Physik auf dem Komputer

Michael Kopp

Version α 0.1 — 6. Juni 2010

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Inhalte von Physik am Komputer

Physik auf dem Komputer besteht aus drei Teildisziplinen:

1. Physik
2. Nummerische Analysis (Mathe)
3. Komputerprogrammierung (Informatik)

Praktisch bedeutet dies

Auswertung Experimenteller Daten
- Extrapolation: Von gegeneben Datenpunkten auf unbekannte Schließen
- Interpolation: Kurven schön an gegebene Datenpunkte anfitten
- Daten Filtern
- etc.

Nummerisches Rechnen : Berechnung Physikalischer Probleme; dazu: Lösen von Differenzial- und Integralgleichungen.

Nummerische Modellexperimente Der Komputer fungiert als Labor. Hier ist es möglich, unzugängliche Orte oder nicht messbare Systeme zu „untersuchen“, bspw. Elementarteilchen, Sonne, Wetter. Außerdem kann man hier die Umweltparameter selbst einstellen, bspw. Schwerkraft etc.

1.2 Erstes Beispiel: Feder-Kugel-System

1.2.1 Formulierung des Systems

An einer *massenlosen* Feder der *Federhärte* $9\,\mathrm{N/m}$ gleitet ein *Gewicht* von 1kg *ohne Reibung* in x-Richtung. Die Bewegung resultiert ausschließlich aus der Federkraft. Die *Ruhelage* liegt bei $x_0 = 0\,\mathrm{m}$ zur Zeit $t = 0\,\mathrm{s}$.

Gesucht ist die Auslenkung zu Zeiten $t = n \cdot \frac{1}{10}\mathrm{s}$ mit $n \in [0, 2] \cap \mathbb{N}_0$.

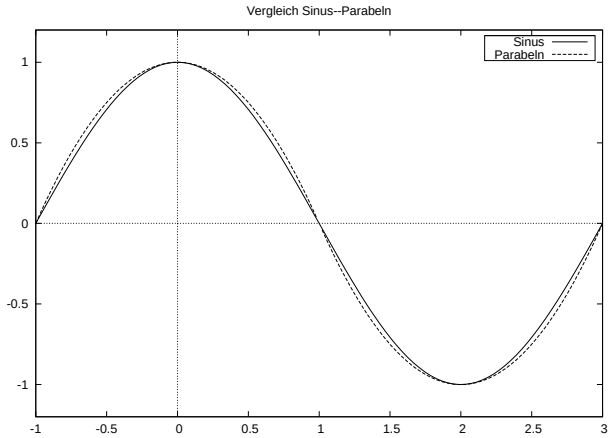


Abbildung 1.1: Vergleich Sinus – Parabel

1.2.2 Allgemein: Unterschied *Modell* vs. *Gesetz*

Die Theoretische Physik soll allgemeine *Gesetze* finden, wie bspw. die MAXWELL-Gleichungen. Für eine Nutzanwendung leitet man spezielle *Modelle* (bspw. das OHMsche „Gesetz“[1]) aus den Gesetzen her. Mit *diesen* kann man Voraussagen treffen und experimentell bestätigen bzw. negieren. Diese Modelle kann man auch mit dem Komputer aufstellen, durchrechnen und die so erhaltenen Daten dann mit der Realität abgleichen.

Gesetze geben also Anleitung zum Aufstellen von *Modellen*.

Beispielsweise kann man die Schwingung des besagten Drahtes *modellieren* durch eine Harmonische oder eine Parabolische Schwingung – diese beiden unterscheiden sich nur wenig, vlg Abb. 1.1.

Ein schöneres *Modell* kann man finden, wenn man die *Gesetze* von NEWTON

$$\frac{\mathrm{d}\,p}{\mathrm{d}\,t} = F \text{ mit } p := m \cdot \frac{\mathrm{d}\,x}{\mathrm{d}\,t} \tag{1.4}$$

und HOOKE

$$F = -k \cdot x \tag{1.6}$$

verwendet.

1.2.3 Mathematische Formulierung

Mit ebendiesen Gleichungen kommt man auf

$$m \cdot \ddot{x} = -k \cdot x \text{ mit } x(t = 0\mathrm{s}) = 1\mathrm{m} \text{ und } \dot{x}(t = 0\mathrm{s}) = 0\,{}^{\mathrm{m}}\!/_{\mathrm{s}}\,. \tag{1.7}$$

Entdimensionalisierung

Wichtig! 1 *Der Komputer kann nicht mit Einheiten rechnen!*

[1] Sollte eigentlich *Ohmsches Modell* heißen

Dies bringt den Vorteil weniger Parameter in den Gleichungen; diese sind leicher zu verallgemeinern.

$$x \mapsto \xi \cdot u \text{ und } t \mapsto \tau \cdot s \tag{1.10}$$

Wichtig! 2 *Die Einheiten ξ und τ müssen so gewählt werden, dass die reinen Zahlen u und s im leicht darstellbaren Bereich liegen.*

Der Komputer macht leicht Fehler, wenn er mit sehr großen oder sehr kleinen Zahlen rechnen muss.

Formel (1.7) wird dann mit den Ersetzungen (1.10) zu

$$m \frac{1}{\tau^2} \frac{\mathrm{d}^2}{\mathrm{d}\, s^2}(\xi\, u) = -k\, \xi\, u \Rightarrow u'' = -C\, u \text{ mit } C = \frac{k\, \tau^2}{m}\,, \tag{1.11}$$

weil

$$\frac{\mathrm{d}}{\mathrm{d}\, t} = \frac{\mathrm{d}}{\mathrm{d}(\tau\, s)} = \frac{1}{\tau} \cdot \frac{\mathrm{d}}{\mathrm{d}\, s} \Leftrightarrow \dot{\psi} = \frac{1}{\tau} \psi'\,. \tag{1.10†}$$

Die Anfangsbedingungen sind dann

$$u(0) = 1 \text{ und } u'(0) = 0\,. \tag{1.12}$$

1.2.4 Analytische Lösung

$$u(s) = A \cdot sin(\omega\, s + \varphi)\ , \omega^2 = C\,. \tag{1.13}$$

Die Anfangsbedingungen liefern

$$\begin{cases} \varphi = \arctan\left(\frac{u_0}{v_0}\omega\right) \text{ und } A = \frac{u_0}{\sin\varphi} & v_0 \neq 0 \\ \varphi = \frac{\pi}{2} \text{ und } A = u_0 & v_0 = 0 \end{cases}\,; \tag{1.14}$$

für uns ergibt sich also

$$u(s) = 1 \cdot \cos(3 \cdot s)\,. \tag{1.15}$$

1.2.5 Numerische Lösung

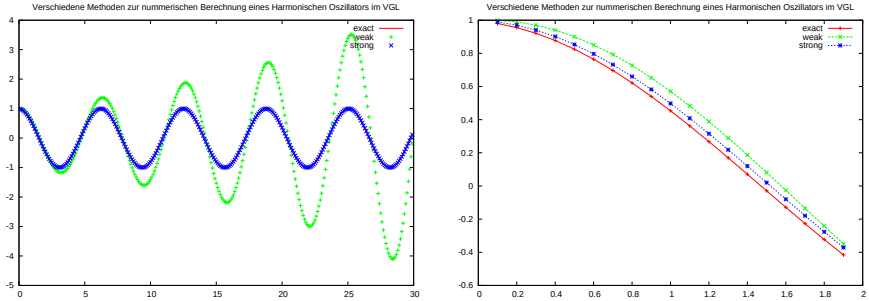
Analytische Lösung verwenden Der Komputer berechnet $cos(3 \cdot n\, \Delta s)$ mit $n \in [0, 20] \cap \mathbb{N}_0$ und $\Delta s = \frac{1}{10}$.

Problem: Der Komputer kennt $\cos(x)$ als

$$\cos(x) = \lim_{N \to \infty} \sum_{k=0}^{N} \frac{(-1)^k}{(2\, k)!} x^{2k}\,. \tag{1.21}$$

Für kleine Zahlen x und große k wird x^{2k} *sehr* klein; der Komputer bekommt Probleme.

Wird die DGL geändert – bspw $u'' = -C\, u \cdot s$ – so kann man die Lösung nicht mehr geschlossen analytisch bestimmen, bzw. nur mit transzendenten Funktionen, die der Komputer nicht kennt.



(a) Im Großen (b) Im Kleinen

Abbildung 1.2: Vergleich von starkem und schwachen Algorithmus

Numerischer Algorithmus Führt man die Hilfsgröße

$$v := u' \tag{1.22}$$

ein, so bekommt man das Gleichungssystem

$$u' = v \text{ und } v' = -C \cdot u \,. \tag{1.23}$$

Nun muss man sich noch überlegen, wie man mit dem Komputer gut ableitet. Die Ableitung ist definiert als

$$\frac{\mathrm{d}\, u}{\mathrm{d}\, s} = \lim_{\Delta s \to 0} \frac{u(s + \Delta s) - u(s)}{\Delta s} \,. \tag{1.24}$$

Wichtig! 3 *Der Komputer kann jedoch keine* lim *ausführen.*

Man nähert nun also an, indem man Δs nicht gegen 0 sondern gegen eine sehr kleine Zahl gehen lässt.

Hier ist die Einführung von v auch ein großer Vorteil: Bei einer zweiten Ableitung, wie sie noch in (1.11) vorkommt, hätte man eine sehr kleine Zahl durch eine sehr kleine geteilt – das wird im Komputer (sehr) ungenau.

Durch Umformen von (1.23) kommt man auf

$$\begin{aligned} v(s + \Delta s) &\approx v(s) - C \cdot u(s) \cdot \Delta s \\ u(s + \Delta s) &\approx u(s) + v(s) \cdot \Delta s \end{aligned} \,. \tag{1.25}$$

Die rechte Seite dieser Gleichungen hängt nur von s ab, die linke von $s + \Delta s$. Da ein Anfangspunkt bekann ist, kann aus Kenntnis von u und v zu einem Zeitpunkt s stets auf $s + \Delta s$ geschlossen werden. Wir haben also eine rekursive Formel gefunden.

Leider ist dieser Algorithmus völlig instabil; vgl. Abb. 1.2.

Um die „starke“ Version zu erstellen verwendet man in dem Gleichungssystem anstatt $v(s)$ auf der rechten Seite untenein $v(s + \Delta s)$. Dieses Gleichungssystem

$$\begin{aligned} v(s + \Delta s) &\approx v(s) - C \cdot u(s) \cdot \Delta s \\ u(s + \Delta s) &\approx u(s) + v(s + \Delta s) \cdot \Delta s \end{aligned} \tag{1.27}$$

enthält die Energieerhaltung besser[2] und liefert deshalb bessere Ergebnisse.

[2] dazu später mehr

1.2.6 Implementierung

Vergleiche Listing 1.1.

Listing 1.1: Simulation Harmonischer Oszillator

```
1 #include<iostream>
2 using namespace std;
3 #include<cmath> // for cos, sqrt
4
5 int main()
6 {
7         // declaration
8         double k, dt, tmax, v0, x0;
9         // user input
10        cout << "#Federhaerte k:          "; cin >> k;
11        cout << "#Anfangsauslenkung:      "; cin >> x0;
12        cout << "#Anfangsgeschw.:         "; cin >> v0;
13        cout << "#Intervalldauer:         "; cin >> tmax;
14        cout << "#Groesse Zeitschritt:    "; cin >> dt;
15
16        // // internal variables
17        // weak
18        double xw[2], vw[2]; // x[0]: x(t), x[1]: x(t+dt) dito
              v[.]
19        // strong
20        double xs[2], vs[2];
21        // analytic
22        double xa, omega, ampl, phase;
23        //frequency
24        omega = sqrt(k);
25        int use_ana = 1; // if k<0 sqrt(k) is ... ungood.
              Output for x_analytic then is 0.
26        if ( k < 0 )
27        {
28                use_ana = 0;
29        }
30        // phase, amplitude
31        if ( x0 != 0 )
32        {
33                phase = atan(-v0/x0)/omega;
34                ampl = x0/cos(phase);
35        }
36        else
37        {
38                phase = -2.0 * atan(1.0); // -pi/2
39                ampl = v0/omega;
40        }
41
42        // initialize
43        xw[0]=x0; vw[0]=v0;
44        xs[0]=x0; vs[0]=v0;
45
46        // loop
47        cout << "# --- start computing --- ";
```

```
48          for ( double t=0; t <= tmax; t+=dt )
49          {
50                  // // compute
51                  // weak method
52                  vw[1] = - k * xw[0] * dt + vw[0];
53                  xw[1] = vw[0] * dt + xw[0];
54                  // strong method
55                  vs[1] = - k * xs[0] * dt + vs[0];
56                  xs[1] = vs[1] * dt + xs[0];
57                  // analytic
58                  xa = use_ana * ampl * cos(omega * (t+dt) +
                        phase);
59                  // output
60                  cout << t << "\t" << xw[0] << "\t" << xs[0] <<
                        "\t" << xa << "\n";
61                  // shift arrays x,v ([0] needs to be _current_
                        value
62                  xw[0]=xw[1]; vw[0]=vw[1];
63                  xs[0]=xs[1]; vs[0]=vs[1];
64          }
65          cout << "# --- loop done ---";
66  }
```

Kapitel 2

Interpolieren

Im Allgemeinen geht es bei Interpolieren und Approximieren darum, eine Funktion f oder Datenpunkte (x_i, y_i) durch eien Funktion g zu approximieren, sodass $\|f - g\|$ bzw. $\|g(x_i) - y_i\|$ minimal wird.

