RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LuFG Informatik 12 (Prof. Dr. Uwe Naumann)

Automatisches Lösen gewöhnlicher Differentialgleichungen mit Taylorkoeffizentenalgebra

Seminararbeit

Martin Beeger (Matr.-Nr. 300976)

1. April 2017

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Martin Beeger

Aachen, den 1. April 2017

Inhaltsverzeichnis

1 Einleitung **3**

2 Repräsentation als Taylorreihe **3**

3 Implementation in C++ **3**

4 Differentiation von Taylorreihen **5**

5 Tayloralgebra **6**

6 Lösung von Differentialgleichungen **8**

7 Verallgemeinerungen **12**
7.1 Nichtautonome DGL 12
7.2 Höhere Ordnung und Systeme von DGL 14

A **15**

B Quellcode **15**

1 Einleitung

Eine gewöhnliche Differentialgleichung mit Anfangsbedingung ist eine implizite Darstellung einer Funktion, zum Beispiel ist $x(t) = e^{5x}$ Lösung der Gleichung

$$\dot{x}(t) = 5x(t), x(0) = x_0 = 1. \tag{1}$$

Diese Seminararbeit entwickelt eine Repräsentation solcher Funktionen als abgeschnittene Taylorreihen in C++ und setzt die dazu nötige Algebra mittels Operatorüberladung um. Es werden die Eigenschaften der Taylorentwicklung in Bezug auf Differentiation verwendet, um ein Lösungsverfahren für gewöhnliche Differentialgleichungen zu entwickeln. Die Implementation in C++ kann damit zum effizienten Lösen geeigneter DGLs verwendet werden.

2 Repräsentation als Taylorreihe

Zur arithmetischen Berechnung einer Lösung benötigen wir zunächst einen Weg, implizit oder explizit definierte Funktionen in einer generischen Datenstruktur darstellen zu können.

[BWZ71] verwenden dafür folgende Taylorreihenentwicklung: Gegeben sei eine Funktion $x(t) \in C^N(t_a)$ und ein Entwicklungspunkt $t_a \in \mathbb{R}$. Jede analytische Funktion kann dargestellt werden als

$$x(t) = \sum_{n=0}^{N} \frac{x^{(n)}(t_a)}{n!}(t - t_a)^n + O((t - t_a)^{N+1}) \quad \forall t - t_a < 1 \tag{2}$$

Um also eine lokale Näherung der Funktion in C++ abzubilden, wird eine Darstellung als Koeffizientenvektor $ts \in \mathbb{R}^{\mathbb{N}}$ der abgeschnittenen Taylorreihe gewählt. Die Einträge ergeben sich zu

$$ts_i = \frac{x^{(i)}(t_a)}{i!} \tag{3}$$

Der Vektor ts besteht aus den skalierten Koeffizienten der Taylorentwicklung und ist damit die Repräsentation der Funktion in C++ in Vektorform.

3 Implementation in C++

Die Implementation führt den Datentypen *ts* (kurz für *taylor series*) nach obiger Definition als Datentyp ein. Das wesentliche Interface sieht wie folgt aus:

Listing 1: Klassendefinition der Taylor-Repräsentation

```
template <typename sc>
class ts: public std::vector<sc> {
public:
    ts() = default;
    ~ts() = default;
    ts(const ts<sc>&) = delete;
    constexpr ts(ts<sc>&&) = default;
    ts(const sc& val);
    ts(std::vector<sc>&& base);
    static ts<sc> fill(int size, sc val = 0);
};
using tsd = ts<double>;
```

Hierbei bezeichnet sc (kurz für *scalar*) den Fließkommazahltypen, üblicherweise double, aber mit begrenztem Set an Operationen (keine Division) ist auch int möglich.

Aufgrund eines Performance-Hits und zur einfacheren Verifikation der Implementation wird das direkt Kopieren untersagt, darüber hinaus kann eine Taylorreihe aus einer Konstante erzeugt werden, da für $x(t) = C$ gilt:

$$x^{(n)}(t) = 0 \quad \forall n > 0 \Rightarrow ts = (C, 0, \cdots, 0) \tag{4}$$

Es wird in der Implementation ein std::vector als Container verwendet, was gegenüber std::array den Vorteil hat, dass die Größe nicht zur Compilezeit bestimmt sein muss, allerdings wird bei einem std::vector üblicherweise zusätzlicher Speicherplatz verbraucht.

