Rheinisch-Westfälische Technische Hochschule
Institut für Geometrie und Praktische Mathematik

Mathematisches Praktikum (MaPra) — Sommersemester 2011

Prof. Dr. Wolfgang Dahmen — M.Sc. Mathieu Bachmann, Dipl.-Math. Jens Berger, M.Sc. Liang Zhang

Aufgabe 5

Bearbeitungszeit: vier Wochen (bis Montag, den 20. Juni 2011)

Testattermin: Donnerstag, der 30. Juni 2011

Mathematischer Hintergrund: Numerische Lösung gewöhnlicher Differentialgleichungen durch explizite Runge–Kutta–Verfahren mit Schrittweitensteuerung — Mehrkörperproblem

Elemente von C++: Klassen, Templates

20 Punkte

Aufgabenstellung

Schreiben Sie die von Ihnen entwickelten Matrix- und Vektorklassen in Templateklassen um, wobei der Datentyp der Einträge noch festzulegen ist. Benutzen Sie diese Templateklassen, um ein Programm zur Lösung von gewöhnlichen Differentialgleichungen mittels expliziter Runge–Kutta–Verfahren mit Schrittweitensteuerung zu erstellen, und wenden Sie es auf das Mehrkörperproblem an.

Runge–Kutta–Verfahren

Gegeben seien eine gewöhnliche Differentialgleichung und ein Anfangswert

$$\frac{d}{dt}y(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

mit dem Vektorfeld $f : \mathbb{R}^{1+n} \to \mathbb{R}^n$ [3]. Die Lösung dieses Anfangswertproblems soll numerisch approximiert werden. Dazu bestimmt man die Näherungslösung $\widetilde{y}(t)$ zu diskreten Zeiten t_i mit $t_{i+1} = t_i + h_i$ für $i = 0, 1, \ldots$. Die Entwicklung von $y(t_{i+1})$ um t_i in eine Taylorreihe

$$y(t_{i+1}) = y(t_i + h_i) = y(t_i) + y'(t_i) \cdot h_i + \mathcal{O}(h_i^2) = y(t_i) + f(t_i, y(t_i)) \cdot h_i + \mathcal{O}(h_i^2).$$

liefert die Idee zu einem einfachen Verfahren, diese Werte zu berechnen. Bei der numerischen Näherungslösung vernachlässigt man die Terme höherer Ordnung und setzt

$$\widetilde{y}(t_{i+1}) := \widetilde{y}(t_i) + h_i \cdot f(t_i, \widetilde{y}(t_i)).$$

Berechnet man mit diesem sogenannten *expliziten Eulerverfahren* die Näherungslösung $\widetilde{y}(1)$ aus einem Startwert $\widetilde{y}(0) = y(0)$ über mehrere Zwischenschritte, so macht man (bei hinreichend glattem f) einen Fehler der Ordnung $\mathcal{O}(h)$, wobei h die größte auftretende Schrittweite ist. Für $h \to 0$ konvergiert $\widetilde{y}(1)$ somit gegen die exakte Lösung. Leider sind für die Praxis sehr kleine Schrittweiten und damit sehr viele Schritte notwendig, was zu hohem Rechenaufwand und Problemen mit Rundungsfehlern führt. Daher ist man an Verfahren interessiert, deren Fehler von der Form $\mathcal{O}(h^p)$ mit $p > 1$ sind. Eine Möglichkeit wäre, mehr Glieder der Taylorreihe zur Berechnung heranzuziehen. Dabei lassen sich die höheren Ableitungen von y durch Ableitungen von f ausdrücken. Diese Ableitungen auszurechnen ist jedoch meist sehr aufwendig oder gar unmöglich (z.B. wenn f nur als eine Folge von Messwerten gegeben ist).

Eine wesentlich einfacher zu handhabende Verallgemeinerung des Euler–Verfahrens stellen die sogenannten *expliziten Runge–Kutta–Verfahren* dar. Hier wird f an mehreren Stellen ausgewertet und ein geeigneter Mittelwert dieser Ergebnisse anstelle von $f(t_i, \widetilde{y}(t_i))$ verwendet.

