**HSBI Bielefeld**

**University of Applied Sciences**

**Fachbereich Ingenieurwissenschaften und Mathematik**

**Studiengang Optimierung und Simulation**

# Lösen von nichtlinearen Gleichungssystemen mit einem Reinforcement-Learning-Agent

## Bericht

| | |
|---|---|
| Vorgelegt von: | Nicolas Schneider |
| Matrikelnummer: | 1208960 |
| Studiengang: | Optimierung und Simulation |
| Abgabedatum: | 07.04.2024 |
| Betreuer: | Prof. Dr. rer. nat. Bernhard Bachmann |

**Abstract**

Nichtlineare Gleichungssysteme (NGS) spielen eine zentrale Rolle in vielen Bereichen der Wissenschaft und Technik, da sie zur Modellierung und Lösung komplexer Probleme in Physik, Chemie, Ingenieurwesen und anderen Disziplinen eingesetzt werden. Trotz ihrer Bedeutung stellt die Lösung von NGS aufgrund ihrer inhärenten Nichtlinearität und des Fehlens geschlossener analytischer Lösungen eine große Herausforderung dar. Traditionelle numerische Verfahren wie das Newton-Raphson-Verfahren oder Optimierungsansätze stoßen oft an ihre Grenzen, insbesondere bei hochdimensionalen Problemen, chaotischem Verhalten oder starker Abhängigkeit von Anfangsbedingungen.

In jüngster Zeit hat der Bereich des Reinforcement Learnings (RL) zunehmend an Bedeutung gewonnen und vielversprechende Ergebnisse bei der Lösung komplexer Probleme geliefert. RL-Agenten lernen durch Interaktion mit einer Umgebung und Belohnungssignale, optimale Strategien zu entwickeln, ohne explizite Programmierung. Dieser Ansatz hat sich in verschiedenen Anwendungsfeldern wie Robotik, Spielen und Optimierungsproblemen als erfolgreich erwiesen.

In dieser Arbeit werden die beiden Ansätze der nichtlinearen Gleichungssysteme kombiniert, wobei ein Fokus auf die Integration von Reinforcement Learning (RL) liegt, um Lösungen zu generieren. Ein RL-Agent wird in einer maßgeschneiderten Umgebung trainiert, die die Struktur des gegebenen nichtlinearen Gleichungssystems widerspiegelt. Durch die Formulierung von Belohnungen für Aktionen, die den Agenten näher an eine Lösung führen, wird dieser befähigt, iterative Strategien zur effizienten Lösungsfindung zu erlernen.

Die vorliegende Arbeit präsentiert einen initiierenden Ansatz und analysiert seine Leistung hinsichtlich Schnelligkeit und Genauigkeit bei der Lösungsfindung von nichtlinearen Gleichungssystemen. Dabei wird eine eingehende Untersuchung durchgeführt, um die Effektivität dieses Ansatzes im Vergleich zu etablierten Methoden wie dem Newton-Raphson-Verfahren zu bewerten. Besonderes Augenmerk wird auf die Identifizierung und Analyse der limitierenden Faktoren dieses Ansatzes gelegt, um potenzielle Schwächen aufzudecken und zu verstehen, inwiefern dieser Ansatz für praktische Anwendungen geeignet ist.

# Inhaltsverzeichnis

**1 Grundlagen nichtlinearer Gleichungssysteme** **3**

**2 Grundlagen Reinforcement Learning** **3**

**3 Reinforcement Learning für die Lösung nichtlinearer Gleichungssysteme** **6**

**4 Umsetzung** **7**

**5 Ergebnisse** **9**

**6 Fazit** **9**

# 1 Grundlagen nichtlinearer Gleichungssysteme

Ein nichtlineares Gleichungssystem kann als Nullstellenproblem wie folgt formuliert werden:

**Definition 1.1 (Nichtlineares Gleichungssystem):**
Es sei $B \subset \mathbb{R}^n$ und $\mathbf{g} : B \to \mathbb{R}^n$. Gesucht sind Lösungen von

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

$\mathbf{f}(\mathbf{x}) = \mathbf{0}$ ist ein System von $n$ nichtlinearen Gleichungen für $n$ Unbekannte $x_1, \ldots, x_n$.

$$\begin{array}{rcl} f_1(x_1, \ldots, x_n) & = & 0 \\ f_2(x_1, \ldots, x_n) & = & 0 \\ \vdots & & \\ f_n(x_1, \ldots, x_n) & = & 0 \end{array}$$

Nichtlineare Gleichungssysteme treten in vielen Bereichen der Wissenschaft und Technik auf, wie beispielsweise in der Physik, Chemie, Biologie, Wirtschaftswissenschaften und Ingenieurwissenschaften. Sie dienen zur Modellierung und Analyse komplexer Systeme, deren Verhalten durch nichtlineare Beziehungen zwischen den Variablen beschrieben wird.
Die Lösung solcher Systeme ist jedoch oft eine große Herausforderung, da nichtlineare Gleichungen im Allgemeinen keine geschlossenen analytischen Lösungen besitzen. Stattdessen müssen numerische Verfahren eingesetzt werden, um approximative Lösungen zu finden. Einige gängige Methoden zur Lösung nichtlinearer Gleichungssysteme sind:

1. **Iterative Verfahren**: Hierzu zählen Methoden wie das Newton-Verfahren, das Quasi-Newton-Verfahren und das Broyden-Verfahren. Diese Verfahren starten mit einer Anfangsschätzung und verbessern diese iterativ, bis eine ausreichend genaue Lösung gefunden ist.

2. **Globale Optimierungsverfahren**: Wenn das Gleichungssystem als Optimierungsproblem formuliert werden kann, können globale Optimierungsverfahren wie die Branch-and-Bound-Methode oder evolutionäre Algorithmen zur Lösung eingesetzt werden.

Die Schwierigkeit, nichtlineare Gleichungssysteme zu lösen, besteht oft darin, geeignete Anfangsschätzungen für die Iterationsverfahren zu finden und die Konvergenz der Verfahren sicherzustellen. Viele Systeme weisen mehrere Lösungen auf, von denen einige instabil oder nicht physikalisch sinnvoll sein können. Darüber hinaus können nichtlineare Systeme eine komplexe Struktur mit mehreren lokalen Extrema aufweisen, was die globale Konvergenz erschwert.

# 2 Grundlagen Reinforcement Learning

Reinforcement Learning (RL) ist ein Teilgebiet des Maschinellen Lernens, bei dem ein Agent lernt, durch Interaktion mit einer Umgebung eine bestimmte Aufgabe oder ein Ziel zu erreichen. Im Gegensatz zu überwachtem Lernen, bei dem ein Modell anhand von Trainingsbeispielen mit bekannten Eingabe-Ausgabe-Paaren gelernt wird, erhält der Agent beim Reinforcement Learning nur eine skalare Bewertung (Reward) für seine Aktionen. Durch Ausprobieren und Lernen aus den erhaltenen Rewards versucht der Agent, eine Strategie (Policy) zu

finden, die die kumulative Belohnung über die Zeit maximiert.

**Markov-Entscheidungsprozess:**

Reinforcement Learning Probleme werden häufig als Markov-Entscheidungsprozesse (Markov Decision Processes, MDPs) formuliert. Ein MDP besteht aus:

- Einem Zustandsraum $\mathcal{S}$, der alle möglichen Zustände der Umgebung enthält.
- Einem Aktionsraum $\mathcal{A}$, der alle möglichen Aktionen des Agenten definiert.
- Einer Übergangswahrscheinlichkeitsfunktion $\mathcal{P}(s, a, s')$, die die Wahrscheinlichkeit angibt, dass der Agent beim Ausführen der Aktion $a$ im Zustand $s$ in den Zustand $s'$ übergeht.
- Einer Belohnungsfunktion $\mathcal{R}(s, a, s')$, die die Belohnung definiert, die der Agent erhält, wenn er aus dem Zustand $s$ durch Ausführen der Aktion $a$ in den Zustand $s'$ übergeht.

