NUMERIK – LINUS STUHLMANN

RECHNERARITHMETIK

FLOAT-DARSTELLUNG IM COMPUTER

Rechner verwenden eine binäre Darstellung von Zahlen. Diese werden in 64 Bit gespeichert. Das erste Bit definiert das Vorzeichen, die nächsten 11 Bit den Exponenten und die folgenden 52 Bit die Mantisse.

$$\pm M * 2^E$$

- $M \in$: ist die Mantisse (52 Bit)
- $E \in \mathbb{Z}$: der Exponent (11Bit)
  - $-1022 \leq E \leq 1023$

NUMERISCHE TAYLOR-ENTWICKLUNG

Mit Taylor-Reihen können Funktionen als Polynome um einen Entwicklungspunkt x_0 dargestellt werden.

$$t_f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

$$t_f(x) = a_0 + a_1(x - x_0)^1 + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n$$

$$a_0 = \frac{f(x_0)}{0!}$$

$$a_1 = \frac{f'(x_0)}{1!}$$

$$\vdots$$

$$a_k = \frac{f^{(k)}(x_0)}{k!}$$

Durch mehrfaches Ableiten und Einsetzen des Entwicklungspunkts, werden die Koeffizienten a_k des Polynoms ermittelt. Wenn die Taylor-Reihe gegen Unendlich strebt, strebt auch der Fehler zwischen der nachzubildenden Funktion gegen Null.

WICHTIGSTE BEKANNTE TAYLORREIHEN

Funktionen $e^x, \sin(x), \cos(x)$ an der Stelle $x_0 = 0$:

$$t_{e^x}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

$$t_{sin}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

$$t_{cos}(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

$$t_{ln}(x) = \sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{k}(x-1)^k, \; x_0 = 1$$

RESTGLIED

Der exakte Fehler an der Stelle x.

$$R_k(x) = f(x) - t_k(x) = \sum_{k=0}^{n} \frac{f^{(k+1)}(z)}{(k+1)!}(x - x_0)^{k+1}$$

- z : liegt dabei im Intervall $[x_0, x]$ ist aber unbekannt. (Zwischenwert Theorem)
- Die obere Fehlerschranke wird dabei oft mit dem grössten Wert im Intervall berechnet.
  - $z = x$
  - Maximaler Fehler (konservativste Fehlerschätzung)

LINEARISIERUNG

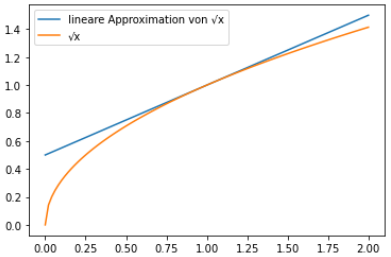
Durch Linearisierung können nicht lineare Funktionen um eine Entwicklungsstelle x_0 linear approximiert werden.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

BEISPIEL

Die Wurzelfunktion um den Entwicklungspunkt $x_0 = 1$.

$$\sqrt{x} \approx 1 + \frac{1}{2}(x - 1)$$

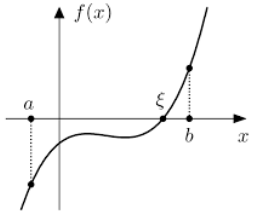


NULLSTELLENSUCHE

BISEKTION

Bisektion basiert auf der Idee der binären Suche. Dabei wird auf einem Intervall $]a, b[$ nach Nullstellen gesucht. Dabei wird das Intervall immer weiter halbiert, bis eine vordefinierte Intervallsgrösse erreicht ist, in der sich die Nullstelle befindet. Dafür müssen folgende Bedingungen erfüllt sein:

- $f(x)$ muss stetig sein.
- Im Intervall muss es zu einem Vorzeichenwechsel kommen.
  - $f(a) * f(b) < 0$
  - $\Rightarrow \exists a < c < b: f(c) = 0$
  - Garantiert Vorhandensein einer Nullstelle
  - Nur eine Nullstelle im Intervall
  - Doppelte Nullstellen werden vermieden



ALGORITHMUS

```
def bisekt(fun, a, b, tol=1e-5):
    if (fun(a)*fun(b) < 0):
        while abs(b-a) > tol:
            mid = (a+b)/2
            fmid = fun(mid)
            if fmid * fun(b) < 0:
                a = mid
            elif fmid * fun(b) > 0:
                b = mid
            else:
                a = mid
                break
        return mid
    else:
        return "f(a) and f(b) must have different signs"
```

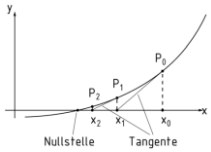
NEWTONVERFAHREN

Das Newtonverfahren verwendet Linearisierung, um iterativ Nullstellen zu finden.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Bedingungen:

- $f(x)$ muss stetig und differenzierbar sein.