Dies geschieht in der folgenden Weise[1]:

$$K^j = f(t + \alpha_j h,\ \widetilde{y}(t) + h \cdot \sum_{l=1}^{j-1} \beta_{jl} K^l) \quad \text{für } j = 1, \ldots, m,$$

$$\widetilde{y}(t+h) = \widetilde{y}(t) + h \cdot \sum_{l=1}^{m} \gamma_l K^l.$$

Lässt man auch noch Koeffizienten β_{jl} für $l = j, \ldots, m$ zu, so spricht man von *impliziten Runge–Kutta–Verfahren*, weil zunächst ein implizites Gleichungssystem für die Werte $K^1, \ldots, K^m$ gelöst werden muss.

Die notwendigen Parameter α_j, β_{jl} und γ_l werden in einer Tabelle (Butcher–Tableau)

α_j	β_{jl}
	γ_l

zusammengefasst.

Die Berechnung der Koeffizienten für Verfahren mit möglichst hoher Ordnung p, aber möglichst wenig Stufen m ist äußerst aufwendig [2]. Daher seien als Beispiele expliziter Runge–Kutta–Verfahren hier nur das Euler–Verfahren der Ordnung $p = 1$ und das klassische Runge–Kutta–Verfahren der Ordnung $p = 4$

0	0
	1

0				
1/2	1/2			
1/2		1/2		
1			1	
	1/6	1/3	1/3	1/6

angeführt.

Schrittweitensteuerung

Benutzt man zwei Runge–Kutta–Verfahren verschiedener Fehlerordnung, so kann man die Differenz der Ergebnisse als Schätzung für den Fehler verwenden, der in einem Schritt gemacht wurde. Die beiden Verfahren sind hier so konstruiert, dass sie sich nur in den Koeffizienten γ_l unterscheiden; die Werte für das zweite Verfahren seien mit $\overline{\gamma}_l$ bezeichnet. Die Fehlerschätzung lautet dann

$$e(t+h) = \widetilde{y}(t+h) - \overline{y}(t+h) = h \cdot \sum_{l=1}^{m} (\gamma_l - \overline{\gamma}_l) K^l =: h \cdot \sum_{l=1}^{m} \delta_l K^l.$$

Die Schrittweite wird nach jedem Schritt wie folgt modifiziert:

$$h_{\text{neu}} = h \cdot \begin{cases} 0.1 & \text{falls } c < 0.1 \\ 5 & \text{falls } c > 5 \\ c & \text{sonst} \end{cases}, \qquad \text{mit} \quad c = 0.9 \cdot \left(\frac{\varepsilon_{\max}}{\varepsilon}\right)^{\frac{1}{p+1}}.$$

Diese Art der Schrittweitensteuerung bewirkt folgendes: Falls der Fehler groß ist, so wird die Schrittweite verkleinert. Im Gegenzug dazu wird die Schrittweite vergrößert, wenn der Fehler klein ist. Wenn die Fehlernorm $\varepsilon = \|e(t+h)\|_\infty$ größer als die vorgegebene Schranke $\varepsilon_{\max}$ ist, so verwirft man darüberhinaus das zunächst berechnete Ergebnis, und berechnet diesen Schritt solange neu (mit jeweils modifizierter Schrittweite), bis der Fehler klein genug geworden ist. Der Sicherheitsfaktor 0.9 vermeidet, daß Schritte zu oft verworfen werden. Die Grenzen 0.1 und 5 für c unterdrücken zu starkes Springen der Schrittweiten. Der Faktor 5 sorgt insbesondere dafür, dass nicht genauer (und damit vor allem auch langsamer) als nötig gerechnet wird.

[1] Der Index i für den Zeitschritt wurde der Übersichtlichkeit halber weggelassen.

Mehrkörperproblem

Als Beispiel für die vielen gewöhnlichen Differentialgleichungen, die in den Naturwissenschaften auftauchen, sei hier das Mehrkörperproblem aus der klassischen (Himmels-)Mechanik herausgegriffen. Gegeben seien n Körper mit den Massen m_i, $1 \leq i \leq n$, die sich zum Zeitpunkt t an den Orten $x_i(t) \in \mathbb{R}^2$ aufhalten[2]. Laut Newtonschem Gravitationsgesetz übt der j–te Körper auf den i–ten Körper die Gravitationskraft

