühwürmchen hergestellt wird.

Wichtige Ergänzungen:

- `start\_server`: Startet den Server auf einem bestimmten Port.
- `send\_phase\_to\_neighbors`: Sendet die aktuelle Phase des Glühwürmchens an alle Nachbarn.
- `update\_phase`: Aktualisiert die Phase basierend auf den empfangenen Nachbarnachrichten.

## 5. Hauptprogramm (main.py)

Das `main.py`-Skript wurde angepasst, um:

- Jedes Glühwürmchen als eigenständigen Prozess zu starten.
- Die Nachbarschaftslogik in einem Torus-Gitter zu berechnen.
- Die Initialisierung des gRPC-Servers und der Phasenlogik zu steuern.

Wichtige Änderungen:

- `get\_neighbors(id, N)`: Berechnet die IDs und Ports der Nachbarn für jedes Glühwürmchen basierend auf der Torus-Struktur.
- Parametersteuerung: Über die Kommandozeile können `K`, `dt` und die Gittergröße `N` angepasst werden.

## 6. Startskript (start\_fireflies.sh)

Das Skript startet mehrere Instanzen von `main.py` und leitet die Ausgaben in separate Log-Dateien um:

```
python main.py --id $id --N $N --K 0.5 --dt 0.1 --max_retries 5 > logs/firefly_$id.log 2>&1  
&
```

## Probleme und Lösungen

Problem: Keine empfangenen Nachrichten

- Anfangs wurden Nachrichten gesendet, aber nicht empfangen. Dies lag daran, dass mehrere Instanzen auf denselben Ports liefen.
- Lösung: Mit `pkill -f main.py` wurden alle alten Instanzen beendet, und die Prozesse wurden korrekt neu gestartet.

Erweiterung der Logs und Visualisierung

- Es wurden Logs und CSV-Exporte hinzugefügt, um die empfangenen und gesendeten Phasen aufzuzeichnen.
- Mit einem zusätzlichen Plot-Skript wurden die Phasenkurven visualisiert, um die Synchronisation der Glühwürmchen zu analysieren.



## Analyse der Ergebnisse

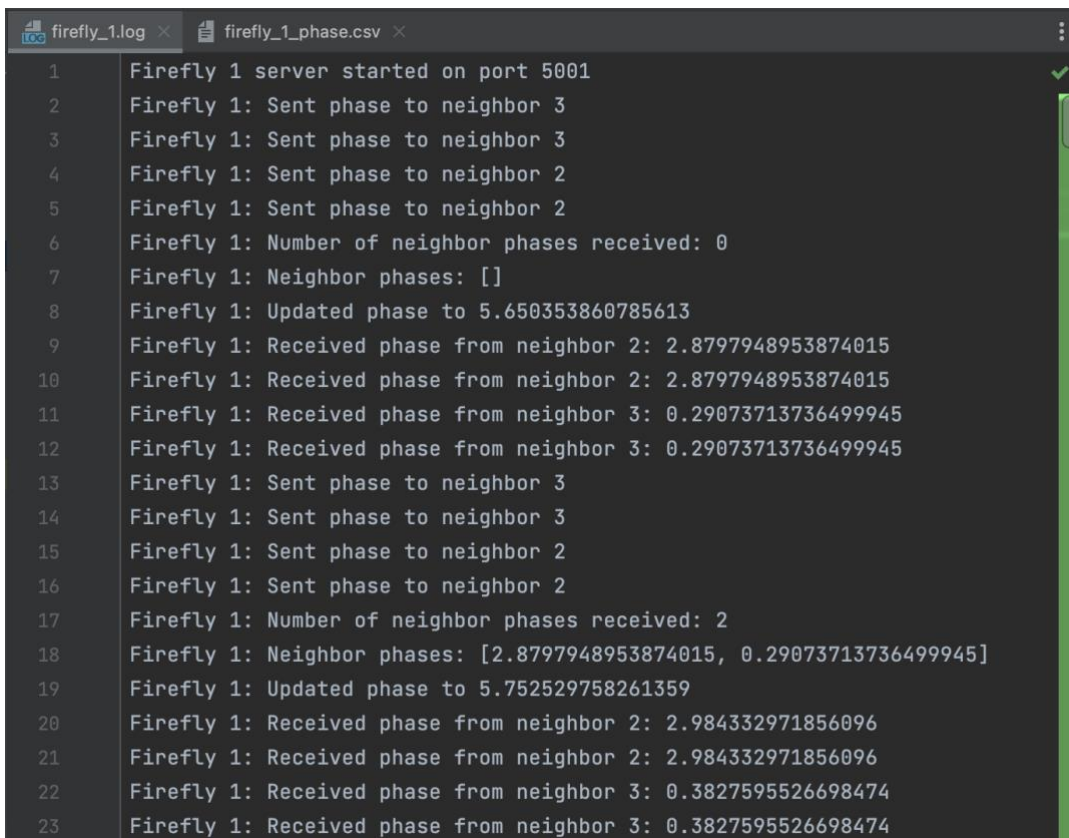
### 1. Übersicht der generierten Logs und CSV-Dateien