Man unterscheidet zwischen Interpolieren und Approximieren: Beim **Approximieren** Nähert man eine Funktion so an, dass sie eine Punktwolke möglichst gut beschreibt – das bedeutet, dass die Summe der Quadrate der Abweichungen minimal wird. Die schließlich gefundene Funktion läuft an vielen Funktionswerten vorbei. Beim **Interpolieren** jedoch findet man ein Funktion, die die Menge von Punkten *enthält* – also durch jeden einzelnen der vorgegebenen Punkte geht.

Es macht also Sinn, viele Messwerte, die von sich aus mit Fehlern (Messfehlern usw) behaftet sind, durch eine Funktion *approximieren*, wohingegen man bei nur sehr dünn gestreuten Punkten eher *interpoliert*.

Eine typische Aufgabe der Interpolation besteht darin, ein Polynom der Ordnung n durch $n + 1$ Stützstellen zu legen, sodass also

$$P_n(x_i) = y_i \text{ für } i \in \{0, ..., n\} \tag{2.1}$$

gilt.

Wichtig! 4 *Dieses Polynom ist eindeutig.*

Die Punkte (x_i, y_i) können dabei beispielsweise von einer zu approximierenden Funktion f stammen: $y_i = f(x_i)$.

2.1 Stützstellen

Möchte man die Funktion f durch ein Polynom P_n approximieren, so benötigt man die Stützstellen x_i, an denen die Punkte (x_i, y_i) als Grundlage verwendet werden sollen. Eine einfache Wahl der Stützstellen ist die *äquidistante*; im Intervall $[a, b]$ liefert

$$x_i = a + \frac{b-a}{n} \cdot i \quad , i \in \{0, ..., n\} \tag{2.2}$$

$n + 1$ Stützstellen.

Bei dieser Wahl wird jedoch der Fehler der approximierenden Polynome am Rand von $[a, b]$ sehr groß. Um dem entgegenzusteuern verwendet man die *Tschebyscheff-Punkte*:

$$x_i = \frac{a+b}{2} + \frac{a-b}{2} \cos\left(\frac{i}{n}\pi\right) \quad , i \in \{0, ..., n\} \; . \tag{2.3}$$

2.2 Lagrange-Interpolation

Eine schöne Möglichkeit zur Interpolation stammt von Lagrange; bei ihr verwendet man die *Lagrange-Polynome*

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)} \; . \tag{2.4}$$

Diese haben die praktische Eigenschaft

$$L_i(x_j) = \delta_{ij} \; . \tag{2.5}$$

Damit fällt es auch leicht, die Bedingung (2.1) zufriedenzustellen:

$$P_n(x) = \sum_{i=0}^{n} y_i L_i \; . \tag{2.6}$$

Aufgeschrieben sieht das schön aus – nur zum Programmieren ist es weniger angenehm.

Hier können wir jedoch schon ein Beispiel geben, wie die Tschebyscheff-Stützstellen die Fehler am Rand verkleinern; betrachte dazu die Abbildungen 2.1. Der Quellcode zu diesen Abbildungen ist in Listing 2.1.

Listing 2.1: Lagrange-Interpolation mit Tschebyscheff und äquidistanten Stützstellen

```
1  #include<iostream>
2  #include<cmath>
3
4  //// GLOBALE VARIABLEN ////
5  const int n = 30;          //anzahl stuetzstellen
6  const double a = 0;        //untere intervallgrenze
7  const double b = 5;        //obere intervallgrenze
8  const int N = 200;         //azahl an Ausgaberasterstellen
9  const double delx = (b-a)/N;     //raster fuer ausgabe
10
11
12 double f(double x) //function
13 {
14         return std::cos(10*x) * std::exp(-x);
15 }
16
17 int main()
18 {
19         //tschebyscheff-stuetzstellen
```

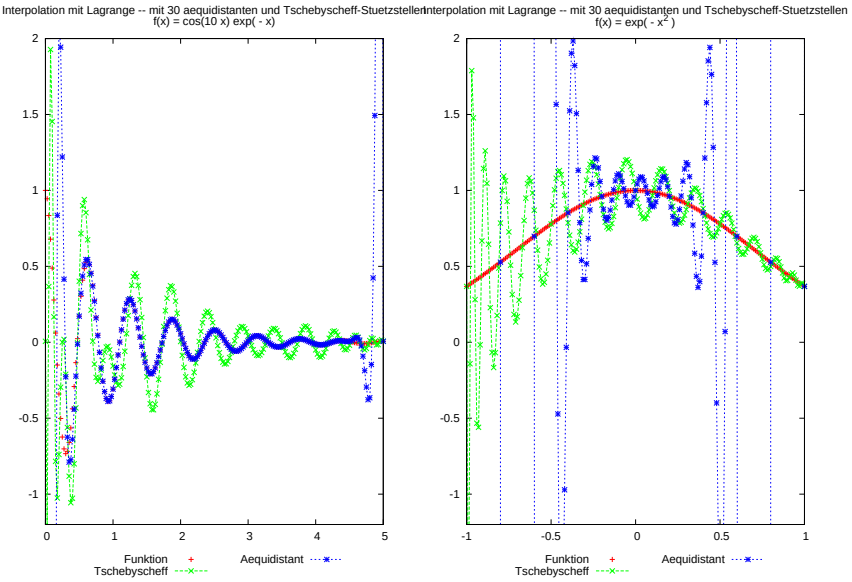


Abbildung 2.1: Lagrange-Interpolation

```
20          double xt[n]; double yt[n];
21          for(int i = 0; i <= n; i++)
22          {
23                  xt[i] = (a+b)/2 - (b-a)/2 * std::cos(M_PI*i/n)
                        ;
24                  yt[i] = f(xt[i]);
25          }
26
27          //aequidistante stuetzstellen
28          double xa[n]; double ya[n];
29          for(int i = 0; i <= n; i++)
30          {
31                  xa[i] = a + (b-a)/n * i;
32                  ya[i] = f(xa[i]);
33          }
34
35          //lagrange interpolation -- tschebyscheff
36          double apprLagTsch[N];
37          for (int k = 0; k <= N; k++)     //schleife ueber
                ausgaberaster
38          {
39                  double xi = a + delx * k;
40                  double summ = 0;          //fuer summe
41                  for (int i = 0; i <= n; i++)
42                  {
43                          double multip = 1;        //fuer produkt
44                          for (int j = 0; j <= n; j++)
45                                  multip *= (j==i?1: ((xi-xt[j])
                                        /(xt[i]-xt[j])) );
46                          multip *= yt[i];          //koeff von
```

```
                                legender polynom drangesetzt
47                          summ += multip;
48                  }
49                  apprLagTsch[k] = summ;
50          }
51
52          //lagrange interpolation -- aequidistant
53          double apprLagAeq[N];
54          for (int k = 0; k <= N; k++)     //schleife ueber
                ausgaberaster
55          {
56                  double xi = a + delx * k;
57                  double summ = 0;          //fuer summe
58                  for (int i = 0; i <= n; i++)
59                  {
60                          double multip = 1;        //fuer produkt
61                          for (int j = 0; j <= n; j++)
62                                  multip *= (j==i?1: ((xi-xa[j])
                                        /(xa[i]-xa[j])) );
63                          multip *= ya[i];          //koeff von
                                legender polynom drangesetzt
64                          summ += multip;
65                  }
66                  apprLagAeq[k] = summ;
67          }
68
69          //Ausgabe
70          for (int k = 0; k <= N; k++ )
71          {
72                  double xi = a + delx * k;
73                  std::cout << xi << "\t" << f(xi) << "\t" <<
                        apprLagTsch[k] << "\t" << apprLagAeq[k] <<
                         "\n";
74          }
75  }
```

2.3 Newton-Interpolation

Die Newton-Interpolation setzt auf eine andere Darstellung des – eindeutigen – Polynoms P_n:

$$P_n(x) = \sum_{i=0} c_i \prod_{j=0}^{i-1} (x - x_j) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \dots . \quad (2.7)$$

Diese Darstellung hat den Vorteil, dass für $x = x_k$ alle Summanden ab Ordnung k wegfallen: $P_n(x_0) = c_0$ oder $P_n(x_1) = c_0 + c + 1(x_1 - x_0)$. Auf diese Weise kann man sukzessive mit Bedingunge (2.1) die Koeffizienten ausrechnen.

Es zeigt sich dabei, dass man diese Koeffizienten elegant ausrechnen kann; die Methode hierzu heißt „Schema der dividierten Differenzen“. Sie beruht auf dem Zusammenhang

$$c_i =: [x_0, ..., x_i] , \quad (2.8)$$

wobei der Term $[x_0, ..., x_i]$ rekursiv definiert ist via

$$[x_i, ..., x_j] = \frac{[x_i, ..., x_{j-1}] - [x_{i+1}, ..., x_j]}{x_i - x_j} \quad \text{und} \quad [x_i] = y_i \; . \tag{2.9}$$

So kann man aus den Stützstellen die Differenzen 0. Ordnung ($[x_i]$) erhalten und aus zwei davon die nächster Ordnung. usw.

Auch die *Auswertung* der so entstehenden Polynome ist relativ einfach machbar: Über das *Horner-Schema*; möchte man den Funktionswert $P_n(\xi)$ wissen und kennt alle Koeffizienten c_i, so kann man folgenden Algorithmus verwenden:

Listing 2.2: Horner-Schema

```
1 double p = c[n];
2 for (int k = n-1; k >= 0; k--)
3       p = c[k] + (xi - x[k]) * p;
```

Das gesuchte Ergebnis ist dann in p, die Stützstellen x_i sind hier x[i].

2.4 Hermite-Interpolation

Die Hermite-Interpolation ist eine Erweiterung der Newton-Interpolation: Sind zusätzlich zu den Werten der Stützstelle noch die *Werte der Ableitung* an den Stellen gegeben – also neben (x_i, y_i) noch (x_i, y_i') – so kann man das obige Schema erweitern: Das approximierende Polynom hat dann die Gestalt

$$P_{2n+1}(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)^2 + c_3(x - x_0)^2(x - x_1) + ... \; , \tag{2.10}$$

wobei das Schema zur Bestimmung der Koeffizienten sich so ändet, dass sich vor der ersten Ordnung noch eien Reihe von Dividierten Differenzen ergeben: $[x_i, x_i]$. Diese sind nach Definition (2.9) und ein wenig Limesvertauschung so zu verstehen:

$$[x_i, x_i] = \lim_{h \to 0} \frac{[x_i + h] - [x_i]}{x_i + h - x_i} = \lim_{h \to 0} \frac{f(x_i + h) - f(x_i)}{h} = f'(x_i) =: y_i' \; . \tag{2.11}$$

Die Koeffizienten c_i sind dann gegeben durch

$$c_0 = y_0, c_1 = [x_0, x_0] = y_0', c_2 = [x_0, x_0, x_1], c_3 = [x_0, x_0, x_1, x_1], ... \; . \tag{2.12}$$

Dieses Verfahren macht aber nur dann Sinn, wenn neben den Funktionswerten auch die Ableitungswerte gegeben sind...

2.5 Numerisches Differenzieren

Um eine Funktion zu differenzieren, nähern wir sie durch ein Polynom an und differenzieren dieses. Das Verfahren, wie man auf dieses Polynom kommt, ist nicht entscheidend, da das entstehende Polynom ja eindeutig ist – also bekommt man über andere Verfahren nur andere Darstellungen des selben Polynoms. Wir verwenden hier die Lagrange-Interpolation (vgl. Kap 2.2) und außerdem äquidistante Stützstellen – das vereinfacht das Rechnen.

Um eine n-te Ableitung zu bestimmen, kann man die Funktion durch ein Polynom n-ten Grades approximieren. Wenn dieses n-mal abgeleitet wird, so hat man eine Konstante – gesuchte Ableitung eben.