Bei binären Operationen ergibt sich zum Teil das Problem, dass die Länge der Taylorreihe der beiden Operanden ggf. variieren kann. Für eine solche Datenstruktur ist jedoch das korrekte Verhalten bei „out-of-bounds“-Zugriff mathematisch klar definiert. Deswegen wird der geprüfte Zugriff wie folgt überladen:

```
template <typename sc>
sc ts::at(size_t i) const
{
  if(i < this->size())
    return this->operator[](i);
  return 0;
}
```

Dadurch muss bei der Operatorüberladung keine spezielle Semantik für abweichende Längen implementiert werden.

4 Differentiation von Taylorreihen

Damit die Datenstruktur zum Lösen von Differentialgleichungen verwendet werden kann, muss neben der Datenstruktur auch grundlegende Algebra zur Verfügung stehen. Dies geschieht in C++ mittels Operatorüberladung für den oben definierten Typen, wodurch eine natürlich Syntax erhalten bleibt. Die in Bezug auf Differentialgleichungen interessanteste Operation ist die Differentiation, welche im Folgenden behandelt wird.

In dieser Darstellung kann eine explizite Formel für Differentation angegeben werden, wie folgende Herleitung zeigt:

$$x'(t) = y(t) \tag{5}$$

$$(\sum_{n=0}^{\infty} ts(x)_n (t-t_a)^n)' = \sum_{n=0}^{\infty} ts(y)_n (t-t_a)^n) \tag{6}$$

$$(\sum_{n=0}^{\infty} ts(x)_n n (t-t_a)^{n-1}) = \sum_{n=0}^{\infty} ts(y)_n (t-t_a)^n) \tag{7}$$

$$(\sum_{n=-1}^{\infty} ts(x)_{n+1} (n+1)(t-t_a)^n) = \sum_{n=0}^{\infty} ts(y)_n (t-t_a)^n) \tag{8}$$

$$ts(x)_{n+1}(n+1) = ts(y)_n \quad \forall n = 0, 1, 2, \cdots \tag{9}$$

$$ts(x)_{n+1} = \frac{ts(y)_n}{(n+1)} \quad \forall n = 0, 1, 2, \cdots \tag{10}$$

Der Eintrag $ts(x)_0$ kann nicht aus dieser sogenannten „recurrence relation“ gewonnen werden. Hier zeigt sich die Bedeutung der Anfangsbedingung für die Bestimmtheit der Differentialgleichung, denn es gilt für eine Anfangsbedingung der Form:

$$x(t_0) = x_0 \tag{11}$$

bei Wahl des Entwicklungspunktes $t_a = t_0$:

$$x_0 = x(t_0) = \sum_{n=0}^{\infty} ts(x)_n (t_0 - t_a)^n = ts(x)_0 \tag{12}$$

Diese Methode erlaubt uns die Approximation der Ableitung einer Funktion. Die meisten gewöhnlichen Differentialgleichungen bestehen aus zusätzlichen Termen auf der rechten Seite, die Rechenoperationen verwenden. Dies gilt für unser eingangs erwähntes Beispiel $x'(t) = 5x(t)$. Deswegen ist der nächste Schritt die Definition und Implementation von Operatoren wie Addition, Subtraktion, Multiplikation, usw für unsere Taylorreihen.

5 Tayloralgebra

Die Addition und Subtraktion in der Form $f(t) = g(t) \pm h(t)$ kann wie folgt hergeleitet werden:

$$f(t) = g(t) \pm h(t) \tag{13}$$

$$\sum_{n=0}^{\infty} ts(f)_n (t-t_a)^n = \sum_{n=0}^{\infty} ts(g)_n (t-t_a)^n \pm \sum_{n=0}^{\infty} ts(h)_n (t-t_a)^n \tag{14}$$

$$\sum_{n=0}^{\infty} ts(f)_n (t-t_a)^n = \sum_{n=0}^{\infty} ts(g)_n \pm ts(h)_n (t-t_a)^n \tag{15}$$

$$ts(f)_n = ts(g)_n \pm ts(h)_n \quad \forall n = 0, 1, 2, \cdots \tag{16}$$

Listing 2: Implementation der Addition

```
1 template <class sc>
2 ts<sc> operator+ (const ts<sc>& left, const ts<sc>&
    right)
3 {
4     auto res = create<sc>(std::max(left.size(), right.
        size()));
5     for(size_t k = 0; k < res.size(); ++k)
6         res[k] = left.at(k) + right.at(k);
7     return res;
8 }
```

Da Konstanten in Taylorreihen konvertiert werden können, sind damit bereits Ausdrücke wie $f(t) = 4 + g(t) - 8 - h(t)$ behandelt. Der Aufwand für die Berechnung ist $O(N)$.

