Übungen zur Vorlesung

Mathematische Aspekte der Modellierung und Simulation in den Neurowissenschaften

Dr. S. Lang, D. Popović

Abgabe: 6. Juli 2010 in der Übung

Übung 22 Backward Euler-Verfahren für das Hodgkin-Huxley-Modell (8 Punkte)

In der Vorlesung und der letzten Übung haben Sie das nicht-lineare Hodgkin-Huxley-Modell kennengelernt und ein einfaches Forward Euler-Verfahren zur numerischen Lösung implementiert. In dieser Übung wollen wir nun das Backward Euler-Verfahren implementieren. Für ein allgemeines nicht-lineares ODE-System mit d Komponenten,

$$\dot{\mathbf{v}} = \mathbf{f}(\mathbf{v}),$$

$\mathbf{v} \in \mathbb{R}^d$, mit einer d-wertigen Funktion $\mathbf{f}$ hat das Backward Euler-Verfahren zur Berechnung der Lösung $\mathbf{v}^n$ im n.ten Zeitschritt die Form

$$\mathbf{v}^n = \mathbf{v}^{n-1} + k_n \cdot \mathbf{f}(\mathbf{v}^n) \tag{0.1}$$

mit der Schrittweite k_n. Im Gegensatz zum Forward Euler- ist das Backward Euler-Verfahren ein implizites Schema, d. h. $\mathbf{v}^n$ steht auch auf der rechten Seite des Systems und zur Lösung muss in jedem Zeitschritt ein nicht-lineares Gleichungssystem gelöst werden.

Dazu kann das Newton-Verfahren verwendet werden, mit dem man Nullstellen einer nicht-linearen Funktion finden kann. Diese Funktion ist hier natürlich $\mathbf{g}(\mathbf{v}^n) = \mathbf{v}^n - \mathbf{v}^{n-1} - k_n \cdot \mathbf{f}(\mathbf{v}^n)$. Für ein allgemeines nicht-lineares System $\mathbf{g}(\mathbf{x}) = 0$ hat das Newton-Verfahren die Iterationsvorschrift

$$\mathbf{x}_{m+1} = \mathbf{x}_m + \triangle \mathbf{x}_m.$$

Es handelt sich um ein iteratives Verfahren, bei dem aus der Lösung $\mathbf{x}_m$ des letzten Iterations-Schrittes mittels eines Update-Vektors $\triangle \mathbf{x}_m$ die neue Näherung $\mathbf{x}_{m+1}$ berechnet wird. Der Update-Vektor ist die Lösung des Gleichungssystems

$$G'(\mathbf{x}_m) \cdot \triangle \mathbf{x}_m = -\tilde{\mathbf{g}}(\mathbf{x}_m). \tag{0.2}$$

Hierbei ist $\tilde{\mathbf{g}}$ der Vektor der Funktionsauswertung $\tilde{\mathbf{g}}(\mathbf{x}_m) = \mathbf{g}(\mathbf{x}_m)$ und G' die Jacobi-Matrix der Abbildung $\mathbf{g}$. Achtung: Der Index m ist nicht der Zeitschritt-Index n, sondern der Index des iterativen Newton-Verfahrens. Für jeden Zeitschritt muss die Lösung $\mathbf{v}^n$ des nicht-linearen Systems (0.1) mittels mehrerer Iterationen eines Newton-Verfahrens ermittelt werden.

Algorithmisch geht man beim Newton-Verfahren wie folgt vor:

1. Setze einen geeigneten Startwert, hier z.B. $\mathbf{x}_0 = \mathbf{v}^{n-1}$.
2. Bestimme die Jacobi-Matrix G'.
3. Setze den (Start-)Vektor in die Update-Gleichung (0.2) ein. Das ergibt ein lineares Gleichungssystem für $\triangle \mathbf{x}_m$.
4. Lösen des linearen Gleichungssystems liefert das Update $\triangle \mathbf{x}_m$.

5. Berechne die neue Näherung $\mathbf{x}_{m+1}$. Falls $\|\mathbf{x}_{m+1} - \mathbf{x}_m\| > \varepsilon$ für eine gegebene Toleranz in einer geeigneten Norm, gehe zu (3).
6. Falls $\|\mathbf{x}_{m+1} - \mathbf{x}_m\| \leq \varepsilon$ setze $\mathbf{v}^n = \mathbf{x}_m$.