Das Ziel des Agenten ist es, eine Policy $\pi : \mathcal{S} \to \mathcal{A}$ zu finden, die die erwartete kumulative Belohnung über die Zeit maximiert.
Abbildung 1 stellt den Lernprozess eines Agenten in einer Umgebung vereinfacht dar.

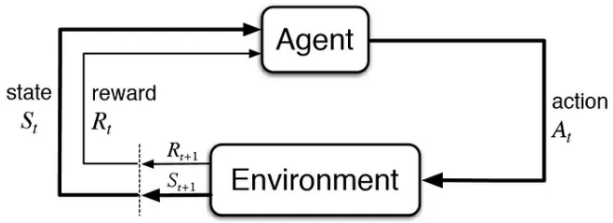


Abbildung 1: Vereinfachter Lernprozess eines Agenten in einer Umgebung[1]

**Value Functions und Bellman-Gleichungen:**

Eine zentrale Rolle beim Reinforcement Learning spielen die Value Functions, die den erwarteten kumulativen Reward für einen gegebenen Zustand oder eine Zustand-Aktions-Paar angeben. Es gibt zwei wichtige Value Functions:

- Die State-Value Function $V^\pi(s)$ gibt den erwarteten kumulativen Reward an, wenn der Agent im Zustand $s$ startet und der Policy $\pi$ folgt.
- Die Action-Value Function $Q^\pi(s, a)$ gibt den erwarteten kumulativen Reward an, wenn der Agent im Zustand $s$ startet, die Aktion $a$ ausführt und danach der Policy $\pi$ folgt.

Die Value Functions erfüllen die Bellman-Gleichungen, die eine rekursive Beziehung zwischen den Value Functions benachbarter Zustände herstellen. Für die State-Value Function lautet die Bellman-Gleichung:

$$V^\pi(s) = \mathbb{E}_\pi \left[ R(s, a, s') + \gamma V^\pi(s') \right] \tag{1}$$

Und für die Action-Value Function:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ R(s, a, s') + \gamma \sum_{s'} \mathcal{P}(s, a, s') V^\pi(s') \right] \tag{2}$$

[1] https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

Hier ist $\gamma \in [0, 1]$ der Diskontierungsfaktor, der bestimmt, wie viel Gewicht zukünftigen Rewards beigemessen wird. Ein Wert von $\gamma$ nahe 0 bedeutet, dass der Agent nur die unmittelbaren Rewards optimiert, während ein Wert nahe 1 längerfristige Belohnungen stärker gewichtet.

**Proximal Policy Algorithmus (PPO):**

Proximal Policy Optimization (PPO) ist ein moderner Algorithmus im Bereich des Reinforcement Learning, der zur Klasse der Policy Gradient Methoden gehört. PPO wurde von Schulman et al. [**schulman2017proximal**] entwickelt und hat sich aufgrund seiner guten Performanz und Stabilität in verschiedenen Anwendungsbereichen etabliert.

Das Ziel von PPO ist es, eine optimale Policy zu finden, die die erwartete kumulative Belohnung maximiert. Im Gegensatz zu klassischen Policy Gradient Methoden, die oft große Schritte im Parameterraum machen und dadurch instabil werden können, verwendet PPO eine Clipping-Technik, um die Größe der Policy-Updates zu begrenzen. Dadurch wird eine stabilere Konvergenz erreicht und die Wahrscheinlichkeit von Divergenz oder schlechten Performanz reduziert.