Multiplikation in der Form $f(t) = g(t)h(t)$ wird wie folgt umgesetzt:

$$f(t) = g(t)h(t) \tag{17}$$

$$\sum_{n=0}^{\infty} ts(f)_n (t-t_a)^n = (\sum_{n=0}^{\infty} ts(g)_n (t-t_a)^n)(\sum_{n=0}^{\infty} ts(h)_n (t-t_a)^n) \tag{18}$$

Dieses Produkt aus zwei Summen generiert Kreuzterme, die jeweils für gleiche Potenzen von $(t-t_a)^n$ zusammengefasst werden müssen. Hierfür wird eine

begrenzte Länge der Taylorreihe N angenommen.

$$f(t) = g(t)h(t) \tag{19}$$

$$\sum_{n=0}^{N} ts(f)_n (t-t_a)^n = (\sum_{k=0}^{N} ts(g)_n (t-t_a)^k)(\sum_{i=0}^{N} ts(h)_i (t-t_a)^i) \tag{20}$$

$$\sum_{n=0}^{N} ts(f)_n (t-t_a)^n = \sum_{k=0}^{N} \sum_{i=0}^{N} ts(g)_k ts(h)_i (t-t_a)^k (t-t_a)^i \tag{21}$$

$$\sum_{n=0}^{N} ts(f)_n (t-t_a)^n = \sum_{k=0}^{N} \sum_{i=0}^{N} ts(g)_k ts(h)_i (t-t_a)^{k+i} \tag{22}$$

Um hier eine Gleichung für die Koeffizienten auszustellen, muss die Summationsreihenfolge so verändert werden, dass $k+i=n$ gilt:

$$n = 0 \Rightarrow \quad (i=0, k=0) \tag{23}$$

$$n = 1 \Rightarrow \quad (0,1), (1,0) \tag{24}$$

$$n = 2 \Rightarrow \quad (0,2), (1,1), (2,0) \cdots \tag{25}$$

Dies führt dann zu folgender Relation der Taylorkoeffizienten:

$$ts(f)_n = \sum_{k=0}^{n} ts(g)_k ts(h)_{n-k} \quad \forall n = 0, 1, 2, \cdots \tag{26}$$

Listing 3: Implementation der Multiplikation

```
1 template <class sc>
2 ts<sc> operator*(const ts<sc>& left, const ts<sc>&
    right)
3 {
4     auto res = create<sc>(std::max(left.size(), right.
        size()));
5     for (size_t k = 0; k < res.size(); ++k)
6     {
7         for (size_t j = 0; j <= k; ++j)
8             res[k] += left.at(j) * right.at(k - j);
9     }
10    return res;
11 }
```

Um die Genauigkeit zu erhalten, muss bei einer Multiplikation die Länge der Ergebnisserie zu $|ts(f)| = |ts(g)| + |ts(h)| - 1$ gewählt werden. Die vorliegende

Implementation nimmt also einen Genauigkeitsverlust in Kauf. Zudem ist der Aufwand mit $O(|ts(f)|^2)$ um eine Größenordnung höher.

Für Divison ergibt sich folgende Relation:

$$ts(f)_n = \frac{1}{ts(h)_0}[ts(g)_n - \sum_{k=0}^{n-1} ts(g)_k ts(h)_{n-k}] \quad \forall n = 0, 1, 2, \cdots \tag{27}$$

Listing 4: Implementation der Divison

```
1 template <class sc>
2 ts<sc> operator/(const ts<sc>& left, const ts<sc>&
    right)
3 {
4     auto res = create<sc>(std::max(left.size(), right.
        size()));
5     for(size_t k = 0; k < res.size(); ++k)
6     {
7         for(size_t j = 0; j < k; ++j){
8             res[k] += res[j] * right.at(k - j);
9         }
10        res[k] = 1./right[0] * (left.at(k) - res[k]);
11    }
12    return res;
13 }
```

Hier ist zu beachten, dass diese Operation nur für $ts(h)_0 \neq 0$ definiert ist.

