

Hodgkin und Huxley Neuronensimulation



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Anonymous
29. Juli 2020

1 Aufgabe 1

1.1 Teilaufgabe a)

Zu zeigen ist, dass die Gleichungen (8)-(10) die Gültigkeit der Gleichungen (11)-(13) bestätigen.

Wir zeigen anhand von Gleichung (8), $\frac{dn}{dt} = \alpha_n(U) \cdot (1 - n) - \beta_n(U) \cdot n$ dass Gleichung (11) $\frac{dn}{dt} = \frac{n_\infty - n}{\tau_n}$ ihre Gültigkeit hat.

Wir wissen, dass $\tau = \frac{1}{\alpha + \beta}$ für alle Gatingvariablen n , m und h ist, d.h. wir können die Gleichung, hier als (2) definiert, auch schreiben als: $\frac{dn}{dt} = (n_\infty - n) \cdot (\alpha_n + \beta_n) = n_\infty \cdot (\alpha_n + \beta_n) - n \cdot (\alpha_n + \beta_n)$.

Aus der Gleichsetzung von (8) und (11) folgt nun

$$\alpha_n(U) \cdot (1 - n) - \beta_n(U) \cdot n = n_\infty \cdot (\alpha_n(U) + \beta_n(U)) - n \cdot (\alpha_n(U) + \beta_n(U)) \Rightarrow n_\infty = \frac{\alpha_n(U)}{\alpha_n(U) + \beta_n(U)} \quad (1)$$

Damit (8) aus (11) folgt, muss also (1) gelten. Da n_∞ ein Gleichgewichtswert ist, muss gelten:

$$\frac{dn}{dt} = 0 \Rightarrow 0 = \alpha_n(U) \cdot (1 - n_\infty) - \beta_n(U) \cdot n_\infty \Rightarrow n_\infty = \frac{\alpha_n(U)}{\alpha_n(U) + \beta_n(U)}$$

Also folgt (8) aus (11) und somit folgt (11) aus (1). Für die Differentialgleichungen der Gatingvariablen m und h , sind die Rechenschritte äquivalent. Deshalb reicht es, die Gültigkeit für n zu zeigen.

1.2 Teilaufgabe b)

Die sinnvollen Anfangsbedingungen für m sowie h sind relativ trivial zu bestimmen. Da m die Aktivierungs- und h die Inaktivierungsvariable ist, können wir $m = 0.01$ und $h = 0.99$ wählen, da $m, h, n \in (0, 1)$ sind. Für n haben wir uns entschieden, die Variable als "halb offen" zu wählen, da anhand der nicht gegebenen Informationen nicht bestimmen können, wie es sich bei inaktiven Neuronen verhält. Deshalb bestimmten wir $n = 0.5$.

2 Aufgabe 2

2.1 Teilaufgabe a)

Wie in der Aufgabe verlangt, wurde hier die explizite Euler Methode als numerische Methode verwendet, um ein Ruhepotential eines Neurons darzustellen.

Wir entschieden uns dazu, den konstanten Strom I_0 als $-5nA$ zu wählen, da es der größte Minimalstrom ist. Das ist in allen Aufgaben der Fall. Die Idee der Implementierung war zunächst, den Code aufzuräumen indem wir DGLs sowie ihre Solver in ein extra Package verlegt haben, damit der Code übersichtlicher wird. In diesem Package ist auch die explizite Euler Methode, diese hat als Parameter die Anzahl der Stützstellen N für das Intervall $[0, 50]$, Anfangsbedingungen für U , n , m und h und den konstanten Strom. Der Code macht dabei folgendes: Er berechnet aus N den Abstand aufeinander Stützstellen h , welchem die Funktion das Intervall in N kleinere Intervalle der Länge h einteilt.