Verwendet man die Vereinfachung $\lambda_i = \left[\prod_{j=0;j\neq i}^{n}(x_i - x_j)\right]^{-1}$, so gilt für die n-te Ableitung des Lagrange-Polynoms P_n:

$$\frac{\mathrm{d}^n}{\mathrm{d}\,x^n}P_n(x) = \sum_{i=0}^{n} y_i\lambda_i\, n! \approx f^{(n)}(x)\ . \tag{2.13}$$

Verwendet man die Äquidistanz der Stützstellen – alle haben den Abstand h voneinander –, kann man den Ausdruck λ_i noch vereinfachen: $\lambda_i = \binom{n}{i} \cdot \left[(-1)^{n-i}\, h^n\, n!\right]^{-1}$. Für die Ableitung kommt man dann auf

$$\frac{\mathrm{d}^n}{\mathrm{d}\,x^n}P_n(x) = \frac{1}{h^n}\sum_{i=0}^{n}\binom{n}{i}(-1)^{n-i}y_i\ . \tag{2.14}$$

Für den Spezialfall $n = 1$ hat man damit die einfachste Numerische Ableitung:

$$f'(x) \approx \frac{1}{h}(y_1 - y_0) = \frac{f(x+h) - f(x)}{h}\ ,$$

für $n = 2$

$$f''(x) \approx \frac{1}{h^2}(y_2 - 2y_1 + y_0) = \frac{\frac{y_0-y_1}{h} - \frac{y_1-y_2}{h}}{h}$$

usw.

Dies Art der Approximation ist aber nicht besonders genau – es ist sogar die schlechteste, die gerade noch so den Job tun kann.

Bessere Approximationen bekommt man, wenn man für die Bildung einer k-ten Ableitung ein Polynom der Ordnung $n > k$ verwendet. Ein entscheidender Unterschied zum obigen Fall ist der, dass hier die Ableitung keine konstante Funktion mehr ergibt – man muss also darauf achten, an welcher Stelle man das abgeleitete Polynom *auswertet*.

Nach der Bildungsvorschrift aus Kap. 2.2 können wir bspw. das Polynom P_2 aufschreiben – wobei wir wieder die Äquidistanz ausnutzen –:[1]

$$P_2(x) = \frac{y_0(x-x_1)(x-x_2)}{2h^2} - \frac{y_1(x-x_0)(x-x_2)}{h^2} + \frac{y_2(x-x_0)(x-x_1)}{2h^2}\ .$$

Leitet man diesen Term an der Stelle x_1 ab, erhält man

$$P_n'(x_1) = \frac{(x_1-x_0)\ y_2 + (2\,x_2 - 4\,x_1 + 2\,x_0)\ y_1 + (x_1-x_2)\ y_0}{2\,h^2} = \frac{+y_2 - y_0}{2h}\ ,$$

was schon eine bessere Approximation ist. Analog kann man noch mit einem Polynom dritter Ordnung approximieren und ableiten und erhält immer bessere Ableitungen. Vergleiche dazu Abb. 2.2.

Auf die gleiche Art und weise kann man natürlich auch höhere Ableitungen berechnen. Hier noch Approximation in 3. Ordnung:

[1] Bemerkung: Das „−“ vor dem zweiten Summanden kommt daher, dass der Nenner eigentlich negativ ist.

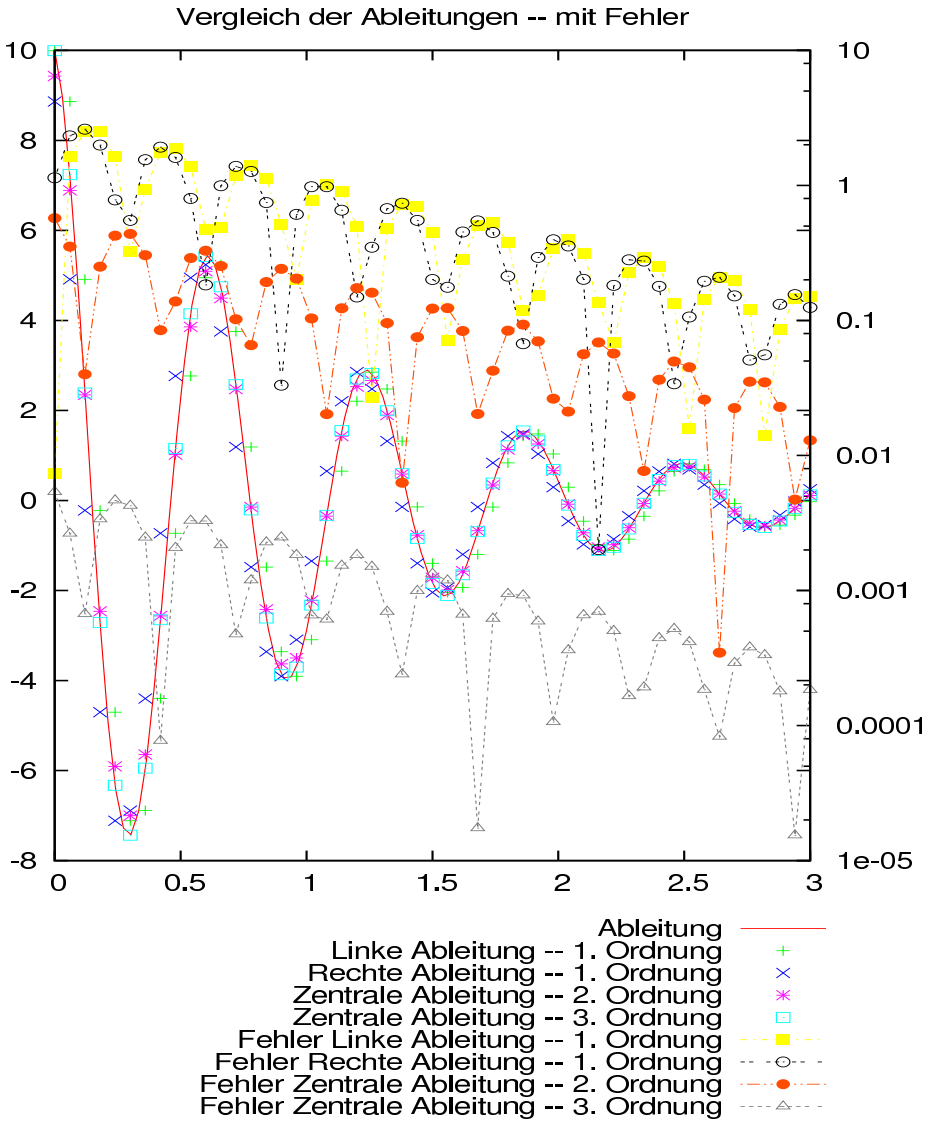


Abbildung 2.2: Vergleich verschiedener (erster) Ableitungen mit Fehler

$$f'(\bar{x}) \approx \frac{1}{24h}(y_0 - 27y_1 + 27y_2 - y_3) \text{ wobei } \bar{x} = \frac{1}{2}(x_0 + x_3) \ ,$$
$$f''(\bar{x}) \approx \frac{1}{2h^2}(y_0 - y_1 - y_2 + y_3) \ .$$

2.6 Extrapolation

Um *noch* genauere Ergebnisse für die Ableitungen zu erhalten, kann man wie folgt vorgehen: Mit der Schrittweite h berechnet man einen Näherungswert η^h der gesuchten Größe. Dann variiert man das h – man bildet eine Folge $(h_j)_j$, sodass $h_{j+1} < h_j$, mit ein paar Gliedern und interpoliert dann die Messpunkte (h_j, η^{h_j}) mit einem der oben genannten Verfahren. In der Interpolation untersucht man dann $h = 0$.

2.7 Gauß-Approximation

Die Gauß Approximation unterscheidet sich im Grundsatz nicht besonders von den bisher angesprochenen Methoden; Eine Funktion f soll durch g so approximiert werden, dass $\|f - g\|$ minimal ist. Wenn $\|\cdot\|$ minimal sein soll, ist das gleichbedeutend damit, dass $\|\cdot\|^2 = \langle \cdot \,|\, \cdot \rangle$ minimal sein soll.

In dem zur Approximation zur Verfügung stehenden Raum U seien die linear unabhängingen Funktionen $\{\varphi_i\}_i$ gegeben – es ist also automatisch $U = \text{span}(\varphi_0, \varphi_1, ..., \varphi_n)$ – , dann ist

$$g = \alpha^i \varphi_i \ .$$

Die beste Approximation von f in U – wobei f natürlich nicht unbedingt in U liegen muss – ist erreicht, wenn $\langle \varphi_i \,|\, f - g \rangle = 0 \ \forall i$, denn dann verschwindet die Projektion von $f - g$ nach U. Mit der Darstellung von g kann man dies umschreiben:

$$\langle \varphi_i \,|\, f \rangle - \langle \varphi_i \,|\, g \rangle = 0 \Leftrightarrow \langle \varphi_i \,|\, f \rangle = \langle \varphi_i \,|\, \varphi_j \rangle \alpha^j \ . \tag{2.15}$$

Und das wiederum ist ein Matrix-Gleichungssystem:

$$\boldsymbol{f} = \mathbf{A} \cdot \boldsymbol{\alpha} \ ,$$

wobei $\det \mathbf{A} \neq 0$ ist, weil die φ_i linear unabhängig sind. Damit existiert $\mathbf{A}^{-1}$ und damit eine eindeutige Lösung für die α^i, nämlich

$$\boldsymbol{\alpha} = \mathbf{A}^{-1} \cdot \boldsymbol{f} \ .$$

2.7.1 Diskrete Gauß-Approximation

Hat man eine Datenmenge (x_i, f_i) vorliegen, so ist das Skalarprodukt der Wahl

$$\langle u \,|\, v \rangle = \sum_i u(x_i)\, v(x_i)\, \omega(x_i) \ , \tag{2.16}$$

wobei ω eine Gewichtungsfunktion ist.

2.7.2 Kontinuierliche Gauß-Approximation

Der Raum U, in dem approximiert werden soll, ist jetzt der Raum der quadratintegrablen Funktionen $L_2([a,b])$ und das Skalarprodukt ist

$$\langle u \,|\, v \rangle = \int_a^b u(x)\, v(x)\, \omega(x)\, \mathrm{d}\, x \tag{2.17}$$

und ω ist wieder eine (integrierbare) Gewichtungsfunktion.

Um das zu lösende Gleichungssystem (2.15) besser lösen zu können, verwendet man als Basis in $U = L_2$ ein System *orthogonaler* Funktionen (also $\langle \varphi_i \,|\, \varphi_j \rangle \propto \delta_{ij}$) – damit hat die Matrix $\mathbf{A}$ nur noch Einträge auf ihrer Diagonalen und es gilt einfach

$$\alpha^i = \frac{\langle \varphi_i \,|\, f \rangle}{\langle \varphi_i \,|\, \varphi_i \rangle} \,. \tag{2.18}$$

Trigonometrische Approximation

Die Funktionen

$$\sin(ix) \text{ und } \cos(ix) \text{ mit } i \in \{0, ..., n\} \tag{2.19}$$

sind auf $[-\pi, \pi]$ orthogonal. Enwickelt (bzw. approximiert) man f mit diesen kommt das eine Fourierreihenentwicklung gleich.

Diese lässt sich mit der *fast fourier transformation* ökonomisch berechnen.

Tschebuscheff-Polynome

Grundlage für diese Orthogonalen Polynome sind die Additionstheoreme des Cosinus; mit diesen ist $\cos(n\varphi)$ als Polynom vom Grad n in $\cos\varphi$ darstellbar.