Mithilfe von [GW08, 308f] wurden die Taylor-Operationen auch für Sinus, Kosinus und Quadratwurzel implementiert. Dabei wird jedoch ein anderer Ansatz gewählt, der lediglich eine Näherung quadratischer Konvergenz gibt.

Die implementierte Überladung ist nicht vollständig, da dies den Rahmen dieser Arbeit sprengen würde, jedoch sind weite Teile gebräuchlicher Arithmetik abgedeckt.

6 Lösung von Differentialgleichungen

Um mithilfe der Taylorarithmetik die eingangs erwähnte Differentialgleichung nach $x(t)$ zu lösen, sind nun folgende Schritte erforderlich. Sei

$$\dot{x}(t) = 5x(t), x(0) = 1 \tag{28}$$

die zu lösende DGL und gesucht sei $x(1) = x_1$. Als Entwicklungspunkt für die Taylorentwicklung muss im ersten Schritt $t_a = 0$ gewählt werden, damit

die Anfangsbedingung genutzt werden kann um den ersten Koeffizienten zu bestimmen. Damit ergibt sich gemäß Gleichung (12) hier

$$ts(x)_0 = 1. \tag{29}$$

Umformung ergibt:

$$\begin{aligned} y(x(t)) = &\quad 5x(t) && \text{(30a)}\\ \dot{x}(t) = &\quad y(x(t)) && \text{(30b)} \end{aligned}$$

Unter Verwendung von Taylor-Arithmetik für Gleichung (30a) und Gleichung (10) für Gleichung (30b) ergibt sich:

$$\begin{aligned} ts(y)_n = &\quad 5ts(x)_n && \text{(31a)}\\ ts(x)_{n+1} = &\quad \tfrac{ts(y)_n}{(n+1)} && \text{(31b)} \end{aligned}$$

Unter Verwendung von Gleichung (29) ergibt sich folgender Lösungweg durch sukzessives Einsetzen in das Gleichungssystem (30):

$$\begin{aligned} ts(x)_0 = &\quad 1 && && \text{(32a)}\\ ts(y)_0 = &\quad 5ts(x)_0 && = 5 && \text{(32b)}\\ ts(x)_1 = &\quad \tfrac{ts(y)_0}{1} && = 5 && \text{(32c)}\\ ts(y)_1 = &\quad 5ts(x)_1 && = 25 && \text{(32d)}\\ ts(x)_2 = &\quad \tfrac{ts(y)_1}{2} && = \frac{25}{2} && \text{(32e)}\\ ts(y)_2 = &\quad 5ts(x)_2 && = \frac{125}{2} && \text{(32f)}\\ ts(x)_3 = &\quad \tfrac{ts(y)_2}{3} && = \frac{125}{6} && \text{(32g)}\\ &\quad \dots && && \text{(32h)} \end{aligned}$$

Eine analytische Betrachtung der hier entstehenden Reihe zeigt die Konvergenz gegen die analytische Lösung:

$$x(\Delta t) = \sum_{n=0}^{\infty} \frac{5^n}{n!}(\Delta t)^n = \sum_{n=0}^{\infty} \frac{(5\Delta t)^n}{n!} = e^{5n} \tag{33}$$

Damit ergibt sich $x_1 = e^5$. Hier

Im folgenden soll jedoch die begrenzte Folge als Approximation für ein geeignet zu wählendes Δt und die Ordnung $N = 3$ dienen:

$$\tilde{x}(\Delta t) = \sum_{n=0}^{N} ts(x)_n(\Delta t)^n = 1 + 5\Delta t + \frac{25}{2}(\Delta t)^2 + \frac{125}{6}(\Delta t)^3 \tag{34}$$

Der Fehler kann für $\Delta t < 1$ abgeschätzt werden durch

$$|x(\Delta t) - \tilde{x}(\Delta t)| = O((\Delta t)^4) \tag{35}$$

Es ist erforderlich die Schrittweite $\Delta t \ll 1$ zu wählen, um eine gute Approximationsgüte zu erreichen. Zur Berechnung von $x_1 \gg \Delta t$ wird dann iteriert. Die Auswertung der DGL bei Δt wird als neue Anfangsbedingung interpretiert und die Taylorreihe wird erneut um $t_a = \Delta t$ entwickelt, um die Approximation bei $2\Delta t$ zu erhalten.