Danach wird eine forSSchleife ausgeführt, welche über alle Elemente von *Timesteps* iteriert. In dieser Schleife werden die Spannungen

aus den Anfangsbedingungen in ein Array U gespeichert. Die Anfangsbedingungen bestehen dabei aus einem Array. Dieses enthält Werte für die Spannung, n , m und h . In der Schleife werden diese immer wieder modifiziert und in U abgespeichert. Für die Modifikation haben wir uns eine Hilfsfunktion definiert, die sich aus den Anfangsbedingungen alle Werte nimmt. Mit diesen berechnet sie die aktuellen Änderungsraten von U , n , m und h und gibt diese als return zurück. Mithilfe des return der Hilfsfunktion können wir U , n , m und h näherungsweise an der nächsten Stützstelle berechnen, indem wir das Produkt aus Intervalllänge und jeweiliger Steigung zu den aktuellen Werten addieren. Dabei nehmen wir näherungsweise lineares Verhalten in dem jeweiligen Intervall an. Mit den neuen Werten von U , n , m und h können dann die Werte für die nächste Stützstelle berechnet werden. Die Resultate dieser Methode werden in Aufgabenteil b) verglichen. Deshalb fügen wir keine zusätzliche Grafik zu dieser Aufgabe ein. Auch die Genauigkeit wird dort diskutiert.

2.2 Teilaufgabe b)

Bei dieser Aufgabe haben wir die Hilfsfunktion aus Aufgabenteil a) verwendet. Für die Runge-Kutta Methode berechneten wir uns zunächst, dieselben Sachen, wie für die explizite Euler Methode und übergeben die gleichen Parameter. D.h. die Stützstellen, das Zeitarray etc. sind gleich.

Für diese Aufgabe berechnen wir 4 sogenannte Zwischenschritte k_j , mit denen wir die DGLs dann numerisch lösen. Die Zwischenschritte werden, wie in der Vorlesung besprochen, berechnet und damit die DGLs numerisch gelöst. Für die `odeint` Methode vom `scipy` Paket, übergeben wir ebenfalls die gleichen Parameter und berechnen auch hier wieder den Abstand der Stützstellen für das Zeitintervall. Wir übergeben der Funktion `odeint` die Hilfsfunktion mit den Parametern der Funktion selber und erhalten ein Array zurück was von uns `solved` genannt wurde. Dabei müssen wir `solved` noch einmal Transponieren da es sonst die Werte falsch ausgibt. Hier haben wir als return noch das Array der Zeit. Dies brauchen wir aber für eine andere Aufgabe, weshalb es hier nicht wichtig.

Diese Rechnungen befinden sich in der `neuronen_rechnung` Datei. Die Plots der Funktionen und auch alle weiteren Teile der Aufgabe 2 der zwei befinden sich jedoch in `Aufgabe2`. In dieser Funktion überprüfen wir die Genauigkeit für verschiedene Anzahlen an Stützstellen. Ansonsten werden nur die Spannungen geplottet der jeweiligen Methode. Wie man im Anhang sieht ist die Euler-Methode für kleine N ungenauer als Runge-Kutta, welche schon für kleine N sehr genau ist. Erst für größere N werden die Euler-Methode und die `odeint` Methode immer mehr identisch, was früher für die Runge-Kutta Methode gegeben ist, siehe Figur 2. Aus diesem Grund haben wir uns entschieden, die Runge-Kutta Methode für die folgenden Aufgaben zu verwenden.

2.3 Teilaufgabe c)

In dieser Aufgabe sollten wir verschiedene Ströme, zwischen $I \in [-5, 15]nA$ wählen und visualisieren. Wir entschieden uns dazu jeweils in vier Schritten, diese zu visualisieren, also $I = \{-5nA, -1nA, 3nA, \dots, 15nA\}$. In dieser Aufgabe benutzen wir die Runge-Kutta Methode, wie oben angedeutet. Die Implementierung ist im Wesentlichen dieselbe. Hier wurde nur der konstante Strom I_0 verändert.