Entsprechend zu 2.7.2 sind die Tschebyscheff-Polynome:

$$T_n(x) = T_n(\cos\varphi) = \cos(n\,\varphi) \quad \text{und } x \in [-1, 1] \,. \tag{2.20}$$

Um wirklich $T_n(x)$ zu erhalten, verwendet man[2]

$$\cos(n\varphi) = \cos^n\varphi - \binom{n}{2} \cos^{n-2}\varphi \sin^2\varphi + \binom{n}{4} \cos^{n-4}\varphi \sin^4\varphi - \binom{n}{6} \cos^{n-6}\varphi \sin^6\varphi + \dots \,; \tag{2.21}$$

mit der Substitution

$$\cos\varphi = x \text{ und } \sin\varphi = \sqrt{1 - x^2} \,. \tag{2.22}$$

Es gilt die Rekursionsformel

$$T_{n_1} = 2x\, T_n - T_{n-1} \text{ für } n \geq 1 \text{ und } T_0(x) = 1, T_1(x) = x \,,$$

sowie die Beziehungen

$$\begin{aligned} T_n(-x) &= (-1)^n T_n(x) \\ |T_n(x)| &\leq 1 \\ T_n(x) &= 2^{n-1}x^n + \dots \text{ (führender Koeff: } 2^{n-1} \text{)}. \end{aligned}$$

[2] Vgl Bronstein

Für das Skalarprodukt[3]

$$\langle u \,|\, v \rangle = \int_{-1}^{1} u(x)\, v(x)\, \frac{1}{\sqrt{1-x^2}}\, \mathrm{d}\, x$$

sind die Polynome *orthogonal*:[4]

$$\langle T_k \,|\, T_j \rangle = \begin{cases} \frac{\pi}{2}\delta_{kj} & k = j \neq 0 \\ \pi & k = j = 0 \end{cases} .$$

Die Approximierende Funktion g – ein Polynom der Ordnung n – stellt man mit den Polynomen T_k dar via

$$g_n = \frac{c_0}{2} T_0 + \sum_{k=1}^{n} c_k T_k \text{ mit } c_k = \frac{2}{\pi} \int_{-1}^{1} f(x) T_k(x) \frac{1}{\sqrt{1-x^2}}\, \mathrm{d}\, x \,. \tag{2.23}$$

Mit Symmetrien[5] und der Fouriertransformation[6] erhält man als Näherung

$$c_k \approx \frac{2}{n} \left(\frac{1}{2} \left(f(1) + f(-1)\cos(\pi k) \right) + \sum_{j=1}^{n-1} f(\cos\left(\frac{j}{m}\pi\right)) \cdot \cos(\frac{j}{m} k\pi) \right) \,, \tag{2.24}$$

wobei man den Fehler abschätzen kann durch

$$\| f - g_n \| \leq_{k=n+1}^{\infty} |c_k| \,.$$

Mit dem *Algorithmus von* Clenshaw kann man die Auswertung von g_n an einer Stelle x effizient programmieren:

```
1 d[n] = c[n];
2 y = 2 * x;
3 d[n-1] = c[n-1] + y * c[n];
4 for(int k = n-2; k >= 0; k--)
5 {
6       d[k] = c[k] + y * d[k+1]-d[k+2];
7 }
8 gn = (d[0] - d[2]) / 2.0;
```

Möchte man eine Menge von n Punkten (x_i, y_i) mit den Tschebyscheff-Polynomen approximieren, sodass also $g_n(x_i) = y_i$ gilt, dann gilt für das g_n aus (2.23) mit dem „Skalarprodukt“

$$\sum_{l=1}^{n+1} T_k(x_l) T_j(x_l) = \begin{cases} 0 & k \neq j \\ (n+1)/2 & k = j \neq 0 \\ n+1 & k = j = 0 \end{cases} ;$$

[3] Der Term $\omega(x) = 1/\sqrt{1-x^2}$ ist formal eine Gewichtungsfunktion.

[4] Das sieht man leicht, weil $1/sqrt1 - x^2 = 1/\sin x$ ist und $\mathrm{d}(\cos x) = -\sin x\, \mathrm{d}\, x$, womit man wieder die Trigonometrischen Funktionen von vorher hat.

[5] bspw des Cosinus

[6] die hier nicht besprochen wird

dafür müssen aber x_l die Nullstellen von T_{n+1} sein:

$$x_l = \cos(\frac{(2l-1)}{(n+1)}\,\frac{\pi}{2}) \;,$$

dann erhält man für die Koeffizienten

$$c_k = \frac{2}{n+1}\sum_{l=1}^{n+1} f(x_l)\,\cos(k\,x_l) \;. \tag{2.25}$$

Legender-Polynome

Die Legender-Polynome sind definiert als

$$P_n(x) = \frac{1}{2^n\,n!}\,\frac{\mathrm{d}^n}{\mathrm{d}\,x^n}\left((x^2-1)^n\right) \;, \tag{2.26}$$

und sind *orthogonal* unter dem Skalarprodukt[7]

$$\langle u \,|\, v\rangle = \int_{-1}^{1} u(x)\,v(x)\,\mathrm{d}\,x \;; \tag{2.27}$$

mit

$$\langle P_k \,|\, P_l\rangle = \frac{2}{2n+1}\,\delta_{kl} \;. \tag{2.28}$$

Das Legendrepolynom P_n hat die Ordnung n und im Intervall $[-1,1]$ n *einfache Nullstellen*, und es gilt die Rekursion

$$P_{n+1} = \frac{2n+1}{n+1}\,x\,P_n - \frac{n}{n+1}\,P_n \text{ für } n \in \{1,2,...\} \text{ und } P_o(x) = 1 \text{ und } P_1(x) = x \;. \tag{2.29}$$

Auf für sie gilt die Eigenschaft $|P_n(x)| \leq 1$ für $x \in [-1,1]$.

Das Nährungspolynom g_n der Ordnung n stellt man dar mit

$$g_n = \sum_{k=0}^{n} c_k P_k \text{ mit } c_k = \frac{1k+1}{2}\int_{-1}^{1} f(x)\,P_k(x)\,\mathrm{d}\,x \;. \tag{2.30}$$

Hier muss man jedoch die Koeffizienten ausrechnen – mit Methoden wie sie weiter unten besprochen werden: Näherungsweise mit der *Gauß-Quadratur*.

[7] Hier ist – formal – die oben angesprochene Gewichtungsfunktion $\omega(x) \equiv 1$.

Kapitel 3

Numerisches Integrieren

Allgemein versucht man, das bestimmte Integral $\int_a^b f(x)\,\mathrm{d}\,x$ durch eine Summe anzunähern:

$$\int_a^b f(x)\,\mathrm{d}\,x \approx \sum_{i=0}^{n} w_i f(x_i)\ . \tag{3.1}$$

Die w_i heißen (Integrations)*Gewichte* und die x_i heißen *Stützstellen*. Die Wahl von w_i und x_i werden durch die jeweilige Integrationsregel festgelegt.

Man Spricht von einem **Genauigkeitsgrad** m, wenn (3.1) eine *Gleich*ung ist mit einem Polynom f mit grad $f \leq m$: Ein Polynom bis zum Grad m wird exakt approximiert.

3.1 Newton-Cotez

Mit $n+1$ Stützstellen $x_0, ..., x_n$ wir die Funktion f mit Lagrange Interpoliert (vgl Kap. 2.2) und das Polynom integriert:

$$\int_a^b f(x)\,\mathrm{d}\,x \approx \int_a^b \sum_{i=0}^{n} f(x_i) L_i(x)\,\mathrm{d}\,x = \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b L_i(x)\,\mathrm{d}\,x}_{w_i}\ . \tag{3.2}$$

Die Stützstellen kann man der Einfachheit halber äquidistant wählen.

Für $n = 1$ erhält man so

$$P_1(x) = f(x_0)\frac{x - x_1}{x_0 - x_1} + f(x_1)\frac{x - x_0}{x_1 - x_0} = f(x_0)\frac{x - x_1}{-h} + f(x_1)\frac{x - x_0}{h}$$

und Integration mit den Grenzen $a = x_0$ und $b = x_1$ liefert die **Trapezregel**

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}\,x \approx (x_1 - x_0)\,\frac{y_0 + h_1}{2}\ . \tag{3.3}$$

Analog kann man mit $n = 3$ vorgehen; setzt man als Stützstellen $x_0 = a$, $x_1 = \frac{a+b}{2}$ und $x_2 = b$ so folgt die **Simpsonregel**

$$\int_{x_0}^{x_2} f(x)\,\mathrm{d}\,x \approx (x_2 - x_0)\frac{f(x_0) + 4f(x_1) + f(x_2)}{6}\ . \tag{3.4}$$

Diese Algorithmen sind jedoch nur für kleine n stabil. Um eine höhere Genauigkeit für das approximierte Integral zu bekommen, macht es eshalb weniger Sinn, die Funktion f durch höhere Polynome zu approximieren, sondern vielmehr teilt man den Integrationsbereich $[a, b]$ in viele kleien Unterbereiche auf, die man dann jeweils wieder mit Trapez- oder Simpsonregel integriert.

Für die Trapezregel ergibt sich so beispielsweise, wenn man $[a, b]$ in n Teilintervalle aufteilt:

$$\int_a^b f(x)\,\mathrm{d}\,x \approx h\left(\frac{y_0}{2}\sum_{i=1}^{n-1}+\frac{y_n}{2}\right)\,. \tag{3.5}$$

Die Methode eignet sich besonders für periodische Funktionen mit der Periode $[a, b]$.

Erstes adaptives Integrieren Nun wollen wir ein Verfahren betrachten, wie die Integartionsgenauigkeit gesteigert werden kann: Der Komputer soll in der Lage sein, zu entscheiden, wie weit er das Intervall $[a, b]$ aufteilen soll.

Dazu machen wir den – etwas naiven – Ansatz zur Integration, der sich aus der Riemann'schen Theorie ergibt: Die **Mittelpunktsregel**: Das Intervall $[a, b]$ wird in n Teilintervalle der Breite h aufgeteilt und in der Mitte des Teilintervalls wird der Funktionswert verwendet:

$$\int_a^b f(x)\,\mathrm{d}\,x \approx \sum_{i=0}^{n-1} h\,y_{i+\frac{1}{2}} \tag{3.6}$$

Bezeichnet man nun mit $T(h)$ die Trapezregel mit Schrittweite h und mit $M(h)$ die Mittelpunktsregel mit Schrittweite h, so sieht man anhand (3.5) und (3.6), dass

$$\frac{M(h)+T(h)}{2} = T(\frac{h}{2})\,, \tag{3.7}$$

einfach weil die Randterme mit $y_0/2$ und $y_n/2$ vorkommen und jetzt zwischen dei einzelnen Summanden von $T(h)$ die Summanden von $M(h)$ „dazwischen geschoben werden". Damit man wieder eine Form wie (3.6) bekommt, muss man noch die Schrittweite anpassen, damit zwischen a und b weiterhin n Schritte passen: Die Schrittweite wird halbiert.

Die Gl. (3.7) kann man als Rekursionsbeziehung für die Verfeinerung der Trapezregel ansehen: Man berechnet $T(h)$ und $M(h)$ und wenn $|T(h)-M(h)| \geq \varepsilon$ ist (mit einer vorher fest gewählten Toleranz ε), dann bestimmt man mit den beiden Integralen über (3.7) $T(h/2)$ und anschließend $M(h/2)$. Um ökonomisch vorzugehen kann man dabei verwenden, dass man bei $M(h/2)$ nur noch die Hälfte der Approximation neu berechnen muss: Jedes zweite Teilintervall ist bereits in $M(h)/2$ eingegangen. So benötigt man für die Berechnung von $M(h/2)$ nur noch n Rechenschritte. Dies ist die Idee, der adaptiven Integration; vgl Kap. 3.4.

Die Idee dabei ist, dass wen zwei Approximationsmethoden beinahe das selbe Ergebnis liefern, dass dann das *tatsächliche* Ergebnis nicht mehr zu stark abweichen darf/kann/sollte.

Bei abklingenden Funktionen kann man in den Algorithmus einbauen, dass weitere Funktionswerte ab $f(x) \leq \delta$ (mit einem vorher gewählten δ) verworfen werden.

3.2 Rombert-Integration

Eine weitere Möglichkeit, die – relativ einfache – Trapezformel zu verbessern ist ein Ansatz, in dem man die Fehlerentwicklung der Approximationen für verschiedene Schrittweiten h so kombiniert, dass in der entstehenden Entwicklung niedrige Ordnungen wegfallen. Im Klartext: Kluge Menschen haben eine Formel entwickelt, nach der man Trapezregeln mit verschiedenen Schrittweiten so miteinander verrechnen kann, dass das Ergebnis noch genauer ist, weil sich bestimmte Fehler gegenseitig wegheben.

Dazu definiert man für die einfache Trapezregel

$$T_{k,0} = T(h_k) \text{ wobei } h_k = \frac{b-a}{2^k} \text{ und } k \in \{0, ..., n\} \,, \tag{3.8}$$

und weiter die Rekursion

$$T_{k,j} = \frac{4^j\, T_{k,j-1} - T_{k-1,j-1}}{4^j - 1} \text{ mit } j \in \{1, 2, ..., n\} \text{ und } k \in \{j, j+1, ..., n\} \,. \tag{3.9}$$

Diese definiert eine Dreiecksstruktur; bspw für $n = 3$:

$$\begin{array}{ccccccc} & & T_{33} & & & & \text{Exaktestes Ergebnis} \\ & T_{32} & & T_{22} & & & \\ T_{31} & & T_{21} & & T_{11} & & \\ T_{30} & & T_{20} & & T_{10} & T_{00} & \text{Trapezregeln} \end{array}$$

Mit den vier Trapezregeln in der untersten Zeile kann man die darüber liegenden $T_{k,l}$-Werte mit geringem Aufwand berechnen und so eine wesentlich genauere Approximation erhalten.

Wieder adaptiv integrieren Möchte man hier wieder den Komputer entscheiden lassen, wie viele Iterationsschritte möglich sind, so kann man sich an den Rechten Rand der Dreiecksstruktur wenden: Man bestimmt $T_{0,0}$ und anschließend $T_{1,1}$. Ist die Differenz der beiden Werte kleiner als eine Toleranz ε, wird kein weiterer Wert berechnet, wenn nicht, wird $T_{2,2}$ berechnet usw.