Der Löser sieht nach obigem Schema dann in C++ wie folgt aus:

Listing 5: Repräsentation einer ODE 1. Ordnung mit Anfangsbedingung

```
1 /// represents a first-order ODE as of:
2 /// x'(t) = rhs(x) , x(t0) = x0
3 template <class sc>
4 struct ode_system {
5   using fct = std::function<ts<sc>(const ts<sc>&)>;
6     fct rhs;
7     double t0 = 0;
8     double x0;
9     ode_system(fct rhs, double t0, double x0): rhs(rhs)
          , t0(t0), x0(x0){}
10    ode_system(fct rhs, double x0): rhs(rhs), x0(x0){}
11 };
```

Listing 6: Algorithmus für ODE 1. Ordnung

```
1 // solves the given ode and returns a vector of
    equidistant solution points
2 template <class sc>
3 std::vector<double> solve(ode_system<sc> ode, double
    t_target, int order, double delta_t)
4 {
5     // computes number of necessary time steps
6     const int num_steps = floor((t_target - ode.t0) /
          delta_t) + 1;
7     std::vector<double> points;
8     points.reserve(num_steps);
9
10    double cur_x = ode.x0;
11    double cur_t = ode.t0;
12    for(int step_num = 0; step_num < num_steps; ++
          step_num)
```

```
13     {
14         points.push_back(cur_x);
15         tsd result{cur_x};
16         result.reserve(order);
17         for(int i = 0; i < order; ++i)
18         {
19             tsd y = ode.rhs(result);
20             result.emplace_back(y[i] / (i+1.));
21         }
22         cur_x = eval(result, delta_t);
23         cur_t += delta_t;
24     }
25     points.push_back(cur_x);
26     return points;
27 }
```

Listing 7: Lösung der Beispiel-ODE

```
1 // x_rhs(x(t)) = x(t) * 5
2 auto x_rhs = [](const tsd& x){ return x* 5.; };
3 // x'(t) = x(t) * 5, x(0) = 1
4 ode_system<double> ode(x_rhs, 1);
5 //computing x(1) = x1;
6 double t_target = 1;
7 // error is 1e-10, computational expense is 100 taylor
    evaluations
8 int order = 10;
9 double delta_t = 0.1;
10 // solve the system
11 auto points = solve(ode, t_target, order, delta_t);
12 for(int i = 0; i < int(points.size()); ++i)
13 {
14     auto analyticSolution = [](double t){ return exp(5*
        t); };
15     ASSERT_NEAR(analyticSolution(ode.t0 + i * delta_t),
        points[i],1e-8);
16 }
```

7 Verallgemeinerungen

7.1 Nichtautonome DGL

Da bei autonomen Systemen die rechte Seite ein Funktional von $x(t)$ ist, kann mithilfe der Taylorarithmetik direkt die Darstellung als $ts(y)$ berechnet werden. Für nicht autonome Systeme der Form $\dot{x}(t) = y(x(t), t)$ muss jedoch zusätzlich die direkte Abhängigkeit modelliert werden. Eine Möglichkeit ist die Gleichung in ein DAE-System umzuwandeln, alternativ kann aber die Gleichung auch zu

$$Id(t) = t \tag{36}$$
$$y(t) = y(x(t), Id(t)) \tag{37}$$
$$\dot{x}(t) = y(t) \tag{38}$$

erweitert werden. Dabei gilt

$$ts(Id) = (t_a, 1, 0 \cdots) \tag{39}$$

Dann kann das Lösungsverfahren wieder wie vorher angewendet werden, dazu muss in jedem Zeitschritt die Taylorreihe $ts(Id)$ neu gesetzt werden. Der Lösungsalgorithmus sieht dann wie folgt aus:

Listing 8: Repräsentation einer nicht-autonomen ODE 1. Ordnung mit Anfangsbedingung

```
1 /// represents a first-order ODE as of:
2 /// x'(t) = rhs(x) , x(t0) = x0
3 template <class sc>
4 struct nonautonomous_ode_system {
5   using fct = std::function<ts<sc>(const ts<sc>&, const
        tsd&)>;
6   fct rhs;
7   double t0 = 0;
8   double x0;
9   nonautonomous_ode_system(fct rhs, double t0, double
       x0): rhs(rhs), t0(t0), x0(x0){}
10  nonautonomous_ode_system(fct rhs, double x0): rhs(rhs
       ), x0(x0){}
11 };
```

Listing 9: Löser für nicht autonome Systeme

```
1 template <class sc>
```

```
2 std::vector<double> solve(nonautonomous_ode_system<sc>
    ode, double t_target, int order, double delta_t)
3 {
4     // computes number of necessary time steps
5     const int num_steps = floor((t_target - ode.t0) /
        delta_t) + 1;
6     std::vector<double> points;
7     points.reserve(num_steps);
8
9     double cur_x = ode.x0;
10    double cur_t = ode.t0;
11    for(int step_num = 0; step_num < num_steps; ++
        step_num)
12    {
13        points.push_back(cur_x);
14        tsd result{cur_x};
15        result.reserve(order);
16        for(int i = 0; i < order; ++i)
17        {
18            tsd id{{cur_t, 1}};
19            tsd y = ode.rhs(result, id);
20            result.emplace_back(y[i] / (i+1.));
21        }
22        cur_x = eval(result, delta_t);
23        cur_t += delta_t;
24    }
25    points.push_back(cur_x);
26    return points;
27 }
```

Der einzige relevante Unterschied ist in Zeile 18f, wo die Identitätskoeffizenten gesetzt werden. Um die Beispiel - ODE:

$$\dot{x}(t) = x(t)\frac{2}{t}, x(1) = 2 \tag{40}$$

mit Lösung $x(t) = 2x^2$ zu lösen, braucht es folgenden Code:

Listing 10: Lösung der Beispiel-ODE

```
1 // x_rhs(x(t)) = x(t) * 2 /t
2 auto x_rhs = [](const tsd& x, const tsd& id){ return x*
    2. / id; };
3 // x'(t) = x(t) * 2/t , x(1) = 2
```

```
4 nonautonomous_ode_system<double> ode(x_rhs, 1., 2.);
5 //computing x(2) = x1;
6 double t_target = 2.;
7 // error is 1e-10, computational expense is 10/0.1 =
    100 taylor evaluations
8 int order = 10;
9 double delta_t = 0.1;
10 // solve the system
11 auto points = solve(ode, t_target, order, delta_t);
12 for(int i = 0; i < int(points.size()); ++i)
13 {
14     auto analyticSolution = [](double t){ return 2*t*t;
         };
15     ASSERT_NEAR(analyticSolution(ode.t0 + i * delta_t),
         points[i],1e-8);
16 }
```

7.2 Höhere Ordnung und Systeme von DGL

Jedes System von gewöhnlichen DGL beliebiger Ordnung lässt sich auf ein System erster Ordnung reduzieren. Der vorgestellte Löser lässt sich auf solche Systeme von DGL erster Ordnung erweitern. Gegeben ein System der Form:

$$\dot{X}(t) = A(X(t), t) \tag{41}$$

wobei $X(t)$ der Vektor der zeitabhängigen Zustandsfunktionen ist. Eine effiziente Implementation eines solchen Vektors ist nicht trivial, da der offensichtliche Ansatz

```
1 template<sc>
2 using ts_vector = std::vector<ts<sc>>;
```

erhebliche Performanceprobleme wegen der Indirektion hat. Gegeben eine solche effiziente Implementation wird dann analog das System separiert in

$$Y(t) = A(X(t), t) \tag{42a}$$
$$\dot{X}(t) = Y(t) \tag{42b}$$

und der Löser arbeitet dann analog mit dem ts_vector statt mit ts.

A

Literatur

[BWZ71] David Barton, IM Willers, and RVM Zahar. The automatic solution of systems of ordinary differential equations by the method of taylor series. *The Computer Journal*, 14(3):243–248, 1971.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

B Quellcode

Der Quellcode findet sich im STCE-Gitlab in dem Projekt TEDES.