Der Kerngedanke von PPO besteht darin, die Policy-Updates so zu gestalten, dass sie innerhalb einer vertrauenswürdigen Region (trust region) bleiben. Dies wird erreicht, indem die Differenz zwischen der alten und der neuen Policy durch eine Clipping-Funktion begrenzt wird. Die Clipping-Funktion schneidet die Wahrscheinlichkeitsverhältnisse (probability ratios) zwischen der alten und der neuen Policy bei vordefinierten Schwellenwerten ab.

Formal lässt sich die Zielfunktion von PPO wie folgt darstellen:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[ \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{3}$$

Hierbei ist $\theta$ der Parametervektor der Policy, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ das Wahrscheinlichkeitsverhältnis zwischen der neuen und der alten Policy, $\hat{A}_t$ der geschätzte Vorteil und $\epsilon$ ein Hyperparameter, der die Größe der vertrauenswürdigen Region steuert.

Der Term $\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$ sorgt dafür, dass die Policy-Updates innerhalb der vertrauenswürdigen Region bleiben. Wenn das Wahrscheinlichkeitsverhältnis $r_t(\theta)$ außerhalb des Intervalls $[1-\epsilon, 1+\epsilon]$ liegt, wird es auf die Grenzen des Intervalls gesetzt. Dadurch werden zu große Policy-Updates vermieden und die Stabilität des Lernprozesses verbessert.

PPO verwendet häufig auch eine Value Function $V_\phi(s)$, um den Wert eines Zustands zu schätzen. Die Value Function wird verwendet, um den Vorteil $\hat{A}_t$ zu berechnen, der die Qualität einer Aktion im Vergleich zum erwarteten Wert des Zustands angibt. Der Vorteil kann beispielsweise durch die Generalized Advantage Estimation (GAE) [**schulman2015high**] geschätzt werden.

Insgesamt bietet PPO eine effektive und stabile Methode zum Lernen von Policies in Reinforcement Learning Problemen. Durch die Verwendung der Clipping-Technik und der vertrauenswürdigen Region werden große Policy-Updates vermieden und eine stabile Konvergenz erreicht. PPO hat sich in verschiedenen Benchmark-Problemen und realen Anwendungen bewährt und ist ein weit verbreiteter Algorithmus im Bereich des Reinforcement Learning.

Neben dem Proximal-Policy-Algorithmus gibt es noch eine Vielzahl weiterer Algorithmen, die sich über die Zeit entwickelt haben, wie bspw. dem *Deep Deterministic Policy Gradient* (DDPG)-, *Twin Delayed DDPG* (TD3)-, oder dem *Soft Actor Critic*-Algorithmus (SAC).

Dieses Kapitel bietet lediglich einen grundlegenden Einblick in die Arbeitsweise des Reinforcement Learning. Es ist wichtig zu betonen, dass die konkrete Umsetzung eines Reinforcement-Learning-Agenten stark vom spezifischen Anwendungsfall und den zugrunde liegenden Daten und Informationen abhängt. Da Reinforcement Learning äußerst flexibel und anpassungsfähig ist, kann die Implementierung eines RL-Agenten in verschiedenen Szenarien und Kontexten erheblich variieren.

# 3 Reinforcement Learning für die Lösung nichtlinearer Gleichungssysteme

Bevor ein RL-Agent nichtlineare Gleichungssysteme lösen kann, sollten zunächst einige wichtige Fragen beantwortet werden, die für die Qualität der Lösung und die Lösungsfindung von Relevanz sind:

1. **Umgebung:** Was ist die Umgebung, in dem der Agent agiert?
2. **Zustand:** Was sind zulässige und sinnvolle Zustände?
3. **Aktionen:** Was sind zulässige und sinnvolle Aktionen, die der Agent wählen kann?
4. **Belohnung:** Wie sieht eine zielführende Belohnungsfunktion aus?

Diese Projektarbeit fokussiert sich auf einen Teil der potenziellen Antworten auf die genannten Fragen. Es sei jedoch angemerkt, dass es alternative Formulierungen geben könnte, die ebenfalls zu einer effektiven Lösung beitragen können.