3.3 Gauß-Integration

Die Gauß-Integration greift auf die orthogonalen Legender-Polynome zurück – vgl. Kap. 2.7.2. Da diese nur auf dem Intervall $[-1, 1]$ ihre Wunscheigenschaften haben (also bspw. die Orthogonalität), kann man für diese Integration auch nur das Intervall $[-1, 1]$ verwenden. Durch die Lineare Transformation

$$\Psi : [a, b] \to [-1, 1];\ x \mapsto t = \frac{b-a}{2}x + \frac{a+b}{2} \tag{3.10}$$

kann man dem jedoch abhelfen:

$$\int_a^b f(x)\,\mathrm{d}\,x = \frac{b-a}{2} \int_{-1}^{1} f(\frac{b-a}{2}x + \frac{a+b}{2})\,\mathrm{d}\,x \tag{3.11}$$

Die Idee bei der Gauß-Integration ist, dass sowohl die Stützstellen x_i als auch die Integrationsgewicht w_i so gewählt werden sollen, dass der *Genauigkeitsgrad*

n	x_i	w_i
1	0	2
2	$\pm\sqrt{\frac{1}{3}}$	1
3	$0; \pm\sqrt{\frac{3}{5}}$	$\frac{8}{9}; \frac{5}{9}$
4	$\pm\sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}};\ \pm\sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$	$\frac{18+\sqrt{30}}{36};\ \frac{18-\sqrt{30}}{36}$
5	$0;\ \pm\frac{1}{3}\sqrt{5 - 2\sqrt{\frac{10}{7}}};\ \pm\frac{1}{3}\sqrt{5 + 2\sqrt{\frac{10}{7}}}$	$\frac{128}{225};\ \frac{322+13\sqrt{70}}{900};\ \frac{322-13\sqrt{70}}{900}$

Tabelle 3.1: Stützstellen und Integrationsgewichte der ersten Gauß-Integrationen

der Integration möglichs hoch ist – also dass ein Polynom von möglichst hohem Grade noch exakt berechnet werden kann.

Bei der Gauß-Integration lässt sich dabei mit n Stützstellen noche in Polynom vom Grad $2n-1$ exakt berechnen[1], bei Newton-Cotez ist geht es nur bis zu Polynomen der Ordnung $n-1$. Den „Preis“, den man dafür zahlen muss, ist, dass Stützstellen und Gewichte für verschiedene n jeweils neu berechnet werden müssen.

Die Stützstellen x_i sind die *Nullstellen* des n-ten Legendrepolynoms und die Gewichte kann man durch Hardcore-Lineare-Algebra[2] erhalten als

$$w_k = 2/\left(1 + 3P_1^2(x_k) + 5P_2^2(x_k) + 7P_3^2(x_k) + ...[2(n-1)+1]P_{n-1}^2(x_k)\right) \tag{3.12}$$

Die ersten Werte sind hier tabelliert in 3.1.

Das Verfahren eigenet sich besonders für *glatte* Funktionen.

In Listing 3.1 ist ein Beispielprogramm – vor allem, wie es mit den Substitutionen funktioniert. In Abb. 3.1 sind die Ergebnisse verschiedener Integrationsmethoden grapisch dargestellt.

Listing 3.1: Beispiel für eine Gauss-Quadratur

```
1 #include<iostream>
2 #include<cmath>          //fuer f(x)
3 #include<cstdlib>        //fuer satoi
4 #include<iomanip>        //fuer nachkommastellen
5
6 //// global variables ////
7 int N;
8 double dx;
9
10 double f(double x)
11 {
12         return std::sin(10*x)*std::exp(-x);
13 }
```

[1] Dass dies das *Maximum* ist, kann man zeigen, indem man ein Polynom P_n vom Grade $2n$ konstruiert mit $P_n(x) = \prod_{i=1}^{n}(x - x_i)^2$: Es ist $P(x) \geq 0$ und damit auch sein Integral, für jede Stützstelle x_i ist jedoch $P(x_i) = 0$ und damit verschwindet das approximierte Integral: Hier macht man also einen Fehler.

[2] Vgl. Schwarz: Numerische Mathematik, Kap. 7.4 + Nachhrechnen

```
14
15 double gauss3(double xi)
16 //flaeche unter kurvenabschnitt der Breite dx um xi mit gauss,
       3 stuetzstellen
17 {
18         const int ordnung = 3;
19         //integrationsintervall ist [xi-dx/2 ; xi+dx/2];
               substitution:
20         //       t = (x-xi)/(dx/2)
21         //damit
22         //       \int_{xi-dx/2}^{xi+dx/2} f(x) \diff x    =
23         //       dx/2 \int_{-1}^{1} f(t) \diff t          =
24         //       dx/2  \sum_{i=0}^{2} w_i f(t_i)
25         // Stuetzstellen: 0, +- sqrt{3/5}
26         //nur fuer die transformierte Version bekannt
27         double x[ordnung] = {-sqrt(3.0/5.0), 0.0, sqrt
               (3.0/5.0) };
28         //transformation
29         double t[ordnung];
30         for( int i = 0; i < ordnung; i++ )
31                 t[i] = xi + dx/2 * x[i];
32         // gewichte
33         double w[ordnung] = {5.0/9.0 , 8.0/9.0 , 5.0/9.0 };
34         // integration
35         double summe = 0;
36         for( int i = 0; i < ordnung; i++ )
37                 summe += f(t[i]) * w[i];
38         return dx/2 * summe;
39
40 }
41
42 int main(int argc, char* argv[])
43 {
44         N = 50;          //in wie viele Abschnitte $[a,b]$
               geteilt wird
45         dx = 3.0 / N;
46
47         //Gauss 3 Ordnung berechnen
48         double integral_gauss3 = 0;
49         for ( int i = 0; i < N; i++ )
50                 integral_gauss3 += gauss3(i*dx+dx/2.0);
51                 //das '+dx/2.0' wird benoetigt, weil die
                       Stuetzstelle in der
52                 //Mitte eines Intervalls liegen soll; fuer  i
                       =0  laege xi
53                 //sonst bei x=0 und das Trapez ginge von -dx/2
                       bis dx/2.
54
55         std::cout << "Wert des Integrals: ";
56         std::cout << std::setprecision(25);
57         std::cout << integral_gauss3 << "\n";
58 }
```

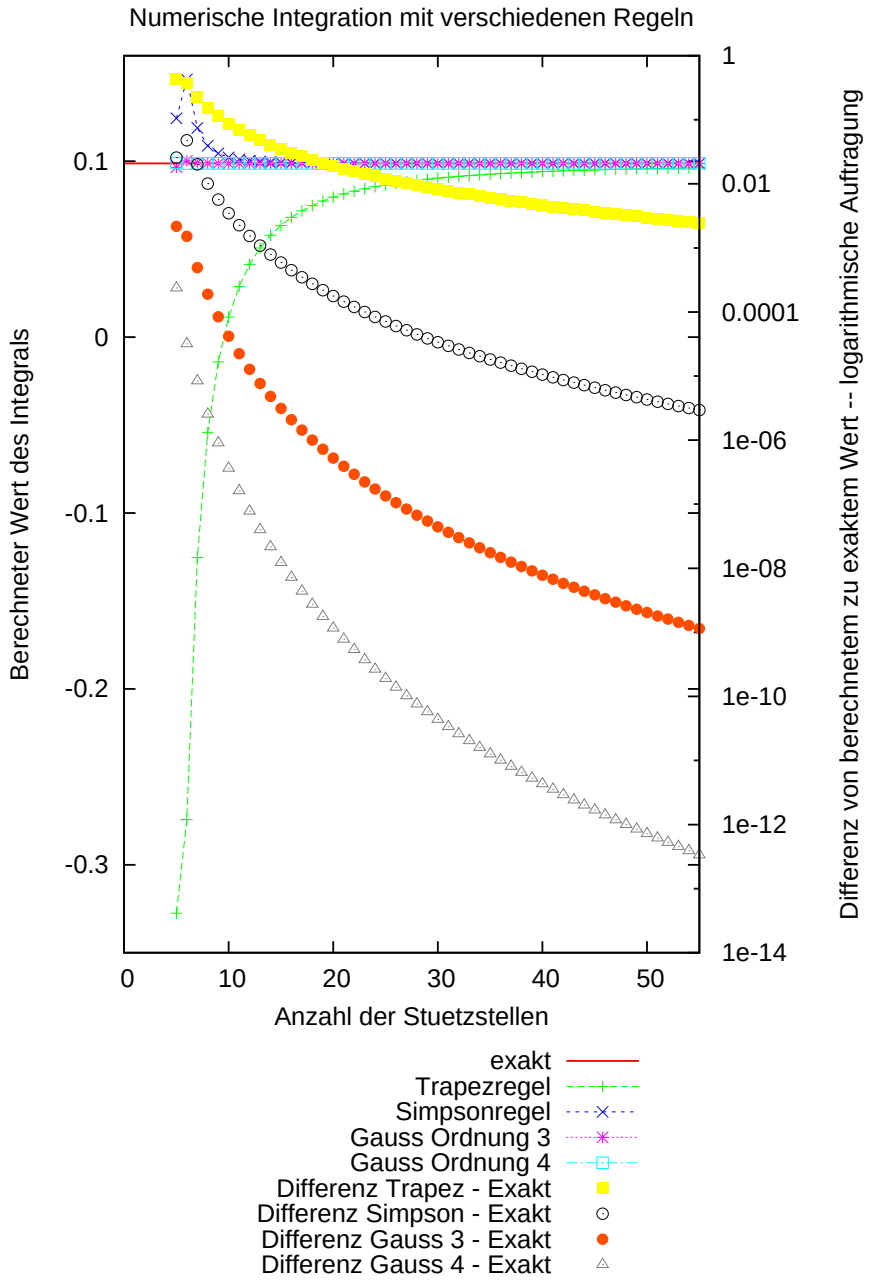


Abbildung 3.1: Vergleich verschiedener Integrationsmethoden

3.4 Adaptive Integration

Dein einfachste Ansatz ist wie oben schon besprochen, das Intervall $[a, b]$ zu halbieren, wo f stärker variiert – also schwerer zu integrieren ist.

Man kombiniert dabei zwei Verfahren, mit denen man jeweils das Teilintervall Integriert und vergleicht anschließend die Ergebnisse; sind diese um mehr als ε verschieden, wird das betreffende Intervall wieder halbiert usw. Elegant kann man dies natürlich in einer Rekursion behandeln.

Die Entscheidung, ob nun geteilt werden soll oder nicht, kann man noch verfeinern, indem die Abbruchbedingung

$$|I_1 - I_2| < \frac{b_j - a_j}{b - a}\,\varepsilon$$

verwendet – hier bei berücksichtigt man, dass die Teilintervalle kürzer sind und setzt den Integrationsfehler ins Verhältnis mit der Intervalllänge. Ebenfalls kann man einen Relativen Fehler δ einführen und die Abbruchbedingung

$$|I_1 - I_2| < \delta\, I_S \text{ oder } \frac{I_1}{I_2} < \delta$$

formulieren, wobei I_S ein grober Schätzwert für das Integral ist.

Kapitel 4

Einschrittverfahren für Anfangswertprobleme

Ganz allgemein behandeln wir Differenzialgleichungen der Form

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(t, \boldsymbol{y}(t)) \text{ mit } \boldsymbol{y}(a) = y(0) \text{ und } t \in [a, b] \,, \tag{4.1}$$

mit Funktionen $\boldsymbol{y} \in \mathbb{R}^N$. Um die Schreibarbeit zu erleichtern werden wir aber in Kurzform schreiben

$$y' = f(t, y) \text{ und } y(a) = y_0 \,. \tag{4.2}$$

Wichtig! 5 (Existenz und Eindeutigkeit) *Ist f* Lipschitz-stetig*, also*

$$\exists L \; \forall t \in [a, b] \; \forall u, v \in \mathbb{R}^N \;\; : \;\; \|f(t, u) - f(t, v)\| \leq L \cdot \|u - v\| \,, \tag{4.3}$$

dann existiert genau eine Lösung.

Auseinanderdriften Löst man die DGL für identsiches f aber zwei verschiedene Anfangswerte $y(a) = y_0$ und $\tilde{y}(a) = \tilde{y}_0$, dann driften die beiden Lösungen höchstens exponentiell auseinander:

$$\|y(t) - \tilde{y}(t)\| \leq \mathrm{e}^{L\,(t-a)} \, \|y_0 - \tilde{y}_0\| \,. \tag{4.4}$$

Differenzierbarkeit der Lösungen Außerdem kann man folgern, dass wenn f p-mal partiell differenzierbar ist, dass y dann $(p{+}1)$-mal partiell differenzierbar ist.

Herleitungen Die hier beschriebenen Herleitungen sind immer nur eine Möglichkeit; oft kann man auch durch (a) Raten oder (b) Taylorentwicklung (auch kombiniert mit (a)) auf die Resultate kommen: Viele Wege führen nach Rom...