Wie wir im Anhang sehen, haben wir für verschiedene Stromstärken verschieden viele Aktionspotentiale. Zuerst haben wir das Ruhepotential bei $I = -5nA$. Dieses entspricht unseren Graphen von davor. Wir sehen nun jedoch, dass wenn wir eine größere Stromstärke als den Strom im Ruhezustand anlegen, erreichen wir einen Schwellwert, welcher dafür sorgt, dass ein Aktionspotential entsteht, hier $I = -1nA$. Ebenfalls zu erkennen ist: Je größer die Stromstärke ist, desto mehr Aktionspotentiale können nacheinander ausgeführt werden.

2.4 Teilaufgabe d)

Für diese Aufgabe war gefordert, einen Impuls mit einer Stromstärke von $I = 50nA$ bei $t = 10ms$ bis $t = 11ms$ zu setzen. Für diese Aufgabe mussten wir unsere Runge-Kutta Methode und unsere Hilfsfunktion leicht modifizieren. Wir mussten unserer neuen Hilfsfunktion `hilfsfunktion_ein_impuls` lediglich den Impuls übergeben und diesen dann zum Zeitpunkt t zu unserem Ruhestrom dazu addieren, um unsere DGL für die Spannung richtig zu lösen. Dazu haben wir eine Funktion `impuls_erstellt`. Dieser Funktion übergeben wir nur den gewünschten Impuls, hier $50nA$ und einen Zeitpunkt t . Die Funktion überprüft dann ob sich dieser Zeitpunkt zwischen $10ms$ und $11ms$ befindet, oder nicht. Falls er das tut, geben wir den Impuls als return zurück, ansonsten geben wir eine 0 zurück. Mit diesen neuen und modifizierten Funktionen führen wir nun die neue Runge-Kutta Methode `runge_kutta_ein_impuls` aus. Diese gibt uns nun außer der Spannung auch die Gatingvariablen n , m und h wieder.

Danach übergeben wir die modifizierte Runge-Kutta Methode der Funktion `impuls_visualisierung` in der Datei `Aufgabe2`. Als Parameter haben wir auch hier wieder die Anzahl der Stützstellen und die Anfangsbedingungen gegeben. In der Funktion selber definieren wir uns wieder die Stützstelle und das Zeitarray. Nach diesen beiden erstellen wir nun eine Kopie des Zeitarrays, welches wir `current` genannt haben. In diesem Array setzen wir über eine Boolean-Bedingung alle Werte, die größer, als $t = 11ms$ oder kleiner als $t = 10ms$ sind, als $0nA$ und alle Werte dazwischen als $50nA$, damit dieses Array unserem Impuls über die Zeit entspricht.

Als nächstes führen wir die Runge-Kutta Methode aus und speichern dessen *returns* als jeweiliges Array, damit wir diese visualisieren können. Ansonsten werden alle Arrays dann über die Zeit dargestellt. Wir haben uns dazu entschieden mit der Funktion "GridSpec" aus dem MatplotlibPaket, die jeweiligen Plots in gewisse Zellen einzuteilen, damit es geordneter und größer aussieht.

Wir sehen für die Gatingvariablen, dass die Aktivierungsvariable zu $m \rightarrow 1$ wird für Werte $t \in [10, 15)$, was auch zu seinem Namen passt, denn in diesem Intervall ist das Neuron aktiviert und gibt ein Aktionspotential aus. Nach dieser Zeit geht $m \rightarrow 0$, da sich das Neuron wieder schließt oder geschlossen hat. Andersherum ist es für die Inaktivierungsvariable h , welche sich genau verkehrt herum zu m verhält, was ebenfalls durch ihren Namen zu erwarten ist, jedoch braucht es etwas länger, um wieder die "volle Inaktivierung" zu erreichen, was bei m gar nicht der Fall ist. Für n sehen wir, dass es sich zuerst einzupendeln scheint auf einen Wert $n \approx 0.3$. Wenn das Aktionspotential eintritt erhöht sich der Wert von n über einen längeren Zeitraum als das Aktionspotential selbst, sinkt dann aber wieder auf den eingependelten Wert.

