

Übungen zur Vorlesung
Modellierung und Simulation in den Neurowissenschaften
http://conan.iwr.uni-heidelberg.de/teaching/numsimneuro_ss2011

Dr. S. Lang, D. Popović

Abgabe: 11. Mai 2011 in der Übung

Übung 3 Matrix-Vektor-Multiplikation mit Octave (5 Punkte)

Eine Hilbert-Matrix $A(n)$ des Ranges n ist definiert über $a_{ij} := \frac{1}{i+j-1}$. Die Hilbert-Matrix des Ranges 3 ist zum Beispiel

$$A(3) = \begin{pmatrix} 1.00000 & 0.50000 & 0.33333 \\ 0.50000 & 0.33333 & 0.25000 \\ 0.33333 & 0.25000 & 0.20000 \end{pmatrix}.$$

Wir wollen in dieser Übung das Gleichungssystem $A(n) \cdot \vec{x} = \vec{b}$ lösen, wobei \vec{x} und $\vec{b} := (1, 0, \dots, 0)^t$ Vektoren des \mathbb{R}^n sind. Eine Methode, ein Gleichungssystem zu lösen, haben wir mit der *Octave*-Funktion `x1 = A\b` in der letzten Übung kennengelernt. Eine andere ist das Verwenden der Gauss-Elimination. In *Octave* geschieht dies mit der Funktion `C = rref(A,b)`, die in der $(n \times n + 1)$ -Matrix C die Matrix A nach Gauss-Elimination und in der letzten Spalte von C das Ergebnis der entsprechenden Operationen angewendet auf \vec{b} speichert. Mit `x2 = C(1:n,n+1)` wird also die Lösung des Gleichungssystems ausgegeben. Durch `[x1, x2]` werden die Lösungsvektoren nebeneinander auf dem Bildschirm ausgegeben.

1. Berechnen Sie „von Hand“ die Lösung für $n = 3$.
2. Lösen Sie für $n = 1, 2, \dots, 10$ das Gleichungssystem $A(n) \cdot \vec{x} = \vec{b}$ mit beiden oben genannten Funktionen und vergleichen Sie die Ergebnisse miteinander und für $n = 3$ mit der analytischen Lösung aus (1).
3. Vergleichen Sie die Lösungen aus (2) mit den Ergebnissen der *octave*-Funktion `x3 = invhilb(n)*b`.

Die Hilbert-Matrizen sind ein Beispiel für schlecht konditionierte Matrizen, die zu numerischen Fehlern in Algorithmen führen können. Dieser Effekt sollte in dieser Aufgabe auftreten und beobachtet werden können. Die letzte Methode mit der Funktion `invhilb` sollte zu numerisch stabilen Ergebnissen führen.

Übung 4 Izhikevich-Modell mit C++ (5 Punkte)

In der Vorlesung wurde ein Punkt-Neuronen-Modell von Izhikevich und eine Implementierung mit *Octave* vorgestellt (Quellcode ist auf der Homepage bereitgestellt). In dieser Aufgabe soll das Modell mit C++ implementiert werden und die Performance beider Implementierungen (C++, *Octave*) verglichen werden.

1. Implementieren Sie das Modell in C++ mit den gleichen Parametern wie in *Octave* und überprüfen Sie die Ergebnisse auf Richtigkeit (Vergleich mit der *Octave*-Ausgabe). Um ein Programm in C++ zu implementieren, öffnen Sie im Pool eine Datei in einem Editor (z.B. `gedit`) und speichern den Code in einer Datei mit der Endung `.cc`:

```
$ gedit hallowelt.cc &
```

Das `&` bewirkt, daß der Editor im Hintergrund geöffnet wird, und man trotzdem in der Konsole weiterarbeiten kann. Das aus den Übungen bekannte Hallo, Welt!-Programm kann dann etwa wie folgt aussehen:

```
1 // include i/o library
2 #include <iostream>
3
4 // main is always the first function to be called
5 int main(int argc, char** argv)
6 {
7     std::cout << "Hello, World..." << std::endl;
8
9     return 0;
10 }
```

Um das Programm zu kompilieren, können Sie den C++-Compiler `g++` verwenden:

```
$ g++ -o hallowelt hallowelt.c
```

Die Option `-o <name>` erstellt ein ausführbares Programm mit Namen `<name>`. Dieses kann dann mit folgendem Befehl in der Konsole ausgeführt werden:

```
$ ./hallowelt
```

2. Messen Sie die Rechenzeiten für die C++- und *Octave*-Implementierungen für $t = 2000ms$ und $dt = \{0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125\}ms$. Achten Sie darauf, daß Sie ausreichend lange Rechnungen machen, da die Zeitmessung einerseits selber mit einem Fehler im Sekundenbereich behaftet ist und andererseits der C++-Code sonst zu schnell ist. Bei mir reichte es, für jedes dt aus der Liste 50 Durchläufe zu machen um hinreichend große Zeiten für die C++-Simulation zu erhalten (Die *Octave*-Simulation dauerte dann natürlich etwas länger). Gerne können Sie dieses Problem anders lösen. Jede Simulation mit 50 Läufen wurde dann noch jeweils drei mal ausgeführt um zu mitteln.

Stellen Sie die Zeiten in einem doppelt-logarithmischen Plot Zeitschrittweite-Rechenzeit gegenüber und untersuchen Sie, ob es ein bestimmtes Verhältnis der Rechenzeiten gibt. Auf jeden Fall ist *Octave* um Größenordnungen langsamer – haben Sie eine Idee, woran das liegt?

Die Rechenzeiten können im Pool und generell unter Unix/Linux am einfachsten mit den Befehlen

```
$ /usr/bin/time -f %e octave -q izhikevich.m > oct.dat
$ /usr/bin/time -f %e ./izhikevich_cpp > cpp.dat
```

gemessen werden. Hierbei ist `izhikevich.m` ein *Octave*-Skript und `izhikevich_cpp` das C++-Executable. Die Rückgabe des Befehls ist die verstrichene Echtzeit in Sekunden. Die Umleitung nach dem Befehl mittels `> datei.dat` bewirkt, daß eventuelle Ausgaben des Programms in eine Datei umgeleitet werden.