ALGORITHMUS

```
def newton(fun, x0, lim=1e-5, max_iter=100):

    x = symbols('x')
    xi = [x0]
    fun_prime = fun.diff(x, 1)

    while True:
        delt = fun.subs(x, xi[-1])/fun_prime.subs(x, xi[-1])
        xi.append(xi[-1] - delt)
        if abs(xi[-2] - xi[-1]) < lim or len(xi) >= max_iter:
            break
    return xi[-1]
```

MEHRDIMENSIONALES NEWTON VERFAHREN

Das Newtonverfahren im mehrdimensionalen ist wie folgt definiert:

$$x_{k+1} = x_k - [\mathbf{J}(x_k)]^{-1} \cdot \mathbf{f}(x_k)$$

Anstelle der einfachen ersten Ableitung der Funktion f wird nun die Jacobi-Matrix $\mathbf{J}(x_1, x_2)$ der Vektorfunktion $\mathbf{f}$ berechnet.

$$\mathbf{J}(x_1, x_2) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}$$

Da die Berechnung der Inversen von $\mathbf{J}(x_1, x_2)$ numerisch instabil ist, berechnen wir stattdessen:

$$[\mathbf{J}(x_1, x_2)] \cdot \mathbf{d}_k = \mathbf{f}(x_k)$$

da

$$[\mathbf{J}(x_1, x_2)]^{-1}[\mathbf{J}(x_1, x_2)] \cdot d_k = [\mathbf{J}(x_1, x_2)]^{-1}\mathbf{f}(x_k)$$

$$d_k = [\mathbf{J}(x_1, x_2)]^{-1} \cdot \mathbf{f}(x_k)$$

BERECHNUNG

1. Berechnung der Jacobimatrix für die gegebenen Funktionen.
2. Einsetzen der Werte $(x_1, x_2)_k$ in Jacobimatrix
3. Einsetzen der Werte in Funktionsvektor.
4. Auslösen des Gleichungssystems nach d_k.

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}_{(x_1, x_2)} \cdot \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix}$$

5.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{k+1} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_k - \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

ALGORITHMUS

```
def mehrdim_newton(funcs: sp.Matrix, start: np.array, tol=1e-10, steps=False, array=False):

x, y = symbols('x y')
J = funcs.jacobian([x, y])
x_k = [np.array(start)]

while True:

    f_subs = funcs.subs({x: x_k[-1][0], y: x_k[-1][1]})
    J_subs = J.subs({x: x_k[-1][0], y: x_k[-1][1]})

    f_np = np.array(f_subs).astype(np.float64)
    J_np = np.array(J_subs).astype(np.float64)

    if np.linalg.norm(f_np) < tol:
        break

    # für d = [J(x_k)]^-1*f(x_k) -> J(x_k) * d = f(x_k)
    # somit muss nicht invertiert werden, da Inverse numerisch instabil
    d = np.linalg.solve(J_np, f_np)

    # x_k+1 = x_k - d
    x_k.append(np.array(x_k[-1] - d.flatten()))

# gibt Wert und Anzahl Berechnungsschritte zurück
if steps:
    return x_k[-1], len(x_k) - 1

# gibt alle Zwischenergebnisse aus
if array:
    return x_k

# gibt nur letzten Wert zurück
return x_k[-1]
```

LU-ZERLEGUNG

Die LU-Zerlegung ist eine Methode zum Lösen von linearen Gleichungssystemen:

$$Ax = b$$

- $A = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 3 \\ 3 & 5 & 4 \end{pmatrix}$

Dabei wird die Matrix A in zwei Matrizen L und U aufgeteilt. Beide sind Dreiecksmatrizen, L (lower) eine untere Dreiecks Matrix und U (upper) eine obere Dreiecksmatrix.

