Institut für Numerische und Angewandte Mathematik
FB Mathematik und Informatik der Universität Münster
Prof. Dr. Christian Engwer, Dipl. Math. Dipl. Inf. Sebastian Westerheide

15.05.2012

Übung zur Vorlesung

Wissenschaftliches Rechnen

SS 2012 — Blatt 4

Abgabe: 22.05.2012, 10:00 Uhr, Briefkasten 89
Code zusätzlich per e-mail an `sebastian.westerheide@uni-muenster.de`

Aufgabe 1 (n-Körper-Problem aus Sicht des Schwerpunktsystems) (4 Punkte)

Wir betrachten erneut das n-Körper-Problem mit den Notationen aus der Vorlesung. Der Schwerpunkt R der n Körper ist definiert durch

$$R := \frac{1}{M} \sum_{i=0}^{N-1} m_i r_i \quad \text{mit} \quad M := \sum_{i=0}^{N-1} m_i,$$

seine Geschwindigkeit V ergibt sich durch

$$V := \frac{1}{M} \sum_{i=0}^{N-1} m_i v_i.$$

Zeigen Sie, dass sich R mit konstanter Geschwindigkeit bewegt.

Aufgabe 2 (Implizites Euler-Verfahren) (7 – 10 Punkte)

Gegeben sei ein System von n autonomen Differentialgleichungen 1. Ordnung

$$y' = f(y) \quad \text{auf} \quad [0,T], \qquad y = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{pmatrix}, \qquad f : \mathbb{R}^n \to \mathbb{R}^n, \qquad T \in \mathbb{R}^+ \tag{1}$$

und Anfangswerte $y(0) \in \mathbb{R}^n$. Dieses Anfangswertproblem wollen wir mit dem impliziten Euler-Verfahren

$$\begin{aligned} y_0 &:= y(0), \\ y_{k+1} &:= y_k + \Delta t \cdot f(y_{k+1}), \quad k = 0, 1, \ldots \end{aligned} \tag{2}$$

lösen. Bislang haben Sie in der Übung nur explizite Verfahren kennengelernt. Im Gegensatz zu expliziten Verfahren kommt die Approximation $y_{k+1} \approx y(t_{k+1})$ bei impliziten

Verfahren auch auf der rechten Seite der Iterationsvorschrift vor, d.h. es ist ein Gleichungssystem zu lösen. Beim betrachteten impliziten Euler-Verfahren ergibt sich aus (2) das äquivalente, im Allgemeinen nichtlineare Gleichungssystem

$$\mathcal{R}_k(y_{k+1}) = 0 \quad \text{mit} \quad \mathcal{R}_k : \mathbb{R}^n \to \mathbb{R}^n, \quad \mathcal{R}_k(z) := z - y_k - \Delta t \cdot f(z).$$

Wir müssen also in jedem Schritt die Nullstelle der möglicherweise nichtlinearen, mehrdimensionalen Funktion $\mathcal{R}_k$ bestimmen. Falls diese differenzierbar ist und (auf ganz $\mathbb{R}^n$) eine invertierbare Jakobi-Matrix $J_k(z) \neq 0$ besitzt, können wir zu diesem Zweck das mehrdimensionale Newton-Verfahren

$$x_{l+1} := x_l - \Big(J_k(x_l)\Big)^{-1} \mathcal{R}_k(x_l), \quad l = 0, 1, \ldots \tag{3}$$

verwenden.

(a) Implementieren Sie das implizite Euler-Verfahren für die Lösung der obigen Klasse von Anfangswertproblemen. Benutzen Sie zur Lösung des Gleichungssystems das Newton-Verfahren.