3 Aufgabe 3

3.1 Teilaufgabe a)

Für diese Aufgabe konnten wir auf dieselben schon implementierten Funktionen zurückgreifen und mussten diese wieder nur modifizieren. Zunächst wurde die Impuls Funktion dahingehend ergänzt, dass es nun zwei Zeitpunkte für einen Impuls gibt und gelten muss $t_1 < t_2$. Ansonsten wurden die neue Hilfsfunktion und Runge-Kutta Methode dahingehend bearbeitet, dass sie nun mit der neuen Impuls Funktion rechnen.

3.2 Teilaufgabe b)

Auch hier ist die Implementierung äquivalent bis auf den Fakt, dass es nun mehr als einen Impuls gibt. Die Weise, wie es implementiert ist und arbeitet, ist gleich. Die Parameter sind hier die Anfangsbedingungen, die Anzahl der Stützstellen, die I_{imp} Stromstärke und die zwei Zeitpunkte.

Hier sind die gleichen Interpretationen wie vorher möglich. Für den ersten Impuls haben wir die jeweiligen Graphen schon diskutiert, deshalb betrachten wir hier den zweiten Impuls. Wir sehen, dass die Aktivierungsvariable m beim zweiten Impuls einen leichten Anstieg hat. Dieser kommt zustande, da der Impuls in der Refraktärzeit nicht die nötige Stromstärke, hatte um ein weiteres Aktionspotential anzuregen. Die Refraktärzeit ist die Zeit nach einem Aktionspotential, in der die Spannung niedriger als das Ruhepotential ist, bis sich die Spannung wieder auf das Ruhepotential einstellt. Biologisch gesehen ist das eine Maßnahme, um dafür zu sorgen, dass nicht jeder Reiz zu einer Aktion im Körper führt, um Energie zu sparen. Ansonsten könnte sich der Körper überlasten. Die Interpretation für h und n sind dabei wie oben angedeutet für einen Impuls relativ gleich. Auch hier hat die Inaktivierungsvariable ein lokales Minimum, neben ihrem lokalen Maximum, das aufgrund des Reizes zu dieser Zeit entsteht. Ebenso ist es bei n , welches auch wieder einen leichten Anstieg bei dem Impuls hat, jedoch dadurch, dass kein Aktionspotential erreicht wird, kleiner ist, als es davor bei dem Impuls war.

3.3 Teilaufgabe c)

Diese Aufgabe bezieht sich auch wieder auf Funktionen, die vorher erklärt wurden. Die Parameter der Funktion sind die Anzahl der Stützstellen, die Anfangsbedingungen, der Strom I_0 , der Zeitpunkt des ersten Impulses und die Stromstärke des Impulses. Auch hier wurden wieder Stützstelle und Zeitarray wie bekannt definiert. Neu ist die variable t_2 . Diese ist die neue Intervalllänge, da alle Werte bis t_1 abgeschnitten werden. Die Schleife soll über alle Punkte t dieser Intervalllänge laufen, dabei haben wir eine Multiplikation von 10 gewählt, damit wir die erste Nachkommastelle für unsere Zeiten erhalten. Diese Zeit dividieren wir dann wieder sobald wir sie unserer Runge-Kutta Methode von davor übergeben. Da unsere `range(t2 * 10)` Bedingung bei 0 anfängt, addieren wir jedes mal die Zeit t_1 dazu, damit wir bei unserem aktuellen Punkt nach dem ersten Impuls sind. Von unserer Runge-Kutta Methode lassen wir uns nur die Spannungen wiedergeben, da wir diese für unsere Bedingung brauchen. Die Idee hinter der Bedingung ist dabei, dass wir die ganze Liste der Spannungen durchgehen. Falls diese eine Spannung $U > 0$ besitzt und diese nach $t = 14ms$ liegt dann soll uns das Programm den Zeitpunkt als *return* geben. Warum wir uns für $t = 14ms$ entschieden haben, liegt daran, dass man ab diesem Zeitpunkt fest sagen kann, dass wir uns nicht mehr im davorigen Aktionspotential befinden, was bedeutet das ein neues Aktionspotential auf jeden Fall vorliegen muss.