4.1 Definition Einschrittverfahren, Verfahrensfehler

Allgemein verwenden wir ein diskretes Rechenverfahren; also anstatt für die Lösung der DGL (4.2) ein Funktion $y(t)$ zu erhalten, bekommen wir Datenpunkte $u_l := u(t_l)$ wobei u die numerisch genäherte Lösung zu y ist und die $\{t_l\}_l$ ein diskretes *Gitter*

$$\Delta := \{a = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = b\} \tag{4.5}$$

bilden mit den *Schrittweiten*

$$h_l = t_{l+1} - t_l \tag{4.6}$$

zwischen den einzelnen Zeitschritten. Im Allgemeinen werden diese h_l konstant gewählt.

Definition 4.1.1 (Einschrittverfahren) *Der Nährungswert u_{l+1} für y bei $t = t_{l+1}$ berechnet sich lediglich aus t_l, u_l und der Schrittweite h_l – hängt also explizit nur von dem vorhergehenden Schritt ab.*

Man definiert hierfür die *Verfahrensfunktion* φ sodass

$$u_{l+1} = u_l + h_l \cdot \varphi(t_l, u_l, h_l) \text{ mit } u_0 = y_0 \ . \tag{4.7}$$

Um die *Güte* eines Verfahrens bemessen zu können, führen wir die *Konvergenzordnung* ein:

Definition 4.1.2 (Konvergenzordnung $p \geq 1$) *Gilt für den globalen Verfahrensfehler die Abschätzung*

$$\underbrace{\max_l \|u_l - y(t_l)\|}_{\text{Glob. Verfahrensfehler}} \leq C \cdot \left(\max_l h_l\right)^p \ , \tag{4.8}$$

so ordnet man dem Verfahnre die Konvergenzordnung p zu.

Die Konvergenzordnung entspricht also der Ordnung des Restterms, wenn man das Taylorpolynom von Näherung u und Funktion y voneinander abzieht.

Es gibt nun ein wichtiges Resultat, um die Konvergenzordnung angeben zu können; dazu benötigen wir:

Definition 4.1.3 (Lokaler Verfahrensfehler $\eta(t,h)$) *Der Lokale Verfahrensfehler gibt die Abweichung an, die dem Verfahren an einer Stelle inhärent sind:*

$$\eta(t,h) := \underbrace{y(t) + h\,\varphi(t, y(t), h)}_{\approx y(t+h)} - y(t+h) \ . \tag{4.9}$$

Man berechnet also die Abweichung, zwischen einer von exakten Werten $y(t)$ ausgehenden Approximation mit der Verfahrensfunktion φ und dem tatsächlich exakten Wert $y(t+h)$; η ist gewissermaßen ein Restglied:

$$y_{l+1} = y_l + h_l \varphi(t_l, y_l, h_l) - \eta_l \ .$$

Diese Definition dient eigentlich nur für die Definition der Wichtigen

Definition 4.1.4 ((Konsistenz)Ordnung $p \geq 1$) *Gilt die Abschätzung*

$$\|\eta(t,h)\| \leq C\, h^{p+1} \tag{4.10}$$

dann spricht man dem Verfahren die Konsistenzordnung p zu.

Es gilt jetzt der Wichtige Zusammenhang

Wichtig! 6 *Erfüllt φ die Lipschitzbedingung ((4.3) mit der Konstanten L_φ) und hat die* Konsistenz*ordnung p so hat das Verfahren die* Konvergenz*ordnung p und der globale Fehler ist*

$$\max_l \|u_l - y(t_l)\| \leq \frac{C}{L_\varphi}\left(\mathrm{e}^{L\varphi(b-a)} - 1\right) \cdot \left(\max_l h_l\right)^p \ . \tag{4.11}$$

4.1.1 Einfache Verfahren

Konsistenzordnung $p = 1$: Euler

Das einfachste und anschaulichste – leider offensichtlich auch schlechteste – Verfahren um Anfangswerte zu lösen besteht darin, sich an jedem Punkt im Phasenraum um die Strecke h in Richtung der Ableitung zu bewegen:

$$u_{l+1} = u_l = h_l\, f(t_l, u_l) \text{ damit } \varphi(t_l, u_l) = f(t_l, u_l) \ . \tag{4.12}$$

Konsistenzordnung $p = 2$: Mod. Euler, Heun

Hier beginnt man mit dem Ansatz

$$\varphi(t,u,h) = a_1 f(t,u) + a_2 f(t + b_1 h, u + h_2 h f(t,u)) \tag{4.13}$$

und versucht, die Konstanten a_i und b_i optimal zu wählen. Dazu taylort man φ und y in h und vergleicht[1]

$$\begin{aligned} \varphi(t+h, y, h) = &\quad (a_1 + a_2) f(t,y) + h\left(\frac{\partial f}{\partial t} b_1 a_2 + \frac{\partial f}{\partial y} a_2 b_2\right) + O(h^2) \ , \\ y(t+h) = y(t) + h\,[&\quad f(t,y) + \frac{h}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^2)\ ] \ . \end{aligned}$$

Hier ergibt sich

$$a_1 + a_2 = 1 \ , \ b_1 a_2 = \frac{1}{2} \text{ und } a_2 b_2 = \frac{1}{2} \ ;$$

das lässt einen gewissen Spielraum in der Bestimmung der Koeffizienten.

Man erhält für $a_1 = 0, a_2 = 1$ und $b_1 = b_2 = \frac{1}{2}$ das *Modifizierte Euler-Verfahren*

$$\varphi(t,u,h) = f(t + \frac{h}{2}, u + \frac{h}{2} f(t,u)) \tag{4.14}$$

und für $a_1 = a_2 = \frac{1}{2}$ und $b_1 = b_2 = 1$ das *Verfahren von Heun*:

$$\varphi(t,u,h) = \frac{1}{2}\left(f(t,u) + f(t+h, u + h f(t,u))\right) \ . \tag{4.15}$$

[1] Beachte $h\, O(h^2) = O(h^3)$

Konsistenzordnung $p = 3$: Niedriges Runge Kutta

Konsistenzordnung $p = 4$: Klassisches Runge Kutta

Es ergibt sich (o.B.)

$$\varphi(t, u, h) = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4.16}$$

mit

$$\begin{aligned} k_1 &= f(t, u) \\ k_2 &= f(t + \frac{h}{2}, u + \frac{h}{2}k_1) \\ k_3 &= f(t + \frac{h}{2}, u + \frac{h}{2}k_2) \\ k_4 &= f(t + h, u + hk_3)\ . \end{aligned}$$

Bemerkung zu Runge-Kutta

Die RUnge-Kutta-Verfahren sind weit verbreitet, deswegen auch in vielen Standardbibliotheken implementiert. Sie haben eine hohe Genauigkeit (vgl. Abb. 4.2) und die Funktion f muss nicht differenzierbar sein – die Möglichen Probleme auf die man das Verfahren anwenden kann sind dementsprechend groß.

Ein bedeutender Nachteil ist jedoch, dass die Funktion f sehr häufig aufgerufen werden muss – im Falle $p = 4$ bspw. fünf mal, was sehr aufwändig wird.

In Listing 4.1 ist eine Implementierung des Verfahrens angegeben: Der harmonische Oszillator. Hier wurde auch die Energie mitausgegeben, damit man sich ein Bild machen kann, wie exakt die Verfahren sind. Der Output ist geplottet in Abb. 4.1.

Listing 4.1: Simulation Harmonischer Oszillator mit Euler und Runge-Kutta

```
1 #include<iostream>
2 #include<cmath>
3
4 const int dim = 2;
5 // Die Variablen y und v werden in einem Array der Dimension 2
      gespeichert.
6
7 const int N = 50000; //Schritte
8 const double a = 0; //Intervall
9 const double b = 100;
10 const double h = (b-a)/N; //Zeitschritt
11
12 const double anfangsbed[dim] = { 0.0 , 10.0 }; // { pos,
     geschw }
13
14 // BEISPIEL HARMONISCHER OSZILLATOR
15 const double om = 7; //frequenz -- masse: m==1
16 const double omq = om * om; //frequenzquadrat
17 void f(double t, double in[], double out[])
18 {
19         out[0] = in[1];
```

```
20          out[1] = - omq * in[0];
21  }
22
23  double energie_harmosz( double ortgeschw[] )
24  {
25          double kin = 0.5 * ortgeschw[1] * ortgeschw[1];
26          double pot = 0.5 * omq * ortgeschw[0] * ortgeschw[0];
27          return kin + pot;
28  }
29
30  int main()
31  {
32
33          //// EULER ////
34          double yeuler[N][dim];
35          for( int k=0; k< dim; k++)
36                  yeuler[0][k] = anfangsbed[k];
37
38          for( int i = 1; i <= N; i++ )
39          {
40                  double foo[dim];
41                  f( i*h , yeuler[i-1] , foo );
42                  for( int k=0; k< dim; k++ )
43                          yeuler[i][k] = yeuler[i-1][k] + h* foo
                                [k];
44          }
45
46          //// KLASS. RUNGE KUTTA /////
47          double yklrunge[N][dim];
48          for( int k=0; k< dim; k++)
49                  yklrunge[0][k] = anfangsbed[k];
50          for( int i = 1; i <= N; i++ )
51          {
52                  double t = i*h;
53                  double k1[dim];
54                  f( t , yklrunge[i-1] , k1 );
55
56                  double k2[dim];
57                  double k2hilfe[dim];
58                  for( int k=0; k< dim; k++)
59                          k2hilfe[k] = yklrunge[i-1][k]+ k1[k] *
                                h/2.0;
60                  f( t+h/2 , k2hilfe , k2 );
61
62                  double k3[dim];
63                  double k3hilfe[dim];
64                  for( int k=0; k< dim; k++)
65                          k3hilfe[k] = yklrunge[i-1][k] + k2[k]
                                * h/2.0;
66                  f( t+h/2 , k3hilfe , k3 );
67
68                  double k4[dim];
69                  double k4hilfe[dim];
70                  for( int k=0; k< dim; k++)
```

```
71                              k4hilfe[k] = yklrunge[i-1][k] + k3[k]*
                                    h;
72                      f( t+h    , k4hilfe , k4 );
73
74                      for( int k=0; k< dim; k++)
75                              yklrunge[i][k] = yklrunge[i-1][k] + (
                                    k1[k] + 2*k2[k] + 2*k3[k] + k4[k]
                                    ) * h/6;
76              }
77
78
79              //// ENERGIE ////
80              double energie_euler[N], energie_klrunge[N];
81              for( int i = 1; i <= N; i++ )
82              {
83                      energie_euler[i] = energie_harmosz( yeuler[i]
                            );
84                      energie_klrunge[i] = energie_harmosz( yklrunge
                            [i] );
85              }
86
87              ///// Ausgabe ////
88              std::cout << "#Zeit \t Euler \t Energie(Euler) \t Kl. 
                    Runge-Kutta  \t Energie(Kl Runge) \n";
89              for( int i = 1; i <= N; i++ )
90              {
91                      std::cout << i*h << "\t";
                                    //1
92                      std::cout << yeuler[i][0] << "\t";
                                    //2
93                      std::cout << energie_euler[i] << "\t";
                                    //3
94                      std::cout << yklrunge[i][0] << "\t";
                                    //4
95                      std::cout << energie_klrunge[i] << "\t";
                                    //5
96                      std::cout << "\n";
97              }
98      }
```