Die Simulation erzeugt während der Laufzeit Log-Dateien und CSV-Dateien, die jeweils die Interaktion und die Synchronisation der Glühwürmchen dokumentieren:

- **Log-Dateien (z.B., firefly\_1.log):** Enthalten detaillierte Informationen über gesendete und empfangene Nachrichten zwischen den Glühwürmchen sowie den Verlauf der Phasenaktualisierungen. Hier können wir nachvollziehen:
  - Welche Nachbarn erreicht wurden.
  - Wie viele Phasen von den Nachbarn empfangen wurden.
  - Den Fortschritt der Phasenaktualisierung gemäß dem Kuramoto-Modell.
- **CSV-Dateien (z.B., firefly\_1\_phase.csv):** Enthalten Zeitstempel und die zugehörige Phase jedes Glühwürmchens. Diese Daten dienen als Grundlage für die Visualisierung der Synchronisation über die Zeit.

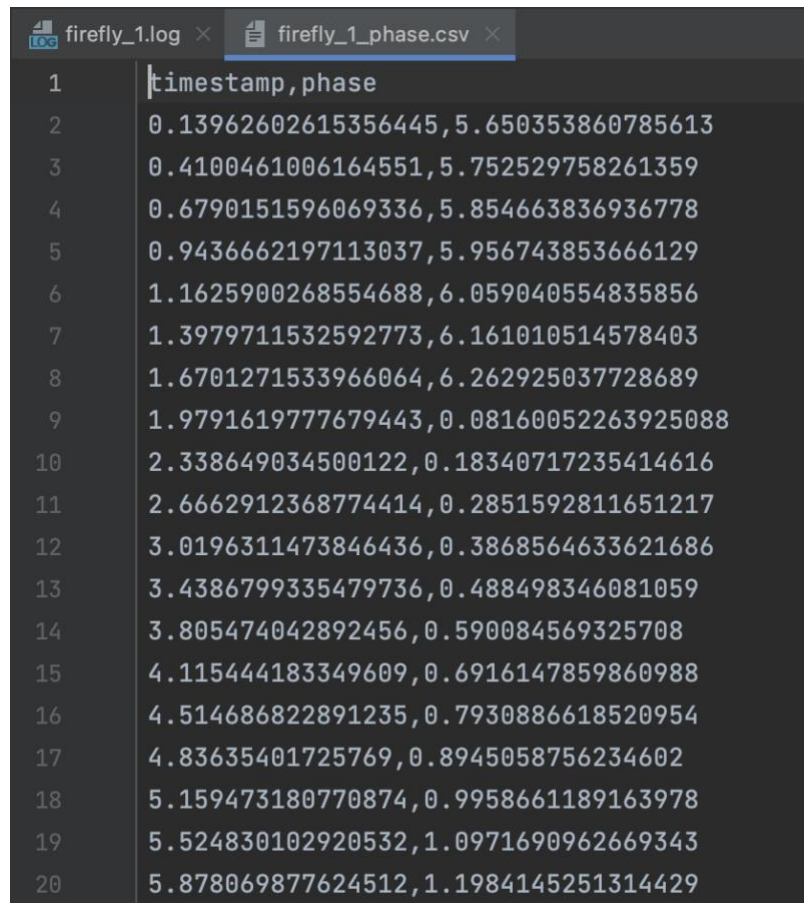
#### Beispiel aus den Logs:

- **Log-Einträge** zeigen, wie Firefly 1 Phasen von Nachbarn (z.B., Firefly 2 und 3) empfangen hat und wie diese Phasen in die eigene Aktualisierung einfließen. Es werden auch gesendete Nachrichten protokolliert.



```
firefly_1.log x firefly_1_phase.csv x
1 Firefly 1 server started on port 5001
2 Firefly 1: Sent phase to neighbor 3
3 Firefly 1: Sent phase to neighbor 3
4 Firefly 1: Sent phase to neighbor 2
5 Firefly 1: Sent phase to neighbor 2
6 Firefly 1: Number of neighbor phases received: 0
7 Firefly 1: Neighbor phases: []
8 Firefly 1: Updated phase to 5.650353860785613
9 Firefly 1: Received phase from neighbor 2: 2.8797948953874015
10 Firefly 1: Received phase from neighbor 2: 2.8797948953874015
11 Firefly 1: Received phase from neighbor 3: 0.29073713736499945
12 Firefly 1: Received phase from neighbor 3: 0.29073713736499945
13 Firefly 1: Sent phase to neighbor 3
14 Firefly 1: Sent phase to neighbor 3
15 Firefly 1: Sent phase to neighbor 2
16 Firefly 1: Sent phase to neighbor 2
17 Firefly 1: Number of neighbor phases received: 2
18 Firefly 1: Neighbor phases: [2.8797948953874015, 0.29073713736499945]
19 Firefly 1: Updated phase to 5.752529758261359
20 Firefly 1: Received phase from neighbor 2: 2.984332971856096
21 Firefly 1: Received phase from neighbor 2: 2.984332971856096
22 Firefly 1: Received phase from neighbor 3: 0.3827595526698474
23 Firefly 1: Received phase from neighbor 3: 0.3827595526698474
```

- **CSV-Einträge** dokumentieren den exakten Verlauf der Phase über die Zeit.  
Beispielsweise zeigt der CSV-Ausschnitt:
  - Zeitstempel: 0.139, Phase: 5.65
  - Zeitstempel: 0.410, Phase: 5.75 Diese Werte zeigen, wie sich die Phase eines einzelnen Glühwürmchens über die Zeit verändert.



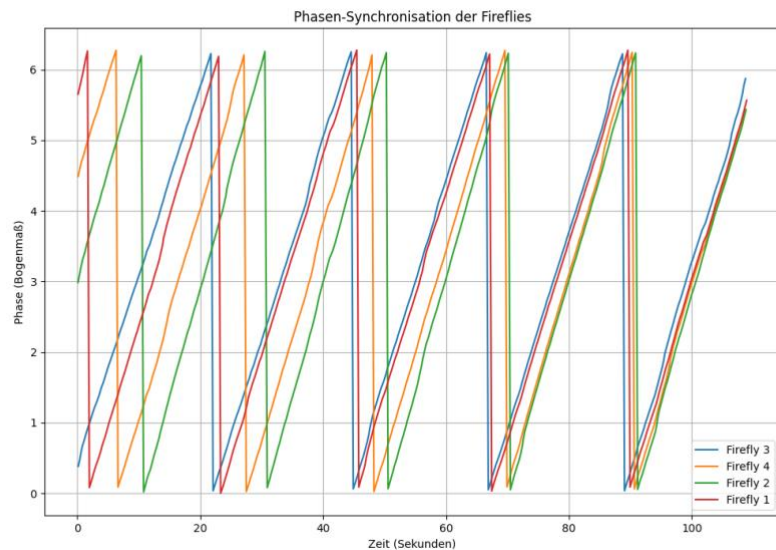
	timestamp, phase
2	0.13962602615356445, 5.650353860785613
3	0.4100461006164551, 5.752529758261359
4	0.6790151596069336, 5.854663836936778
5	0.9436662197113037, 5.956743853666129
6	1.1625900268554688, 6.059040554835856
7	1.3979711532592773, 6.161010514578403
8	1.6701271533966064, 6.262925037728689
9	1.9791619777679443, 0.08160052263925088
10	2.338649034500122, 0.18340717235414616
11	2.6662912368774414, 0.2851592811651217
12	3.0196311473846436, 0.3868564633621686
13	3.4386799335479736, 0.488498346081059
14	3.805474042892456, 0.590084569325708
15	4.115444183349609, 0.6916147859860988
16	4.514686822891235, 0.7930886618520954
17	4.83635401725769, 0.8945058756234602
18	5.159473180770874, 0.9958661189163978
19	5.524830102920532, 1.0971690962669343
20	5.878069877624512, 1.1984145251314429

## 2. Beobachtungen aus den Plots

Die Visualisierungen der Phasen (erzeugt mit `plot_phases.py`) verdeutlichen die Synchronisationsdynamik für unterschiedliche Werte der Kopplungskonstante  $K$ . Die Parameter  $N$  (Gittergröße) und  $dt$  (Zeitschritt) blieben in allen Experimenten gleich. Folgende Werte für  $K$  wurden untersucht:

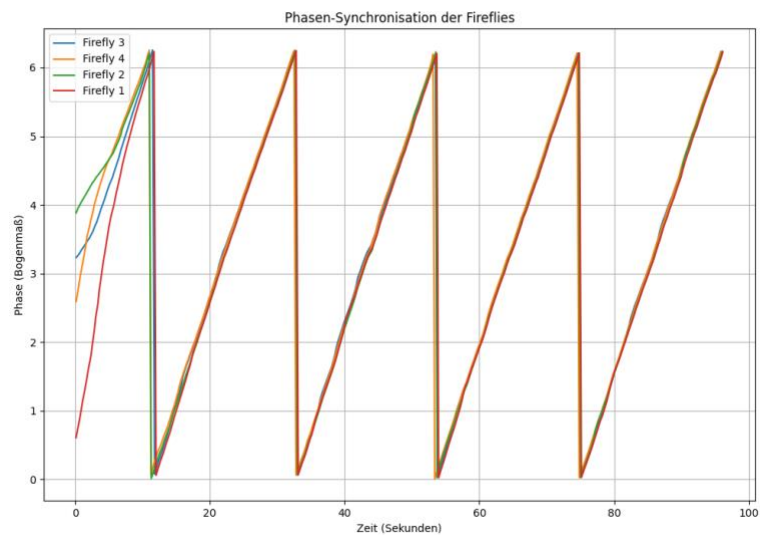
1.  **$K = 0.1$**  (erster Plot): Langsame Synchronisation
  - Die Phasen der Glühwürmchen bewegen sich nur geringfügig aufeinander zu.
  - Es dauert länger, bis eine annähernde Synchronisation erreicht wird. Einige Phasen bleiben deutlich asynchron.

- **Interpretation:** Eine geringe Kopplungskonstante führt zu schwacher Interaktion zwischen den Glühwürmchen, was die Synchronisation erschwert.



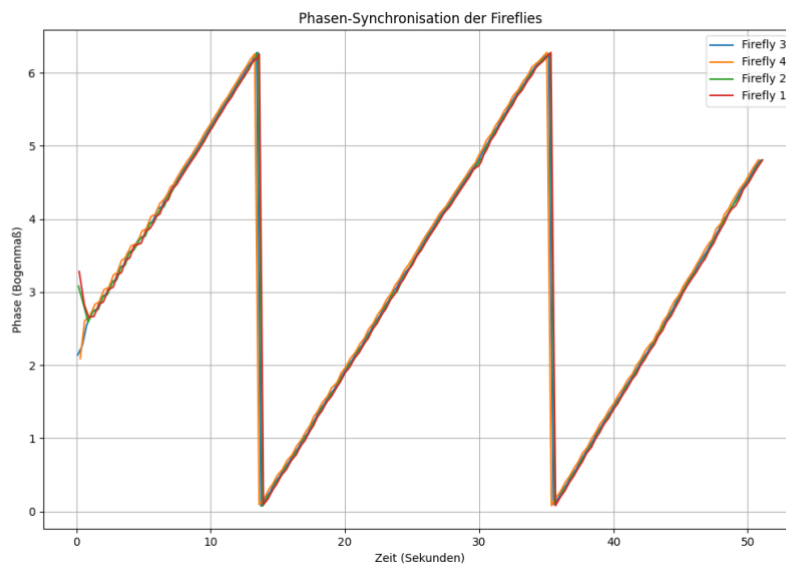
## 2. $K = 2.0$ (zweiter Plot): Mittlere Synchronisation

- Die Phasen beginnen schneller, sich anzupassen.
- Die Differenz zwischen den Phasen wird zunehmend kleiner, und ein höherer Synchronisationsgrad wird erreicht.
- **Interpretation:** Eine mittlere Kopplungskonstante verstärkt die Interaktion und erleichtert die Synchronisation.



### 3. $K = 5.0$ (dritter Plot): Schnelle Synchronisation

- Die Phasen der Glühwürmchen passen sich innerhalb kürzester Zeit aneinander an.
- Es wird fast sofort eine vollständige Synchronisation erreicht.
- **Interpretation:** Eine hohe Kopplungskonstante führt zu starker Interaktion und beschleunigt die Synchronisation



### 3. Anforderungen an die Ausführung

Um sicherzustellen, dass die Simulation reibungslos läuft, müssen vor jedem neuen Start folgende Schritte durchgeführt werden:

1. **Löschen der alten Log- und CSV-Dateien:** Die Dateien im Verzeichnis logs/ müssen entfernt werden, um Verwechslungen oder Überlagerungen mit alten Ergebnissen zu vermeiden.

```
bash
```

```
rm logs/*
```

2. **Beenden laufender Instanzen:** Alle laufenden Prozesse müssen beendet werden, da doppelte Instanzen mit denselben Ports dazu führen, dass Nachrichten nicht empfangen werden. Dies kann durch den folgenden Befehl erreicht werden:

```
bash
```

```
pkill -f main.py
```

Dieser Schritt ist entscheidend, da sonst Konflikte in der Portzuweisung auftreten, was die Kommunikation zwischen den Glühwürmchen verhindert.