$$F_{ij} = G\frac{m_i m_j}{\|x_j - x_i\|_2^3}(x_j - x_i) \quad \text{mit} \quad \|F_{ij}\|_2 = \|F_{ji}\|_2 = G\frac{m_i m_j}{\|x_j - x_i\|_2^2}$$

aus, wobei $G \approx 6{,}67 \cdot 10^{-11}\,\frac{\mathrm{m}^3}{\mathrm{kg\,s}^2}$ die Gravitationskonstante ist. Durch die Summe aller Gravitationskräfte (es wirken keine weiteren Kräfte) erfährt der i–te Körper eine Beschleunigung $\ddot{x}_i := \frac{d^2}{dt^2}x_i$ gemäß

$$m_i\ddot{x}_i = \sum_{j\neq i} F_{ij} = Gm_i \sum_{j\neq i} \frac{m_j}{\|x_j - x_i\|_2^3}(x_j - x_i) \quad \text{bzw.} \quad \ddot{x}_i = G\sum_{j\neq i} \frac{m_j}{\|x_j - x_i\|_2^3}(x_j - x_i).$$

Weil die Ortsvektoren x_i jeweils zwei (drei) Komponenten haben, ergibt dies ein System von $2n$ ($3n$) gewöhnlichen Differentialgleichungen zweiter Ordnung

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

Mit Hilfe der Substitution $y_0 := \mathbf{x}$ (Orte der Körper) und $y_1 := \dot{\mathbf{x}}$ (Geschwindigkeiten der Körper) erhält man daraus ein System von $4n$ ($6n$) Differentialgleichungen erster Ordnung

$$\dot{y} = \begin{pmatrix}\dot{y}_0\\ \dot{y}_1\end{pmatrix} = \begin{pmatrix}\dot{\mathbf{x}}\\ \ddot{\mathbf{x}}\end{pmatrix} = \begin{pmatrix}y_1\\ \mathbf{f}(y_0)\end{pmatrix} =: f(y).$$

Da f nicht explizit von t abhängt, spricht man von einem autonomen Differentialgleichungssystem.

Templates

Um Ihre Vektorklasse auf einem anderen Typ als `double` operieren zu lassen, können Sie entweder überall den Typ (z.B. durch *int*) ersetzen, oder aber Ihre Vektorklasse in ein sogenanntes *Template* umschreiben. Das funktioniert so: Angenommen Ihre bisherige Klasse operiert auf dem Datentyp *double* und hat die Form:

```
class vector {
  private:
    double*          m_p;          // Zeiger auf Speicher
    unsigned int     m_nLen;       // Laenge des Vektors

    // ...

    double& operator[](unsigned int n) { return m_p[n]; }
    const double& operator[](unsigned int n) const { return m_p[n]; }

    // ...
};
```

[2]Da die Bildschirmausgabe nur zweidimensional ist, wird hier in $\mathbb{R}^2$ statt in $\mathbb{R}^3$ gerechnet, um nicht unnötig Rechenzeit zu verschenken.

Dann ist

```
template <typename T>
class vector {
  public:
    typedef T element_type;       // Template-Datentyp merken

  private:
    element_type*   m_p;          // Zeiger auf Speicher
    unsigned int    m_nLen;       // Laenge des Vektors

    // ...

    element_type& operator[](unsigned int n) { return m_p[n]; }
    const element_type& operator[](unsigned int n) const { return m_p[n
       ]; }

    // ...
};
```

ein Template, welches durch Angabe eines beliebigen Typs eine Vektorklasse mit genau diesem Typ zur Verfügung stellt. Aus dem Template werden also Klassen generiert, Beispiel:

```
typedef vector<unsigned int> uintvector;
```

oder für den Verbund

```
typedef vector<element> elementvector;
```

Nun kann man die Klasse `uintvector` als einen Vektor operierend auf `unsigned int` nutzen. Der Compiler ersetzt `T` durch den in `< >` angegebenen Typ und verhält sich so, als wenn Sie diesen Typ direkt in die Klasse geschrieben hätten.

Implementierung

Durch die Header–Datei unit5.h werden Ihnen wieder einige Variablen und Funktionen zur Verfügung gestellt. Außerdem benötigen Sie Ihre Vektor- und Matrixklasse sowie die Grafik–Bibliothek in IGL.o, die wir Ihnen zur Verfügung stellen.