Im Pool finden Sie im Verzeichnis `/export/home/dune/neuroDUNE/` das Zip-Archiv `hh.zip`. Dieses enthält das Hodgkin-Huxley Modell mit Forward Euler-Verfahren sowie ein Makefile.am, das das entsprechende Ziel baut. Legen Sie die enthaltenen Dateien im `Pointneuron`-Ordner Ihrer neuroDUNE-Installation ab, und fügen Sie Ihrem bisher verwendeten `Makefile.am` ein `make`-Ziel wie im bereitgestellten `Makefile.am` hinzu (siehe dazu auch Übungsblatt 7). Übersetzen Sie das Programm.

Aufgaben

1. Führen Sie Simulationen des Programms mit der Simulationsdauer $t^N = 200ms$ und den konstanten Zeitschrittweiten $k_n = 0.1ms$ und $k_n = 0.01ms$ aus und vergleichen Sie die Lösungen. Wo treten Unterschiede auf? Haben Sie eine Erklärung für die von Ihnen beobachteten Phänomene?

2. Berechnen Sie eine Lösung mit sehr feiner Zeitschrittweite $k_n = 0.001ms$ und messen Sie das maximale Potential v_{max} im Zeitintervall $I = [90, 110]ms$. Führen Sie anschließend Simulationen mit den Zeitschrittweiten $k_n = 0.05, 0.025, 0.0125, 0.00625$ und 0.003125 aus und bestimmen Sie jeweils das maximale Potential $\tilde{v}$ in I. Tragen Sie für alle Schrittweiten $|\tilde{v} - v_{max}|$ gegenüber k_n in einem doppelt-logarithmischen Plot auf. Können Sie das lineare Konvergenz-Verhalten des Forward Euler Verfahrens erkennen?

3. Stellen Sie die Jacobi-Matrix G' der Funktion $\mathbf{g}(\mathbf{v}^n)$.

4. Implementieren Sie nun das Backward Euler-Verfahren mit dem Newton-Löser wie oben beschrieben und wiederholen Sie die Messungen aus 2. Dazu ist in der Datei `hodgkinhuxley.hh` bereits ein Funktionsgerüst `BEStep(double t, double dt)` angelegt, das Sie ausimplementieren können. Wie ist das Konvergenz-Verhalten nun? Als Toleranz können Sie $\varepsilon = 0.0001$ verwenden, alle anderen Parameter sind wie bei der Forward Euler-Simulation.

Hinweise

Die verwendete Software neuroDUNE baut auf der modularisierten Numerik-Toolbox Dune auf. In Dune gibt es bereits Datenstrukturen und Algorithmen, um das vorliegende Problem zu lösen (Sie müssen diese natürlich nicht verwenden!).

- Da es sich um ein 4-komponentiges System handelt, ist $d = 4$ und die Unbekannten können in einem 4×1-Vektor gespeichert werden. In Dune kann man durch `#include "dune/common/fvector"` die Datenstruktur `FieldVector<T,dim>` verwenden, die `dim x 1`-Vektoren mit Einträgen des Typs `T` handeln kann. Beispiel zur Verwendung:

```
Dune::FieldVector<double,4> a(0.0); a[0] = 1.0
```

  generiert einen Einheitsvektor in Richtung der ersten Komponente.

- Wie bei den `FieldVector`en gestattet `#include "dune/common/fmatrix"` das Verwenden der Datenstruktur `FieldMatrix<T,dim,dim>`, die `dim x dim`-Matrizen des Typs `T` handeln kann. Die `FieldMatrix`en haben eingebaute Methoden `invert()` und `solve(Dune::FieldVector<T,dim>& x, const Dune::FieldVector<T,dim>& b)` um eine Matrix `A` zu invertieren oder das Gleichungssystem `A x = b` zu lösen.

- unter `http://www.dune-project.org/doc-1.2/doxygen/html/classes.html` findet man generelle Hilfe zu allen Dune-Komponenten und Datenstrukturen wie z.B. `FieldVector<T,dim>`.