- L enthält die zum Lösen des Gleichungssystems durchgeführte Operationen.
  - $L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix}$
  - $L_{21} = -\frac{A_{21}}{A_{11}} = -\frac{2}{1}, \quad L_{31} = -\frac{A_{31}}{A_{11}} = -\frac{3}{1}$
- U enthält die Ergebnisse in der unteren Dreiecksmatrix, die aufgelöst werden kann.
  - $\begin{pmatrix} 1 & 2 & 1 \\ 0 & -2 & 1 \\ 0 & 0 & 0.5 \end{pmatrix}$

PLU-ZERLEGUNG

Beim Pivoting werden **Zeilen** vertauscht, so dass die grössten absoluten Elemente in jeder Spalte in die Diagonalposition kommen.

$$PA = LU$$

Dabei ist P eine Einheitsmatrix mit vertauschten Elementen um A zu transformieren.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 8 & 7 & 2 \\ 1 & 5 & 1 \end{pmatrix}, P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, PA = \begin{pmatrix} \mathbf{8} & 7 & 2 \\ 1 & \mathbf{5} & 1 \\ 1 & 2 & \mathbf{3} \end{pmatrix}$$

LÖSEN VON (P)LU

Wenn kein Pivot durchgeführt wurde, ist $P = I_n$.

1. $Ly = Pb$
   a.
2. $Ux = y$

Matrixnorm

Die Matrixnorm quantifiziert die Grösse einer Matrix. Matrixnormen sind besonders nützlich, um die Stabilität und Konvergenz von numerischen Algorithmen zu analysieren und um das Verhalten von Matrixoperationen zu verstehen.

$$\|A\| := \max_{v \neq 0} \frac{\|Av\|}{\|v\|} = \max_{\|v\|=1} \|Av\|$$

Fehler

$$\boldsymbol{r = b - A * \tilde{x}}$$

$$= A * x - A * \tilde{x} = A(x - \tilde{x})$$

- r: Residuum (numerischer Fehler)
- b: ursprünglicher Konstanten-Vektor
- $\tilde{x}$: Lösungsvektor des LGS
- A: Ursprüngliche Matrix

KONDITIONSZAHL

Die Konditionszahl misst die Sensitivität der Lösung gegenüber Änderungen in A oder b und gibt an, wie genau die Lösung ist. Eine hohe Konditionszahl (z.B. 1000), deutet auf potenzielle numerische Instabilitäten hin. Eine gut konditionierte Matrix hat eine KZ nahe 1.

$$\kappa(A) = \|A\| * \|A^{-1}\|$$

FEHLERSCHRANKE

Die Schranke für den relativen Fehler bietet eine obere Grenze für den Fehler in der Lösung, abhängig von der Konditionszahl und dem Residuum.

$$\kappa(A) * \frac{\|r\|}{\|b\|}$$

LINEARE REGRESSION

Die Parameter θ eines linearen Regressionsmodells kann durch die konvexe Kostenfunktion SSE (Sum of Squared Errors).

SSE:

$$f(\theta) = \|X\theta - y\|^2 = (X\theta - y)^2$$

$$\frac{\partial f}{\partial \theta} = 0$$

Ergibt:

$$\theta = (X^T X)^{-1} X^T y$$

Da die Berechnung der Inverse numerisch instabil ist, kann die Gleichung wie folgt umgestellt werden:

$$X^T X\theta = X^T y$$

Das ergibt ein Gleichungssystem, das durch Gauss bzw. LU gelöst werden kann. X^T

$$X^T X = A$$

$$X^T y = b$$

$$\theta = x$$

$$\Rightarrow Ax = b$$

DESIGNMATRIX

Die Designmatrix X enthält in der ersten Spalte die Polynome des höchsten Grades und ganz links nur Einsen, die für den Bias reserviert sind.

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_n x_n$$

$$\begin{pmatrix} 1 & x_{11} & \cdots x_{n1} \\ \vdots & \vdots & \ddots \ \vdots \\ 1 & x_{n1} & \cdots x_{nm} \end{pmatrix}$$

POLYNOMIELLE REGRESSION

Variablen können transformiert werden um die Daten besser zu fitten. Dies kann durch die Potenzierung erfolgen oder durch Variablentransformation → $\exp(x)$.