Stromstärke[nA]	10	20	30	40	50	60	70	80	90	100
Zeit[ms]	-	24.3	20.5	18.9	18	17.3	16.9	16.5	16.2	16.0

4 Aufgabe 4

Die Anzahl der benötigten Neuronen ist 5, 4 Neuronen werden zur Verarbeitung des Schachbrettes gebraucht. Diese vier Neuronen geben je nach Farbe ein Aktionspotential oder ein Ruhepotential aus. Wir haben die Schwarzen Felder als Aktionspotentialfelder gewählt. Das fünfte Neuron wird dazu verwendet, die vorherigen Impulse zu verarbeiten und je nach Impulsstärke die entscheidende Ausgabe zu geben.

Die Wichtungsfaktoren müssen dabei für die Positionen der schwarzen Felder 1 sein und für die Weißen Felder 0 sein. Also muss das Array $[1, 0, 0, 1]$ sein, wenn wir das Schachbrett als Vektor beschreiben wollen. Deshalb haben wir in der Aufgabe c) diese auch so gesetzt. Dabei sahen wir, dass das Programm entscheiden konnte, was ein Schachbrett ist und was nicht. Bei der Methode, bei der wir die Wichtungsfaktoren $[1, 1, 1, 1]$ wählen sollten, erkannte das Programm alles als kein Schachbrettmuster, was erwartet war.

Die Implementierung des Lernprozesses hatte sich als schwierig erwiesen, denn das Vergrößern und Verkleinern der Wichtungen musste richtig angepasst werden. Die Idee ist dabei, eine Liste von Schachbrettern dem Programm zu geben, mit dem es die Faktoren selbst anpasst. Zunächst werden die Schachbretter in Spannungen umgerechnet. Mit diesen neuen Arrays wird dann ein Strom ausgerechnet über $\sum_{k=1}^4 \omega_k * U_k = I$. Dieser Strom wird dann in das fünfte Neuron übergeben und ausgewertet. Die Gewichtungen werden mit Hilfe der Sigmoidfunktion angepasst. Diese verläuft zwischen $f(x) \in (0, 1)$. An jeder Stelle, an der sich eine 1 für schwarz befindet wird nun über die Sigmoidfunktion errechnet, um wie viel die Gewichtung erhöht oder gesenkt wird. Dabei übergeben wir der Sigmoidfunktion die gewünschten Ausgaben und ziehen von diesen die realen Ausgaben ab. Die Rechnung sieht wie folgt aus $Gewichtungsfaktor_i = Gewichtungsfaktor_i * (1 + \frac{1}{1+e^{-x}} - 0,55)$, mit x der eben beschriebenen Subtraktion. Wir multiplizieren es mit den Gewichtungsfaktoren damit wir nie kleiner als 0 werden. Wir ziehen 0,55 von der Summe noch ab, damit wir garantieren, dass wir immer zwischen Werten von $[0,45, 1,55]$ schwanken und diese mit der jeweiligen Position multiplizieren.

Das Programm ist sehr stabil. Egal wie wir versucht haben das Programm lernen zu lassen, es hatte immer am Ende die Ergebnisse die wir erwartet bzw. erhofft haben. Wir geben der Lernmethode jedes mal wieder die selben Schachbretter mit dem es lernen soll. Nach etwa 15 ± 3 Iterationen ist das Programm immer richtig. Nur am Anfang mit den zufälligen Werten macht das Programm noch Fehler. Die Graphen hierzu sind im Anhang zu finden.