Umgebung:

Im Kontext dieses Projekts wird die Umgebung als der Raum betrachtet, in dem sich die Gleichungen befinden. Die Besonderheit dabei ist, dass, sollten die Gleichungen sich im Verlauf der Lösungsfindung nicht ändern, die Umgebung statisch ist.

Zustand:

Der Zustand wird aufgefasst, als die gewählte Aktion.

Aktion:

Abhängig von der Dimensionalität des Problems kann der Agent aus einem kontinuierlichen Raum von Punkten wählen. Der Agent wählt eine Aktion $a \in \mathcal{A}$, wobei $\mathcal{A} \subseteq \mathbb{R}^{n-1}$ und $n$ die Dimensionalität des Problems widerspiegelt.

Belohnungsfunktion:

Die Formulierung der Belohnungsfunktion konzentriert sich darauf, die Lösung(en) des Gleichungssystem(s) zu finden, sofern sie denn existieren. In dieser Arbeit wird das Residuum verwendet, welches den Fehler zur eigentlichen Lösung darstellt. Das Residuum $R$ an der Optimallösung $x^*$ ist 0, sodass das Ziel dabei ist diesen Fehler zu minimieren.

**Definition 3.1 (Residuum):**
Gegeben seien $n$ nichtlineare Gleichungen mit $\mathbb{R}^n \to \mathbb{R}^n$. Es sei vorausgesetzt, dass mindestens eine Lösung $\mathbf{x}^*$ existiert, die das Gleichungssystem löst. Dann ist das Residuum an der Stelle $\mathbf{x}$ mit $\mathbf{x} \in \mathbb{R}^n$ definiert als

$$R(\mathbf{x}) = \sum (f_i(\mathbf{x}) - f_j(\mathbf{x}))^2 \quad \forall 1 \leq i < j \leq n. \tag{4}$$

Da der Agent das Ziel verfolgt die Belohnung über die Zeit zu maximieren, kann das negative Residuum als Belohnung interpretiert werden. $Reward = -R(\mathbf{x})$.

Dieses Kapitel stellt die verfolgten Ansätze dar, die bei der Umsetzung verfolgt wurden. Im folgenden Kapitel wird genauer auf die Umsetzung dieses Ansatzes eingegangen.

# 4 Umsetzung

Bevor mit der Implementierung der Problemstellung, der Umgebung und des Agents begonnen wird, ist es von entscheidender Bedeutung, die Auswahl der richtigen Programmiersprache und Bibliothek sorgfältig zu prüfen. Diese Entscheidung bildet das Fundament für das gesamte Projekt und kann einen erheblichen Einfluss auf die Effizienz, Skalierbarkeit und den Erfolg der Umsetzung haben.

Für dieses Projekt wurde die Programmiersprache *Python* als Programmiersprache gewählt, zusammen mit der Reinforcement-Learning-Bibliothek *stable-baselines3*. Diese Entscheidung basiert darauf, dass *stable-baselines3* eine breite Palette von wichtigen RL-Algorithmen implementiert und gleichzeitig eine benutzerfreundliche Integration mit *Python* bietet. Außerdem bietet diese Bibliothek eine ausgezeichnete Kompatibilität mit der Programm-Bibliothek *gymnasium* von OpenAI, die essensziell für Erstellung von RL-Umgebungen ist. Die Kombination aus *Python*, *stable-baselines3* und *gymnasium* ermöglicht eine effiziente Entwicklung und Umsetzung von Reinforcement-Learning-Algorithmen für die gegebenen Anforderungen des Projekts.