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{pmatrix}$$

POLYNOM-INTERPOLATION

Ein Polynom vom Grad n hat $n + 1$ Koeffizienten.

$$p(x) = a_n + x^n + a_{n-1} * x^{n-1} + \cdots + a_1 x + a_0$$

VANDERMONDE-MATRIX

Um eine Funktion mit gegebenen (x_n, y_n)-Paaren mit $x_1 \neq x_j$ zu interpolieren, können die Parameter des Polynoms $p(x)$ durch das lineare Gleichungssystem $Vp = y$ gelöst werden. In jeder Zeile ein x_i.

$$V = (x_1, x_2, \ldots, x_n) = \begin{bmatrix} 1 & x & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}$$

$$Vp = y$$

NACHTEILE

Konditionszahl ist bei dieser Methode hoch, was auf numerische Instabilität hindeutet, deswegen sollten andere Methoden bevorzugt werden. Ineffizient $\mathcal{O}(n^3)$.

LAGRANGE POLYNOM-INTERPOLATION

Um eine Funktion mit gegebenen (x_n, y_n)-Paaren mit $x_1 \neq x_j$ zu interpolieren, kann aus $n + 1$ Grundpolynomen $l_i(x)$ jeweils vom Grad n ein Interpolationspolynom konstruiert werden.

$$p(x) = \sum_{i=0}^{n-1} y_i * l_i(x)$$

Um l_i zu berechnen kann wie folgt vorgegangen werden:

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - x_j}{x_i - x_j}$$

NACHTEILE

- hohe Empfindlichkeit gegenüber Änderungen der Stützstellen.
- Numerische Instabilität
  - Je mit zunehmender Anzahl Stützstellen, erhöht sich der Grad des Polynoms.
- Ineffiziente Berechnung

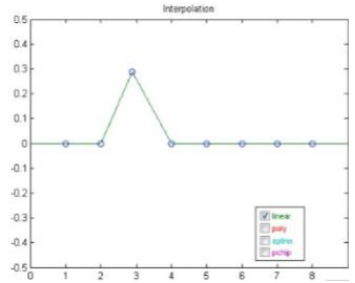
LINEARE SPLINE-INTERPOLATION

Die lineare Spline-Interpolation ist ein linearer Streckenzug zwischen jeweils zwei benachbarten Punkten $(x_1, y_1), (x_2, y_2)$. Dieser Ansatz liefert sehr ungenaue Ergebnisse.

$$s_i(x) = mx + b$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - \frac{y_2 - y_1}{x_2 - x_1}$$



KUBISCHE SPLINE-INTERPOLATION

Anstelle von linearem Spline, werden Polynome dritten Grades verwendet.

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

- $a_i = y_i$
- $b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6} * (s''_{i+1} + 2 * s''_j)$
- $c_i = \frac{s''_i}{2}$
- $d_i = \frac{s''_{i+1} - s''_i}{6h_i}$
- $h_i = x_{i+1} - x_1$

$$s''_i =$$

$$\begin{bmatrix} \mu_0 & \lambda_0 & & & & \\ \frac{h_0}{6} & \frac{h_0+h_1}{3} & \frac{h_1}{6} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \frac{h_{i-1}}{6} & \frac{h_{i-1}+h_i}{3} & \frac{h_i}{6} & \\ & & & \ddots & \ddots & \ddots \\ & & & & \lambda_n & \mu_n \end{bmatrix} \cdot \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_i \\ \vdots \\ M_n \end{bmatrix} = \begin{bmatrix} b_0 \\ \frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0} \\ \vdots \\ \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \\ \vdots \\ b_n \end{bmatrix}$$

NUMERISCHE INTEGRATION

Die numerische Quadratur nach Newton-Cotes ist ein Verfahren zur näherungsweisen Berechnung von Integralen. Die bekanntesten Newton-Cotes-Formeln sind:

- Rechteck-, Mittelpunktregel
- Trapez-Regel
- Simpson-Regel / Simpson 3/8-Regel

QUADRATURFORMEL

Die Quadraturformel ist eine Methode zur numerischen Integration, die verwendet wird, um das Integral von $f(x)$ über ein bestimmtes Intervall $[a, b]$ zu approximieren.