5 Anhang

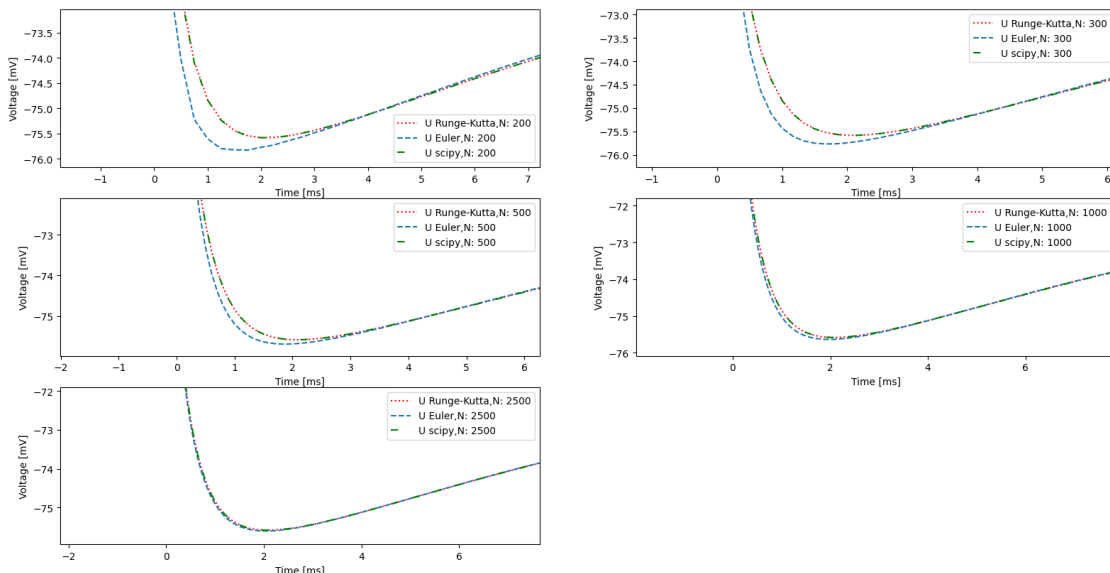


Abbildung 1: Aufgabe 2b) Zoom des Vergleichs

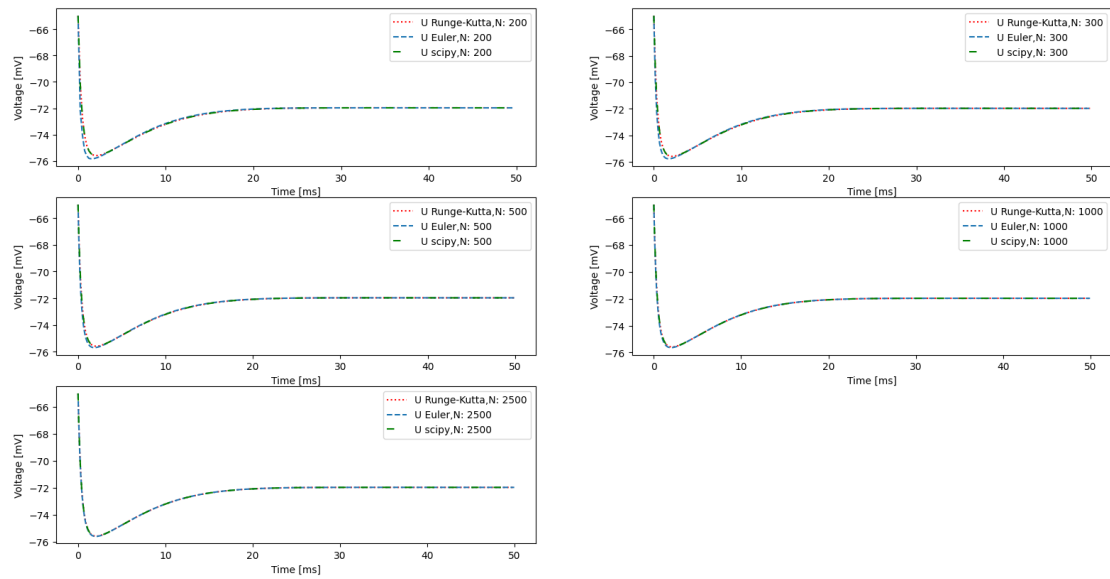


Abbildung 2: Aufgabe 2b) Vergleich der jeweiligen Methoden pro Anzahl der Stützstellen