Für die Umsetzung des Agenten bzw. der Umgebung sind folgende Methoden für die Problemstellung relevant:

- `get_distance(x)`: Berechnet das Residuum für einen Vektor $\mathbf{x} \in \mathbb{R}^{n-1}$.
- `reward(residuum, *args)`: Berechnet den Reward basierend auf dem Residuum und weiteren zusätzlichen Parametern.
- `step(action)`: Führt einen Schritt während des Lernens in der Umgebung aus und aktualisiert dabei die Belohnung auf Grundlage verschiedener, vorher festgelegter Regeln.
- `reset()`: Setzt die Umgebung auf einen zufälligen Zustand zurück.

Nichtlineares Gleichungssystem:

Das nichtlineare Gleichungssystem kann mit einfachen Python-Funktionen umgesetzt werden. Nachfolgend ist ein Bespiel für eine Variante der Rosenbrock-Funktion angegeben, wie sie im Projekt-Code umgesetzt ist.

Listing 1: Nichtlineares Gleichungssystem in Python

```
1 def nse():
2     eq1 = lambda x: 10 * (x[1] - x[0] ** 2)
3     eq2 = lambda x: 1 - x[0]
4
5 return [eq1, eq2]
```

Das Gleichungssystem kann dabei mit beliebigen und beliebig vielen Gleichungen erweitert werden.

Umgebung:

Neben vielen vorgegebenen Umgebungen, die in der Programm-Bibliothek *gymnasium* implementiert sind wie z.B. *Cart Pole*, *Acrobot* oder *Pendulum* ist es auch möglich eine eigene Umgebung für einen speziellen Anwendungsfall zu erstellen. Für die Problemstellung nichtlineare Gleichungssysteme mit einem RL-Agent zu lösen, wird eine eigene Umgebung erstellt.
Nachfolgend ist eine verkürzte Initialisierung der Umgebung dargestellt.

Listing 2: Initialisierung der Umgebung

```
1  class CustomEnv(gym.Env):
2      def __init__(self, dimension):
3          super(CustomEnv, self).__init__()
4          self.dimension = dimension
5          self.low_bounds = -5.0 * np.ones(dimension)
6          self.high_bounds = 5.0 * np.ones(dimension)
7          self.action_space = gym.spaces.Box(low=self.low_bounds, high=self.
               high_bounds, dtype=np.float64)
8          self.observation_space = gym.spaces.Box(low=-np.infty, high=np.
               infty, shape=(dimension,))
9          self.state = np.array([random.uniform(self.x_min, self.x_max),
               random.uniform(self.y_min, self.y_max)] + [0.0] * dimension)
```

Bei der Initialisierung der Umgebung muss der Bereich der möglichen Aktionen (action_space) angegeben werden. Da mögliche Lösungen für ein nichtlineares Gleichungssystem sich im ganzen Raum $\mathbb{R}^n$ befinden können, bietet sich dieser auch für die möglichen Aktionen an. Jedoch ist es nicht möglich den ganzen Raum (von $-\infty$ bis $\infty$) anzugeben, sodass Grenzen gesetzt werden müssen. Diese sind im Code mit den Variablen `self.low_bounds` und `self.high_bounds` beschrieben. Hierbei zeichnet sich auch einer der Nachteile ab. Da bei den allermeisten nichtlinearen Gleichungssystemen von Beginn an nicht abzuschätzen ist, in welchem Teil des Lösungsraumes sich die Lösung(en) befindet/befinden, stellt die Wahl des Aktionsraumes auch eine Herausforderung dar. Ähnlich wie beim Newton-Raphson-Verfahren, bei dem ein initialer Startpunkt notwendig ist, der die Laufzeit des Verfahrens maßgeblich beeinflussen kann, so kann auch die Größe des Aktionsraumes die Laufzeit der Lösungsfindung maßgeblich beeinflussen.

Berechnugn des Residuums (Distanz):

Da die Güte der aktuell gefundenen Lösung mithilfe des Residuums ermittelt wird, wird an dieser Stelle kurz dargestellt wie das Residuum im Projekt-Code berechnet wird.