- Die Stützpunkte müssen gleichen Abstand haben.
- Jedes Integral eines Polynoms vom Grad $n-1$ kann perfekt approximiert werden, wenn n die Anzahl der Stützstellen ist.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

- x_i: sind die Stützstellen im Intervall $[a, b]$
- w_i: sind die Gewichte, die die Bedeutung der Funktionswerte $f(x_i)$ im Integral berücksichtigen.

GEWICHTE

Um die Gewichte zu berechnen, müssen wir die Lagrange-Polynome berechnen.

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Über diese Lagrange-Polynome muss integriert werden was dann folgende Gewichte ergibt:

$$w_i = \int_a^b l_i(x)dx$$

Das Ganze kann auch in Matrix-Schreibweise definiert werden. Wobei A eine Vandermonde-Matrix ist, welche die Stützstellen als x in jeder Zeile eingesetzt hat und b die Integrale der jeweiligen Zeilen aus den Stützstellen.

$$A = \overbrace{\begin{bmatrix} 1 & 1 & 1 \\ x & x & x \\ x^2 & x^2 & x^2 \end{bmatrix}}^{Spalten\ sind\ Stützpunkte} \quad b = \begin{bmatrix} \int_a^b 1 \\ \int_a^b x \\ \int_a^b x^2 \end{bmatrix}$$

$$Aw = b$$

Aus dem Skalarprodukt $w \cdot y$ ergibt sich das Integral.

$$w \cdot y \approx \int_a^b f(x)dx$$

SIMPSON-REGEL

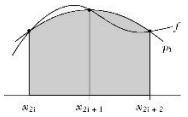
Simpson ergibt sich aus der Berechnung der Gewichte für **3** Stützpunkte:

Aus diese Lagrange-Polynome können integriert werden und ergeben dann folgende Gewichte:

1. $w_0 = \int_a^b l_i(x)dx = \frac{h}{3}$
2. $w_1 = \int_a^b l_i(x)dx = \frac{4h}{3}$
3. $w_2 = \int_a^b l_i(x)dx = \frac{h}{3}$

Wobei $h = \frac{b-a}{n}$ und n Anzahl Teilintervalle. Daraus folgt die geschlossene Form der Simpson-Regel:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4 * f\left(\frac{a+b}{2}\right) + f(b)\right]$$



SIMPSON 3/8-REGEL

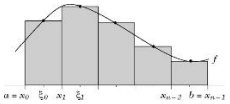
Simpson mit **4** Stützpunkten / Gewichte.

$$\int_a^b f(x)dx \approx \frac{b-a}{8}\left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\right]$$

MITTELPUNKTREGEL

Die Mittelpunktregel ergibt sich aus der Berechnung des Gewichts für **1** Stützpunkt / Gewicht.

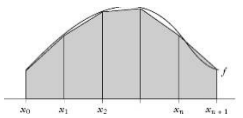
$$\int_a^b f(x)dx \approx (b-a) * f\left(\frac{a+b}{2}\right)$$



TRAPEZREGEL

Die Trapezregel ergibt sich aus der Berechnung 2 Stützpunkte / Gewichte.

$$\int_a^b f(x)dx \approx \frac{b-a}{2} * [f(a) + f(b)]$$



GEWÖHNLICHE DIFFERENTIALGLEICHUNGEN

Bei Differentialgleichungen kennen wir nur die Änderungsrate einer Funktion, also ihre Ableitung, aber nicht die Funktion selbst. Durch Auflösen einer Differentialgleichung erhalten wir die Funktion zur zugehörigen Änderungsrate (Ableitung).

$$\frac{\partial y}{\partial x} = f(x, y)$$

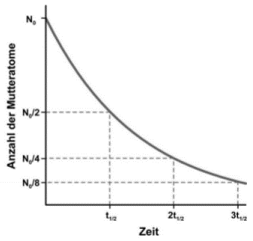
BEISPIEL – RADIOAKTIVER ZERFALL

Wir wissen, dass sich radioaktiver Zerfall wir folgt, verhält:

$$\frac{\partial y}{\partial t} = -\lambda * y(t), \qquad \lambda \in \mathbb{R}^+$$