Verwenden Sie für alle Berechnungen den Typ `real`, dessen Deklaration

```
typedef long double real;
```

Sie in unit5.h finden. So könnten Sie gegebenenfalls leicht den Datentyp ändern. Ebenso finden Sie die Deklarationen:

```
typedef TVektor<real> Vektor;

typedef TMatrix<real> Matrix;
```

Sehen Sie sich in tvektor.h die Deklaration der Funktion `Norm` für verschiedene Datentypen an, und denken Sie daran, für die Wurzel die Funktion `sqrtl` zu verwenden. Der Funktionszeigertyp

```
typedef Vektor (*FunktionVF) ( real t, const Vektor &y );
```

ist für das Vektorfeld[3] gedacht. Zunächst ist die Funktion

```
void Start ( int Bsp, Vektor &Masse, FunktionVF &f, Vektor &y0,
             real &tAnf, real &tEnd, real &h0, int Grafik, int Ausgabe );
```

aufzurufen, wobei `Bsp` wieder die Werte 1 bis `AnzahlBeispiele` durchlaufen kann. `Grafik=1` ruft die Grafikausgabe auf, `Grafik=0` schaltet sie aus. Der Parameter `Ausgabe` steuert, ob eine Ausgabe erfolgen soll oder nicht. Der Anfangswert des Problems steht dann in `y0`, die Anfangszeit in `tAnf`, die Zeit, bis zu der die Differentialgleichung gelöst werden soll, in `tEnd` und schließlich die Startschrittweite in `h0`.

Die rechte Seite der Differentialgleichung ergibt sich wie folgt: Ist der Vektor `Masse` der Nullvektor, dann ist `f` bereits die benötigte Funktion. Ist `Masse` jedoch nicht der Nullvektor, dann müssen Sie die Funktion für die rechte Seite selbst aufstellen. Dazu benötigen Sie die Massen m_i der einzelnen Himmelskörper, die Sie im Vektor `Masse` finden. Die Gravitationskonstante G wird durch die Konstante `Grav` geliefert. Obwohl Sie für diese rechte Seite den Parameter t nicht benötigen, sollte Ihre Funktion dennoch vom Typ `FunktionVF` sein, damit sie Ihren Differentialgleichungslöser nicht modifizieren müssen.

Nun fehlen nur noch die Parameter für das Runge–Kutta–Verfahren. Diese sind gegeben durch die Vektoren `Alpha` und `Gamma`, die Matrix `Beta` sowie für die Schrittweitensteuerung durch den Vektor `Delta`, die Ordnung p und die Fehlertoleranz $\varepsilon_{\max}$ in `eps`, die Ihnen als globale Variablen zur Verfügung gestellt werden[4]. Schreiben Sie eine Funktion

```
void RKSchritt ( FunktionVF f, real &t, Vektor &y, real &h );
```

die jeweils einen (nicht verworfenen) Schritt des Runge–Kutta–Verfahrens durchführt und eine neue Schrittweite h für den nächsten Schritt schätzt. Zu Beginn und nach jedem Schritt sollten Sie die Funktion

```
void Schritt_Ausgabe ( real t, const Vektor &y, int Grafik, int Ausgabe );
```

aufrufen, die die Schritte grafisch darstellt. Wenn Sie die Zeit `tEnd` erreicht haben (und nicht darüber hinaus geschossen sind), müssen Sie noch die Funktion

```
void Ergebnis ( real t, const Vektor &y, int Ausgabe );
```

aufrufen, die Ihr Resultat bewertet.

Literatur

[1] Dahmen, W. und A. Reusken: *Numerische Mathematik für Ingenieure und Naturwissenschaftler*. Springer Verlag, Heidelberg, 2. Auflage, 2008.

[2] Hairer, E., S. P. Nørsett und G. Wanner: *Solving Ordinary Differential Equations I: Nonstiff Problems*. In: *Springer Series in Comput. Mathematics*, Band 8. Springer Verlag, Heidelberg, 2. überarbeitete Auflage, 1993.

[3] Walter, W.: *Gewöhnliche Differentialgleichungen*. Springer Verlag, Heidelberg, 7. Auflage, 2000.

[3] Das Vektorfeld ist hier die rechte Seite der Differentialgleichung.
[4] Diese werden beim Aufruf von `Start` jeweils neu gesetzt.