Für manche Art von Problemen hat es sich als vorteilhaft erwiesen, das Residuum auf eine festgelegte Skala zu skalieren (bspw. logarithmische Skalierung, Min-Max-Skalierung etc.). Im folgenden ist ein Ausschnitt aus der Methode zur Berechnung des Residuums zu betrachten.

Listing 3: Berechnung des Residuums

```
1  def get_distance(point):
2      scaling = "logarithmic"
3      equations = self.nse()
4
5      # Calculate the function values for all equations
6      values = np.array([eq(point) for eq in equations])
7
```

```
8       # calculate normal residuum
9       normal_res = sum((values[i] - values[j]) ** 2 for i in range(len(values
            )) for j in range(i + 1, len(values)))
10      self.all_residuals.append(normal_res)
11
12      elif scaling == "logarithmic":
13          scaled_values = np.log1p(np.abs(values)) / np.log(10)
14          res = sum((scaled_values[i] - scaled_values[j]) ** 2 for i in range
                (len(scaled_values)) for j in
15          range(i + 1, len(scaled_values)))
16          return res, normal_res
```

Ausführung eines Lernschritts:

Die Ausführung eines Lernschritts ist entscheidend für die Lösungsfindung. Daher werden im nachfolgenden die wichtigsten Schritte beschrieben.
Zuerst seien die wichtigsten Variablen in folgender Tabelle definiert.

Tabelle 1: Variablendefinitionen

| Variable | Beschreibung |
|---|---|
| *distances* | Eine Liste, die Entfernungen enthält. |
| *best_actions* | Eine Liste, die die besten Aktionen enthält. |
| *best_distances* | Eine Liste, die die besten erreichten Entfernungen enthält. |
| *improvement_rate_history* | Eine Liste, die Verbesserungsraten enthält. |
| *good_points_threshold* | Der Schwellenwert für einen "guten Punkt". |
| *no_improvement_limit* | Die Grenze für aufeinanderfolgende Iterationen ohne Verbesserung. |
| *dynamic_reward* | Die dynamisch akkumulierte Belohnung. |
| *dynamic_penalty* | Die dynamisch akkumulierte Strafe. |
| *total_reward* | Die insgesamt akkumulierte Belohnung. |
| *improvement_rate* | Die berechnete Verbesserungsrate. |
| *done* | Ein boolescher Wert, der den Abschluss anzeigt. |
| *consecutive_no_improvement* | Die Anzahl aufeinanderfolgender Iterationen ohne Verbesserung. |

Beschreibung:

1. **Initialisierung:**
    - Die Variable `residuum` wird durch `get_distance(action)` initialisiert, wobei `action` die aktuelle Aktion darstellt.
    - Die Variable `reward` wird als das Negative des Residuums initialisiert, um eine Strafe für ungünstige Zustände zu erhalten.
    - Die dynamisch akkumulierte Belohnung (`dynamic_reward`) und Strafe (`dynamic_penalty`) sowie die Gesamtbelohnung (`total_reward`) werden auf Null gesetzt.
    - Die `no_improvement_limit` wird auf 50 festgelegt, um die Anzahl aufeinanderfolgender Iterationen ohne Verbesserung zu begrenzen.
    - Die Variable `done` wird auf `False` gesetzt, um den Algorithmus als nicht abgeschlossen zu markieren.