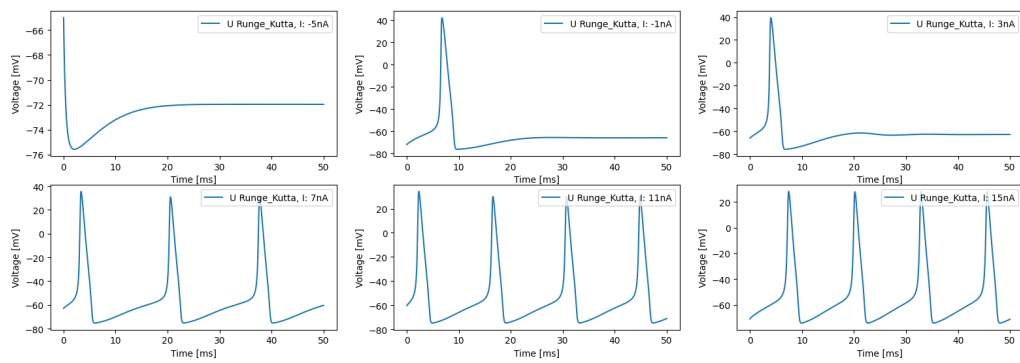


Abbildung 3: Aufgabe 2c) Verschiedene Stromstärken

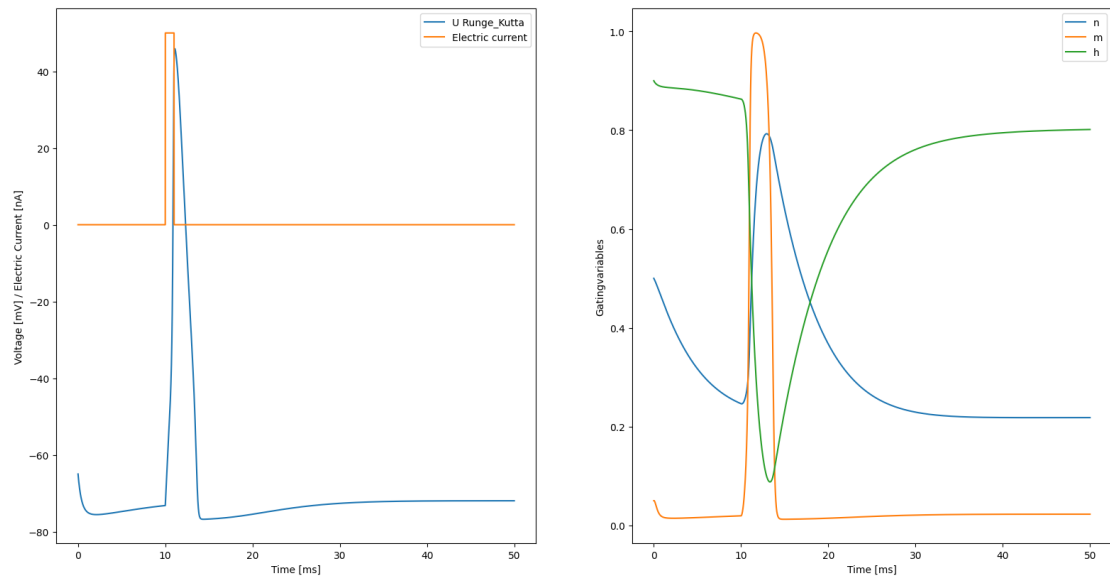


Abbildung 4: Aufgabe 2d) Links: Stromstärke und Spannung, Rechts: Gatingvariablen