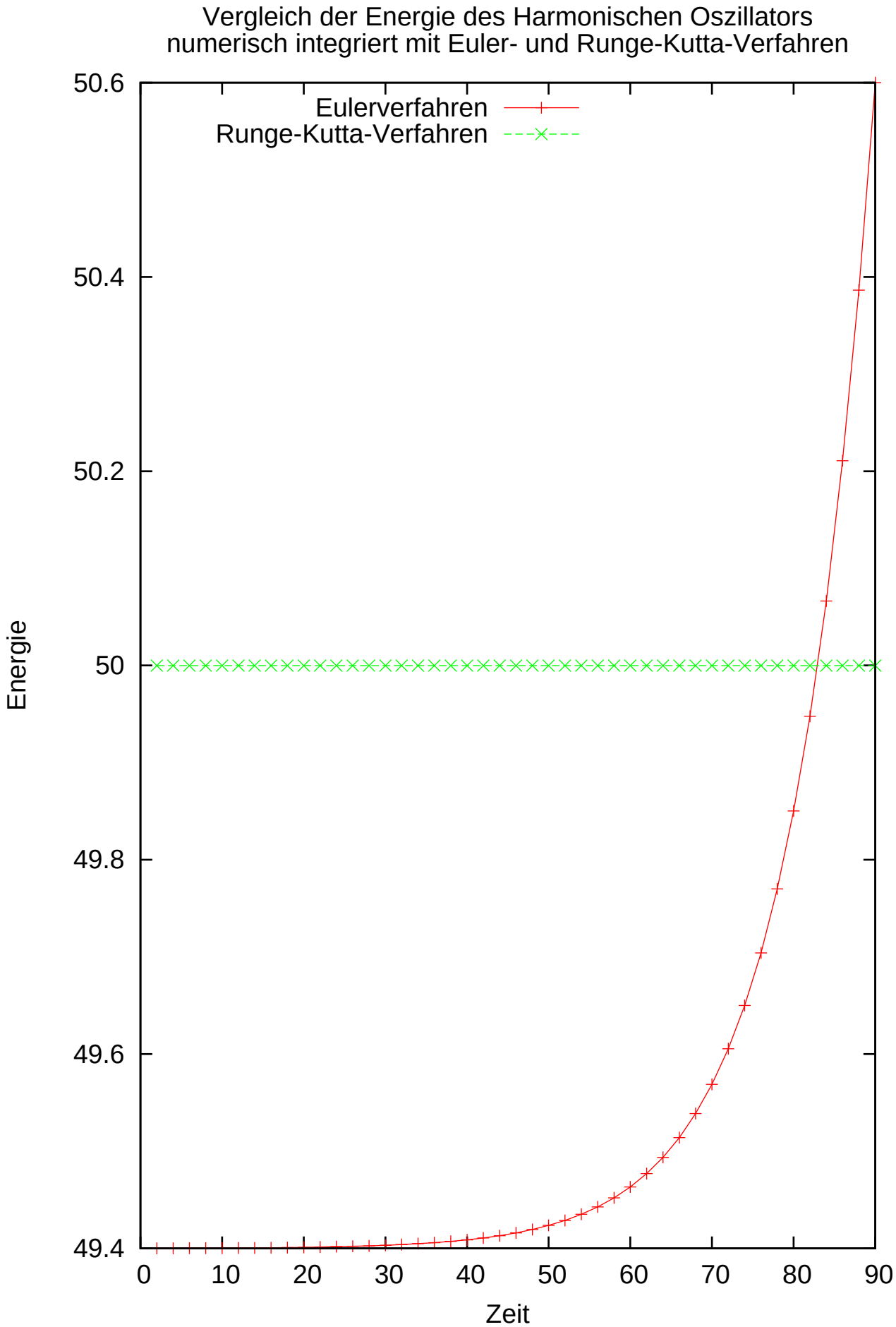
4.2 Extrapolation

Wie auch bei Integralen oder Ableitungen (vgl. Kap. 2.6 kann man die Genauigkeit mittels Extrapolation steigern: Ein Wert u_{l+1} wird mit mehreren Schrittweiten h_i berechnet

$$u_h := u_{l+1}, \text{ berechnet mit } h_l = h$$

und die Werte dafür interpoliert – also die Punktwolke $(h_i u_{h_i})$. Anschließend wird in der Interpolation $h \to 0$ ausgeführt und dieser Wert verwendet.

Für ein Einzelschrittverfahren der Konsistenzordnung $p \geq 1$ kann man den


Vergleich der Energie des Harmonischen Oszillators
numerisch integriert mit Euler- und Runge-Kutta-Verfahren
Eulerverfahren
Runge-Kutta-Verfahren
Energie
Zeit
50.6
50.4
50.2
50
49.8
49.6
49.4
0
10
20
30
40
50
60
70
80
90

durch die endliche Größe von h verursachten fehler abschätzen via

$$u_h(t) - y(t) = c_p(t)h^p + c_{p+1}h^{p+1} + ... + c_{p+r-1}h^{p+r-1} + O(h^{p+r}) \ .$$

Bildet man nun eine Endliche Folge $h_o > h_1 > ... > h_m$ mit $0 \leq m \leq r$ und approximiert die Punkt (h_i, u_{h_i}) mit einem Polynom

$$P_{0,...,m}(h) = d_0 + d_{p+1}h^{p+1} + ... + d_{p+m-1}h^{p+r-1} \ ,$$

dann lässt sich das Verfahren um die Ordnung m verbessern:

$$y(t) = P_{0,...,m}(0) + O(h^{p+m}) \ .$$

4.3 Schrittweitensteuerung

Der Schritt $(t_l, u_l) \to (t_{l+1}, u_{l+1})$ wird mit einem Zwischeschritt $(t_l, u_l) \to (t_{l+\frac{1}{2}}, u_{l+\frac{1}{2}}) \to (t_{l+1}, u_{l+1})$ gemacht:

$$w = u_l + \frac{h_l}{2}\varphi(t_l, u_l, \frac{h_l}{2} \text{ und } u_{l+1} = w + \frac{h_l}{2}\varphi(t_l + \frac{h_l}{2}, w, \frac{h_l}{2}) \ . \tag{4.17}$$

Es soll nun die Schrittweite h_l so gewählt werden, dass für einen fest gewählten absoluten Fehler $\varepsilon > 0$

$$\|u_{l+1} - z(t_l + h_l)\| \approx \varepsilon \ , \tag{4.18}$$

wobei z eine Lösung für $z' = f(t, z)$ mit $z(t_l) = u_l$ ist.

Dieses „$\approx$“ macht deshalb Sinn, weil bei großen Fehlern genauer approximiert werden soll, das h für kleine Fehler aber ruhig größer werden darf, weil man sonst Rundungsfehler bekommen kann.

Analog kann man auch als Abbruchbedingungen

$$\varepsilon_1 \leq \|u_{l+1} - z(t_l + h_l)\| \leq \varepsilon_2$$

wählen.

Das z muss logischerweise auch mit einem anderen Verfahren bestimmt werden – oder man schätzt den gesamten Fehler $\|\cdot\|$ ab. Sei $\tilde{u}$ der mit (4.17) bestimmte Wert u_{l+1}, dann kann man setzen

$$z_h = \tilde{u} - \frac{u_l + h\varphi(t_l, u_l, h) - \tilde{u}}{2^p - 1} = z(t_l + h) + O(h^{p+2}) \ , \tag{4.19}$$

wobei man für den Fehler in (4.18) die Abschätzung

$$\delta(h) = \|\tilde{u} = z_h\| = \frac{\|u_l + h\varphi(t_l, u_l, h) - \tilde{u}\|}{2^p - 1} \tag{4.20}$$

erhält – was offensichtlich proportional zum Fehler zwischen dem Verfahren mit und dem ohne Zwischenschritt ist; besagter Faktor ergibt sich aus diversen Rest-Abschätzungen.

Erfüllt dieses $\delta(h) \approx \varepsilon$, so kann man das h verwenden. Wenn nicht, so muss man $h \mapsto h' < h$.

Dieses h' wählt man (für $h \ll \varepsilon^{1/(p+2)}$) am besten via

$$h' = \left(\frac{\varepsilon}{\delta(h)}\right)^{1/(p+1)} \cdot h \,, \tag{4.21}$$

was sich aus weiteren Restterm-Abschätzungen ergibt. Daraus folgt auch, dass ein günstiger Startwert

$$h_0 = \varepsilon^q \quad \text{mit } 1 < q < \frac{1}{2+p} \tag{4.22}$$

ist.

Der Algorithmus ist also insgesamt:

1. Wähle $h = h_0$ nach (4.22).
2. Berechne aus u_l: u_{l+1} und $\tilde{u}$
3. Schätze den Fehler ab mit (4.20)
4. Ist die Bedingung (4.18) erfüllt, so wähle $u(t_{l+1}) = \tilde{u}$; wenn *nicht*, so verfeinere h mittels (4.21) und beginne wieder bei 1.

Energieerhaltung Eine elegante Methode in Systemen mit erhaltener Energie ist auch, anstatt der Forderung (4.18) zu verlangen, dass $\Delta E < \varepsilon$ wobei ΔE die Abweichung der aktuell berechneten Energie zur tatsächlichen Energie (die man ja aus den Anfangsbedingungen berechnen kann) ist.

In Abb. 4.2 ist dargestellt, wie sich die Integration damit verbessert: Hier sieht man auf der Y2-Achse wie starkt das normale Eulerverfahren auseinanderdriftet. Für das „symplektische“ Verfahren wurde die Schrittweite jedes mal halbiert, wenn die Differenz zwischen aktueller und vorhergehender Energie größer als 0.5 war (die Energie für die Anfangsbedingungen betrug 50).

Im Gegensatz dazu sieht man aber auch, wie gut das Runte-Kutta-Verfahren funktioniert – jedoch muss man bedenken, dass der Aufwand hier auch *wesentlich* höher ist.

Vergleiche zu dieser Idee auch Kap. 4.4.

4.4 Symplektische Integration

Die Symplektische Integration ist ein Verfahren, bei dem physikalische Gegebenheiten eingesetzt werden, um – physikalische – Differenzialgleichungen exakter integrieren zu können. Grundlage dazu bildet der Hamiltonformalismus, wobei für uns nötig ist, dass sich der Hamiltonian H schreiben lässt via

$$H(p,q) = T(p) + V(q) \,. \tag{4.23}$$

Mit den Hamilton-Jacobi-DGL

$$\dot{p} = -\frac{\partial H}{\partial q} = -\frac{\partial V}{\partial q} \text{ und } \dot{q} = \frac{\partial H}{\partial p} = \frac{\partial T}{\partial p} \tag{4.24}$$

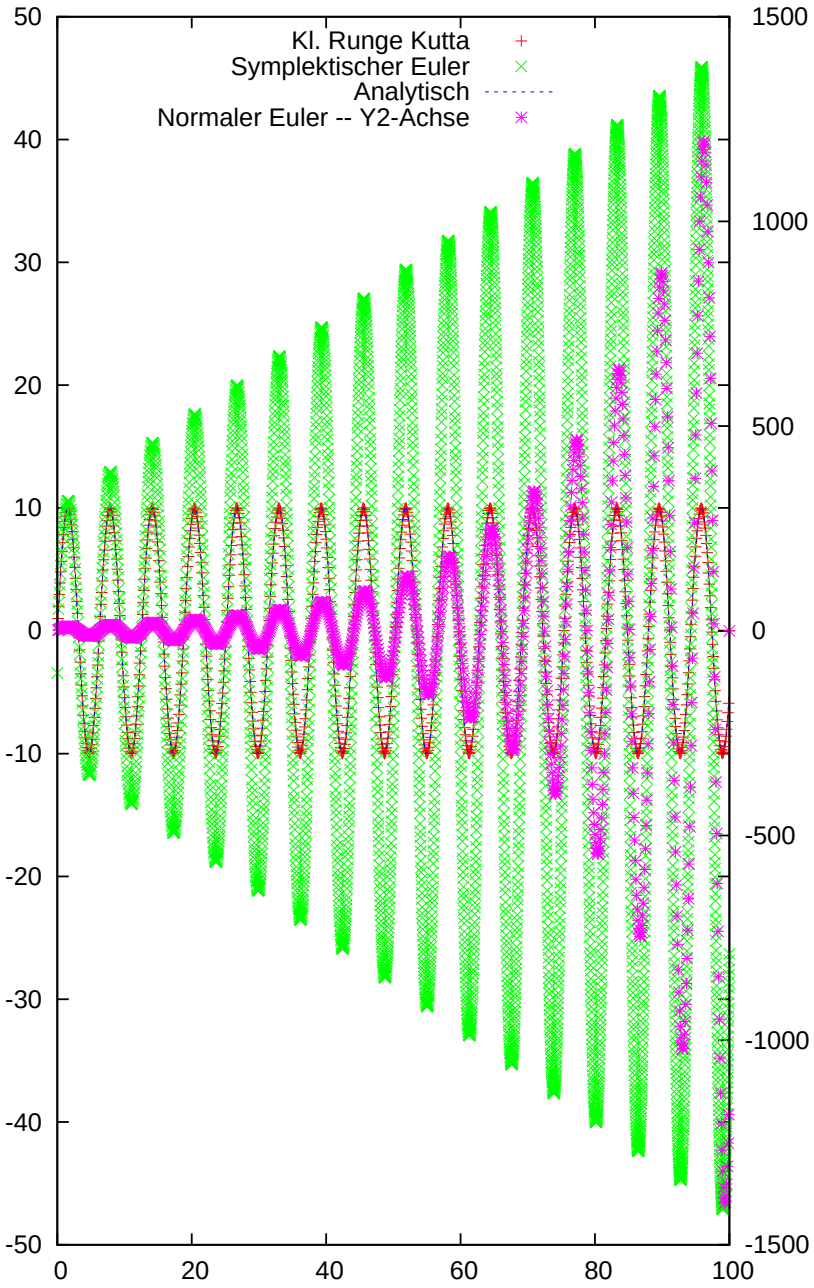


Abbildung 4.2: Vergleich zwischen normalem Euler, Euler mit Schrittweitensteuerung durch Energierhaltung und Runge Kutta für den Harmonischen Oszillator

für den Spezialfall (4.23) kann man Verfahren herleiten, bei denen die Gesamtenergie $T+V$ besser erhalten bleibt[2], als bei vergleichbaren Verfahren ähnlicher Ordnung.