**Algorithm 1:** Ausschnitt zur Ausführung eines Schrittes

**Input:** action
**Output:** reward, done

**Initialization:**
residuum = get_distance(action), reward = −residuum, dynamic_reward = dynamic_penalty = 0, total_reward = 0, no_improvement_limit = 50, done = False;
# Belohnung, wenn die derzeitige Aktion besser als die vorherige ist
**if** *len*(*distances*) > 1 ***and*** *residuum* < *distances[n-1]* **then**
    residuum_difference_reward = distances[n-1] − residuum
    dynamic_reward = dynamic_reward + residuum_difference_reward · (distances[n-1] − residuum)
**end**
# Bestrafung, wenn die Aktion schlechter als die vorherige ist
**if** *len*(*best_actions*) > 1 ***and*** *residuum* > *best_distances[-2]* **then**
    residuum_difference_penalty = residuum − distances[n-1]
    dynamic_penalty = dynamic_penalty · (residuum − distances[n-1])
**end**
reward = reward + dynamic_reward
penalty = penalty + dynamic_penalty
# Berechnet die Verbesserungsrate, passt den Schwellenwert für gute Punkte an und setzt das Limit für aufeinanderfolgende Nichtverbesserungen.
**if** *len*(*best_distances*) > 1 **then**
    $\text{improvement\_rate} = \frac{\text{distances[n-1]} - \text{distances[n]}}{\text{distances[n-1]}}$
    append improvement_rate to improvement_rate_history
    good_points_thrs = $\max\{0.9, \text{good\_points\_thrs} \cdot (1 - \text{improvement\_rate})\}$
    no_improvement_limit = $\max\{50, 100 \cdot \text{improvement\_rate}\}$
**end**
**else**
    no_improvement_rate = 50
**end**
# Wenn residuum besser als Schwellenwert
**if** *residuum* ≤ *good_points_threshold* **then**
    done = True
    consecutive_no_improvement = 0
**end**
**else**
    consecutive_no_improvement = consecutive_no_improvement +1
    **if** *consecutive_no_improvement* ≥ *no_improvement_limit* **then**
        done = True
        reward = 0
    **end**
**end**
# Belohnung, wenn die Verbesserungsrate besser ist als 0.01
**if** *improvement_rate* > 0.01 **then**
    reward = reward +1
**end**
**return** state, reward, done

2. **Überprüfung der Bedingungen:**
   - Wenn die Länge der Liste `distances` größer als eins ist und das aktuelle Residuum kleiner als das vorherige Residuum ist, wird eine Belohnung für die Verbesserung berechnet und zu `dynamic_reward` addiert.
   - Wenn die Länge der Liste `best_actions` größer als eins ist und das aktuelle Residuum größer als das vorherige beste Residuum ist, wird eine Strafe für die Verschlechterung berechnet und zu `dynamic_penalty` addiert.
3. **Aktualisierung der Belohnung und Strafe:**
   - Die dynamisch akkumulierte Belohnung (`dynamic_reward`) und Strafe (`dynamic_penalty`) werden zu `reward` und `penalty` addiert.
4. **Aktualisierung der Verbesserungsrate und des Schwellenwerts für gute Punkte:**
   - Die Verbesserungsrate wird berechnet und zur Verbesserungsraten-Historie (`improvement_rate_history`) hinzugefügt.
   - Der Schwellenwert für gute Punkte (`good_points_thrs`) wird entsprechend der aktuellen Verbesserungsrate angepasst.
   - Die Grenze für aufeinanderfolgende Nichtverbesserungen (`no_improvement_limit`) wird basierend auf der Verbesserungsrate aktualisiert.
5. **Überprüfung der Terminierungsbedingung:**
   - Wenn das aktuelle Residuum kleiner oder gleich dem Schwellenwert für gute Punkte ist, wird der Algorithmus als abgeschlossen markiert und die Anzahl aufeinanderfolgender Nichtverbesserungen wird auf null zurückgesetzt.
   - Andernfalls wird die Anzahl aufeinanderfolgender Nichtverbesserungen inkrementiert, und wenn diese die Grenze erreicht oder überschreitet, wird der Algorithmus ebenfalls als abgeschlossen markiert und die Belohnung auf null gesetzt.
6. **Belohnungsformung:**
   - Eine zusätzliche Belohnung wird vergeben, wenn die Verbesserungsrate größer als 0,01 ist.
7. **Rückgabe:** Die Zustandsinformation, die Belohnung und der Abschlussstatus werden zurückgegeben.

# 5 Ergebnisse

# 6 Fazit