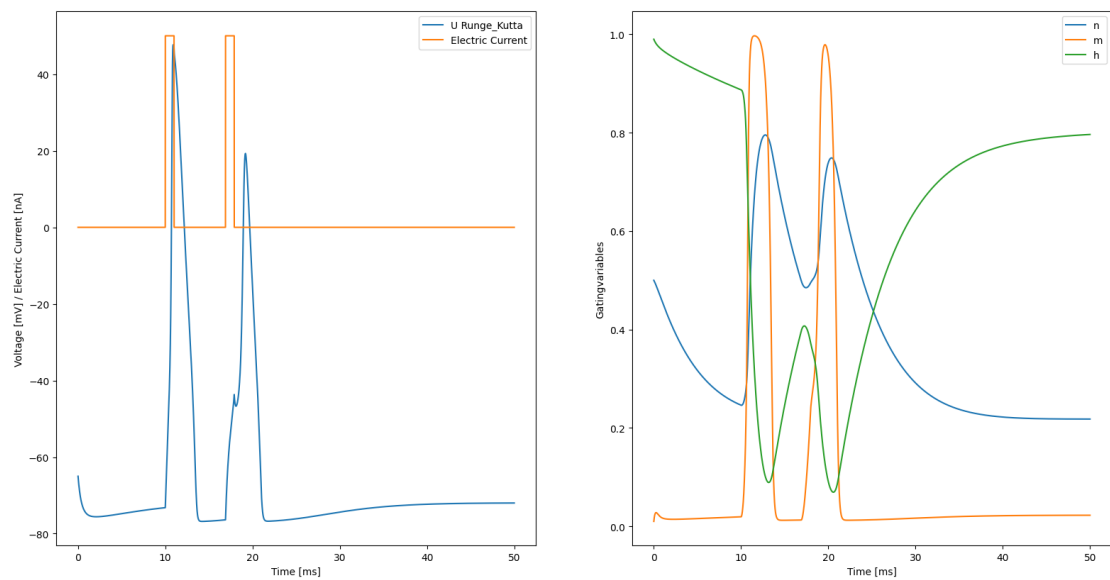


Abbildung 5: Aufgabe 3b) Links: Stromstärke und Spannung, Rechts: Gatingvariablen

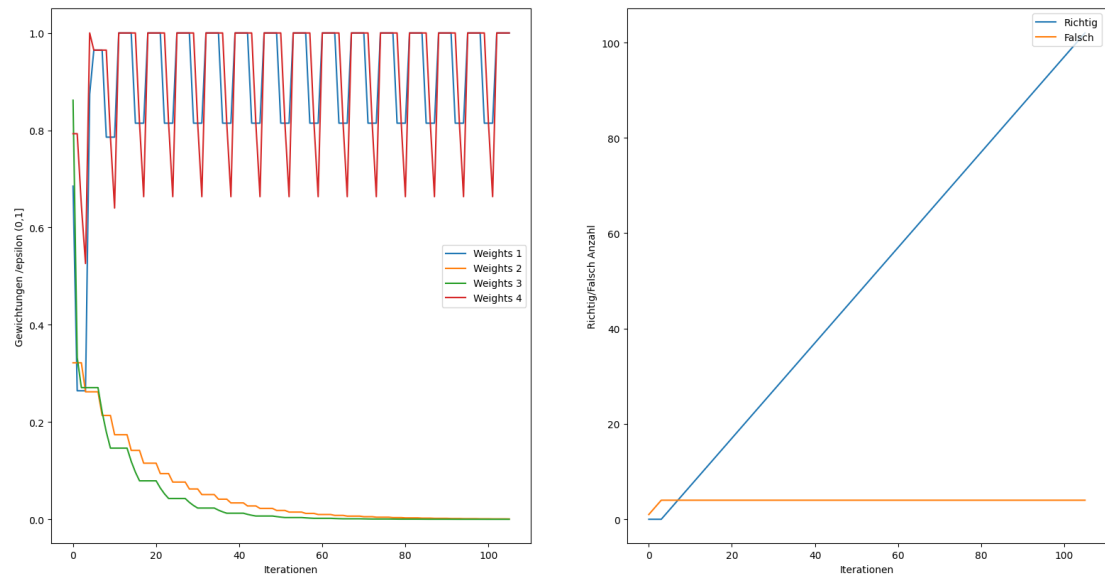


Abbildung 6: Aufgabe 4e) Links: Verlauf der Gewichtungen, Rechts: Fehlerbestimmung