- Überlegen Sie sich, wie die Jakobi-Matrix von $\mathcal{R}_k$ mit der Jakobi-Matrix von f zusammenhängt
- Benutzen Sie folgendes Interface `VectorFunction` für differenzierbare Funktionen $g : \mathbb{R}^n \to \mathbb{R}^n$

```
#ifndef VECTORFUNCTION_HH
#define VECTORFUNCTION_HH

#include <vector>

/**
 * Interface zur Repraesentation von differenzierbaren Funktionen
 * g: R^n --> R^n.
 */
class VectorFunction
{
public:
  typedef std::vector<double> VectorType;
  typedef std::vector<VectorType> MatrixType;

  // Rein virtuelle Methoden zum Auswerten der Funktion und ihrer
  // Jakobimatrix
  virtual void evaluate (const VectorType& x, VectorType& result) const = 0;
  virtual void evaluateJacobian (const VectorType& x,
                                 MatrixType& jacobian) const = 0;
};

#endif
```

- Implementieren Sie die Iterationsvorschrift (2) als Prozedur

```
void implicit_euler_step (const VectorFunction::VectorType& y_k,
                          double dt, const VectorFunction& f
                          VectorFunction::VectorType& y_next);
```

- Implementieren Sie das Newton-Verfahren (3) als Prozedur

```
void newton_method (const VectorFunction::VectorType& x_0,
                    const VectorFunction& R, double eps,
                    VectorFunction::VectorType& solution);
```

- Beschränken Sie sich bei der Implemetierung des Newton-Verfahrens auf den Fall $n \in \{1, 2, 3\}$, für den sich die Invertierung einer regulären $n \times n$-Matrix explizit hinschreiben lässt (siehe Lineare Algebra I); behandeln Sie den Fall $n > 3$ mit einer Exception oder fangen Sie ihn mit einer Assertion ab
- Benutzen Sie zudem das Abbruchkriterium von Blatt 2 (Fehlerschranke `eps`)
- Verwenden Sie in jedem Schritt des impliziten Euler-Verfahrens den Startwert $x_0 = y_k$ für das Newton-Verfahren

(b) Testen Sie Ihre Implementierung an folgendem System, welches den diffusionslosen Spezialfall des Gierer-Meinhardt-Modells aus der Vorlesung darstellt:

$$a' = s\left(\frac{a^2}{b} + b_a\right) - r_a a,$$
$$b' = sa^2 - r_b b - b_b.$$

- Überlegen Sie sich, wie in diesem Fall die Funktion y und die rechte Seite f aus (1) aussehen
- Leiten Sie vom Interface `VectorFunction` eine entsprechende Klasse für die rechte Seite ab
- Verwenden Sie testweise die Parameter $s = r_a = r_b = b_a = b_b = 1$, die Anfangswerte $a(0) = 17$ sowie $b(0) = 42$, die maximale Zeit $T = 50$ und die Schrittweite $\Delta t = 0.01$.

  Hinweis: Diese Konfiguration ist ungetestet, es muss sich also nicht unbedingt ein interessantes Ergebnis einstellen
- Plotten Sie die zeitliche Entwicklung der Konzentrationen a und b

(c) Da die Systemgröße n schon beim Kompilieren feststeht, ist es eigentlich nicht sinnvoll mit einer dynamischen Datenstruktur wie `std::vector` zu arbeiten. Effizienter ist es in diesem Fall, feldartige Datenstrukturen wie `std::array` zu benutzen (bzw. `std::tr1::array`, falls der Compiler älter ist). Sie bekommen drei Bonuspunkte, falls Sie statt des obigen Interfaces das folgende, Template-basierte Interface und die dafür benötigten Prozedur-Signaturen zur Bearbeitung der Aufgabe verwenden:

```
#ifndef VECTORFUNCTION_HH
#define VECTORFUNCTION_HH

#include <tr1/array>  // #include <array> in C++11

/**
 * Interface zur Repraesentation von differenzierbaren Funktionen
 * g: R^n --> R^n.
 */
template <int n>
class VectorFunction
{
public:
  typedef std::tr1::array<double,n> VectorType;
  typedef std::tr1::array<VectorType,n> MatrixType;

  // Rein virtuelle Methoden zum Auswerten der Funktion und ihrer
  // Jakobimatrix
  virtual void evaluate (const VectorType& x, VectorType& result) const = 0;
  virtual void evaluateJacobian (const VectorType& x,
                                 MatrixType& jacobian) const = 0;
};

#endif
```