Dazu betrachtet man Bewegungen als Vektor $\boldsymbol{z} = (q,p)^T$; dann ist die Bewegungsdgl

$$\dot{\boldsymbol{z}} = \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}\,t} & 0 \\ 0 & \frac{\mathrm{d}}{\mathrm{d}\,t} \end{pmatrix} \cdot \boldsymbol{z} =: D_H \boldsymbol{z} = (D_T + D_V)\boldsymbol{z} \;, \tag{4.25}$$

wobei die letzte Gleichung sich ergibt, weil man diese Gleichung mit den Poissonklammern auch als[3]

$$\dot{\boldsymbol{z}} = \{\boldsymbol{z}, H\} = \{\boldsymbol{z}, T\} + \{\boldsymbol{z}, V\}$$

schreiben kann. Eine DGL der Form (4.25) löst man durch den Exponentialmatrixansatz:

$$\boldsymbol{z}(t) = \exp(tD_H)\,\boldsymbol{z}(0) = \exp(tD_T + tD_V)\,\boldsymbol{z}(0) \;. \tag{4.26}$$

Diese Exponentialfunktion kann man nun umschreiben in ein Produkt

$$\exp[t(D_T + D_V)] = \prod_{i=1}^{k} \exp(c_i t D_T)\exp(d_i t D_V) + O(t^{n+1}) \;,$$

mit gewissen Koeffizienten c_i und d_i. $k \in \mathbb{N}$ heißt *Ordnung* des Verfahrens. Für die einzelnen „kleinen“ Exponentialmatrizen gilt nun

$$\exp(c_i t D_T) : \begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q' \\ p' \end{pmatrix} = \begin{pmatrix} q + t c_i \frac{\partial T}{\partial p}(p) \\ p \end{pmatrix}$$

und

$$\exp(d_i t D_V) : \begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q' \\ p' \end{pmatrix} = \begin{pmatrix} q \\ p - t d_i \frac{\partial V}{\partial q}(q) \end{pmatrix} \;.$$

4.4.1 Euler-Chromer-Verfahren

Für $k = 1$ und die Zahlen $c_1 = d_1 = 1$ erhält man so[4] für $t = \Delta t = h$ für den Zeitschritt $t \to t + \Delta t$:

$$v_{n+1} = v_n + h\, g(t_n, y_n) \tag{4.27}$$
$$y_{n+1} = y_n + h\, v_{n+1} \;, \tag{4.28}$$

wobei g von der DGL $\ddot{y} = g(t,y)$ kommt.

[2] Präziser: es bleibt das Phasenraumvolumen $\mathrm{d}\,p \wedge \mathrm{d}\,v$ erhalten

[3] Vlg auch Von-Neumann-Gl. oder allgemein Hamilton-Formalismus

[4] das Produkt der Exponentialmatrizen ist als Komposition von rechts nach links zu verstehen

4.4.2 Verlet-Methode

Für $k = 2$ und mit $c_1 = c_2 = \frac{1}{2}$, $d_1 = 1$ und $d_2 = 0$ erhält man

$$y_{l+1} = 2y_l - y_{l-1} + h^2\, g(t, y_l) \tag{4.29}$$

wobei g wieder die Zweite Ableitung bezeichnet: $\ddot{y} = g(t, y)$.

Dies ist eine Methode der Ordnung 4 – aber eigentlich ist es kein klassisches Einschrittverfahren, weil auf der rechten Seite auch y_{l-1} auftaucht.

Energieerhaltung

Diese Methode eignet sich natürlich gut, wenn man eine Funktion integrieren will, bei der die Energie erhalten bleibt – beim angegebenen Hamiltonian (4.23) ist das der Fall. Sobald jedoch Reibungskräfte etc. (Dissipative Elemente in H) auftreten, wird das Verfahren nicht mehr so gut sein...

4.5 Chaos

4.6 Molekulardynamik

Hier betrachtet man N Teilchen – jedoch mit sehr großen N – die miteinander wechselwirken, bspw. Gase, Flüssigkeiten, Membrane, Schüttgüter, Moleküle, Atome, ... Hier kommt es im Allgemeinen nicht unbedingt zu genau auf die einzelnen Bahnen der Teilchen an, sondern auf das Verhalten der Gesamtheit – dementsprechend ist man meits nur an Größen interessiert, die die Gesamtheit der Teilchen konstituieren, bspw. Temperatur, Druck, Spannung, ...

In der Praxis sind hierbei noch $N \sim 10^8$ Teilchen praktikabel. Wählt man bspw. für Atome $\Delta t \sim 10^{-12}\,\mathrm{s}$, dann kann man $\sim 10^6$ Zeitschritte weit berechnen, also $t \sim 1\,\mu\mathrm{s}$.

Optimierungen Wenn jedes Teilchen mit jedem anderen in Wechselwrikung tritt, und man möchte jede einzelne der Wechselwrikungen berechnen, dann hat man (unter Ausnutzung von Actio = Reactio) $\frac{N(N-1)}{2} = O(N^2)$ Wechselwirkungen zu berechnen. Im Gegensatz zu bspw. Himmelsmechanik, wo man eine langreichweitige Wechselwirkung (Gravitation) betrachtet, kann man im Allgemeinen die Kraft eines Teilchens auf die Kraft reduzieren, die es auf seine (mehr oder weniger direkten) Nachbarn ausübt – i.A. also nur Teilchen, für die der Abstand $< R$ ist. In der Praxis verwendet man für dieses R:

$$R \sim 2.5\,\lambda\,,$$

wobei λ der *Mittlere Teilchenabstand* ist. Dies ist jedoch sehr stark abhängig von der Weitreichweitigkeit der Wechselwirkung.

Um dies Effektiv durchzuführen gibt es mehrere Verfahren:

Nachbartafeln auch *Verlet-Tafeln* genannt: Für jedes Teilchen wird eine Liste der Nachbarn mit $r < R'$ geführt. Es müssen dann nur diese untersucht werden, ob $r < R$ ist.

Der Aufwand hierfür – die Listen zu aktualisieren – ist jedoch auch $O(N^2)$; der Nutzen ist nicht groß. Es sei denn, dass man die Listen nicht nach jedem Zeitschritt aktualisiert, sondern erst nach mehreren!

Linked Lists Auch *Zell-Listen genannt*: Der Raum wird in Zellen (ein Gitter) mit Kantenlänte R aufgeteilt und nur Teilchen in den benachbarten Feldern müssen beachtet werden. Die Aktualisierung dieser Liste ist nur $O(N)$ aufwändig, weil man nur über alle Teilchen Iterieren muss um diese einer Zelle zuzuweisen.

Kombination Der Nachteil der Linked Lists ist natürlich, dass die Wechselwirkung im Allgemeinen Symmetrisch sein wird – also einen Ball mit Radius R um jedes Teilchen aufspannt, wohingegen die Zellen Eckig sind: Der Abstand zweier Teilchen in benachbarten Zellen kann so – wenn sie genau auf der Diagonalen sitzen – auch $\sqrt{2}R$ betragen. Das kann man entschärfen, indem man für die Teilchen einer Zelle wieder jeweise Nachbartafeln führt.

Kapitel 5

Mehrschrittverfahren für Anfangswertprobleme

todo...

Kapitel 6

Popylationsdynamik

6.1 Eine Spezies

6.1.1 Einfachste Modelle

Exponentielles Wachstum Geht man von dem einfachen Modell aus, dass eine Spezies eine gewisse Geburtenrate b und eine Mortalitätsrate m hat, so ergibt sich aus einer Population von P_n Wesen nach einem Zeitschritt

$$P_{n+1} = b\,P_n - m\,P_n = \alpha\,P_n\ ,\ \text{und damit}\ P_n = \alpha^n P_0\ .$$

In Form von Differenzialgleichungen bekommt man die rate α in die Änderung der Population:

$$\dot{y} = \tilde{\alpha}\,y \text{ und damit } y(t) = \mathrm{e}^{\tilde{\alpha}t} y(0)\ .$$

Logistisches Wachstum Berücksichtigt man darüber hinaus noch die Kapazität k eines gewissen Lebensraumes, so dämpft diese im einfachsten Modell das Wachstum α dahingehend, dass die Vermehrung nur noch mit $\alpha' = (\alpha - k\,P_n$ stattfindet: Je größer die Population bereits ist, desto weniger stark kann sich die Population vermehren. In Diskreter Form führt dies auf

$$P_{n+1} = (\alpha - kP_n)P_n$$

und in differenzieller auf

$$\dot{y} = (\tilde{\alpha} - \tilde{k}y)y\ .$$

Diese Form kann man lösen und findet die drei Funktionen:

$$y_1 = \frac{\tilde{\alpha}/\tilde{k}}{1 + (\frac{\tilde{\alpha}/\tilde{k}}{y_0} - 1)\mathrm{e}^{-\tilde{\alpha}t}}, y_2 = 0, y_3 = k$$

Unterschied diskret–differenziell Bei der Formulierung mit DGLs kann man das „Arsenal“ der Analysis auffahren und die Gleichungen lösen (versuchen) – in diskreter Form fehlen diese Hilfsmittel. Für große Populationen ist es auch nicht mehr relevant, ob man mit einer Funktion bspw. 149.23 oder 149 Wesen ausrechnet – der Unterschied kann getrost Vernachlässigt werden.

Wieter ist wichtig, dass bei den differenziellen Modellen eine Zeitskala gegeben ist – die man auch weiter verfeinern kann.

Nachteile sind jedoch, dass

- kleine Änderungen haben großen Einfluss
- Erzwingen einer gewünschten Population – später eines bestimmten Verhältnisses – ist schwierig

Rauschen

Um die Stabilität einer Lösung zu testen, adiert man ein *Rauschen*, bspw.

$$\xi = A\cos(t)$$

mit $A \ll y_0$ (A soll gegen y klein sein) und betrachtet dann in der DGL $y_\xi = y+\xi$. Wenn das Ergebnis dadurch nur ein wenig „wackelt“ hat man einen stabile Lösung gefunden und eine instabile, wenn die Lösung sich sofort stark verändert.

6.2 Zwei und mehr Spezies

Hier betrachtet man meist zwei Konkurrenten; bspw. Räuber und Beutetiere.

Diese Modelle sind jedoch nach wie vor Simpel, weil sie keinerlei Fluktuationen aufweisen und nicht räumlich auflösen – also die Formeln sind so, als würden sich alle Tiere an einem Ort aufhalten.

6.2.1 Einfaches Modell: Volterra-Gleichung

Die Kapazitäten seien unbegrenzt; die Beutetiere sterben nur, wenn sie gefressen werden und die Räuber sterben nur, wenn sie nicht genug Beute bekommen – also verhungern. Die Geburtenrate ist proportional zur Population.

Bezeichne Beute mit y_1 und Räuber mit y_2; m ist die Mortalitätsrate, b die Geburtenrate:

$$\dot{y}_1 = (b_1 - r_{21}y_2)y_1 \ , \tag{6.1}$$
$$\dot{y}_2 = (-m_2 + R_{12}y_1)y_2 \ . \tag{6.2}$$

6.2.2 Lotka-Gleichung

Das System wird entwas verfeinert: Die Zahl der Bäutetiere ist jetzt durch eine Kapazität begrenzt; zu (6.2) kommt jetzt

$$\dot{y}_1 = (b - r_{21}y_2 - \frac{1}{k}y_1)y_1 \ . \tag{6.3}$$

Der neue Term y_1/k wirkt wie eine Reibung auf das System; es ist möglichk, dass es zu Gleichgewichten kommt und das Aussterben von Arten ist möglich.

6.3 Diskrete Modelle

Den in 6.2 angesprochenen Mängeln versucht man jetzt zu begegnen: Der Lebensraum der Wesen wird in ein Gitter zerlegt und pro Gitterfeld werden Zahl und Spezies festgehalten.

In dem neuen Modell gibt es jetzt Aktivitäten; bspw

Migration Mit bestimmter Wahrscheinlichkeit bewegt isch ein Tier auf einen anderen Platz

Jagd Räuber machen Beute, wenn die Beutetiere in einer gewissen Umgebung sind

Geburt/Tod Satte Räuber haben mehr Nachkommen, hungrige sterben; Tiere sterben durch Alter ...

Beobachtet man ein solches System, so bemerkt man, dass das Verhalten stark von der Größe des Lebensraumes abhängig ist:

zu klein Die Beute wird ausgerottet – und die Räuber damit auch

mittel Die Bestände oszillieren

zu groß Populationen sind i.A. konstant, nur kleine Fluktuationen

Für diese Modelle ist es offensichtlich nötig, sich von einer deterministischen Beschreibung des Systems zu lösen und zu einer stochastischen überzugehen: Bei den einzelnen Aktionen soll der *Zufall* eine (große) Rolle spielen.

Um solche Systeme also zu modellieren, sind gute Zufallszahlen-Generatoren nötig.