Durch Auflösen dieser Differentialgleichung erhalten wir die Funktion, die dieses Phänomen beschreibt:

$$y(t) = C_1 e^{-\lambda t}$$

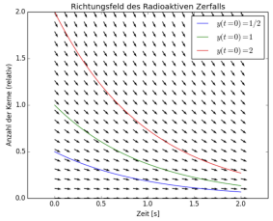


ANFANGSWERTPROBLEM

Da die Lösungen zu gewöhnlichen Differentialgleichungen nicht eindeutig sind und eine unendliche Anzahl von Funktionen zu unterschiedlichen Zeitpunkten beschreiben können, ist es notwendig, einen Anfangswert zum Zeitpunkt t_0 zu definieren, um eine eindeutige Lösung zu erhalten. Dieser Anfangswert wird als Anfangsbedingung bezeichnet.

BEISPIEL – RADIOAKTIVER ZERFALL

Bevor ein Anfangswert festgelegt ist, haben wir ∞ Funktionen, hier dargestellt als Richtungsfeld:



Definieren wir nun zum Zeitpunkt t_0 einen Startwert,

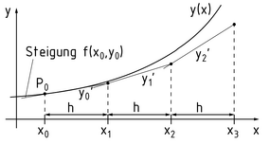
$$y(t) = C_1 e^{-\lambda t}, \qquad y(0) = 1, \qquad \lambda = 1$$

bekommen wir eine eindeutige Funktion (grün), da wir den Koeffizienten C_1 eindeutig bestimmen können:

$$C_1 = 1 \Rightarrow y(t) = e^{-t}$$

EXPLIZITES EULERVERFAHREN

Da nicht alle Differentialgleichungen analytisch gelöst werden können, gibt es numerische Verfahren, um die Lösung zu approximieren. Dabei wird ein Streckenzug mit Schrittweite h und der Änderungsrate $y'(t)$ berechnet.



Die Funktionswerte y werden wie folgt berechnet. Hier die Berechnung in rekursiver Schreibweise:

$$y_{k+1} = y_k + h * y'(t_k)$$

Dabei werden die Funktionswerte linear approximiert. Je grösser die Schrittweite, umso grösser der Fehler. Dieser Verhält sich additiv je weiter man geht.

BERECHNUNG

$$\begin{cases} y'(t) = f(y, t) \\ y(t_0) = y_0 \end{cases}$$

$$\begin{cases} t_k = t_0 + k * h \\ y_{k+1} = y_k + h * f(y_k, t_k) \end{cases}$$

ALGORITHMUS

```python
def euler_verfahren(dydt, y0, t0, T, h):
    n = int((T - t0) / h) + 1
    ts = [t0 + k * h for k in range(n)]
    ys = [y0]

    for k in range(1, len(ts)):
        y_k = ys[k - 1] + h * dydt(ts[k - 1], ys[k - 1])
        ys.append(y_k)

    return ts, ys
```

DIFFERENTIALGLEICHUNGSSYSTEME

DGL-Systeme bestehen aus mehreren Differentialgleichungen, die voneinander abhängen. Solche DGL-Systeme können durch numerische Verfahren wie Euler approximiert Werten. Dafür werden zwei Anfangswerte zum Zeitpunkt t_0 benötigt:

$$\begin{cases} y_1'(t) = f(t_k, y_1(t_k), y_2(t_k) \\ y_2'(t) = f(t_k, y_1(t_k), y_2(t_k) \end{cases}$$

$$\begin{cases} y_1(t_0) = y_{1,0} \\ y_2(t_0) = y_{2,0} \end{cases}$$

$$\begin{cases} y_1(t_{k+1}) = y_1(t_k) + h * f(t_k, y_1(t_k), y_2(t_k)) \\ y_2(t_{k+1}) = y_2(t_k) + h * f(t_k, y_1(t_k), y_2(t_k)) \end{cases}$$

BEISPIEL RÄUBER-BEUTE PROBLEM

Die Anzahl der Räuber hängt immer von der Beute ab.

$$\begin{cases} \frac{\partial B}{\partial t} = 2 * B - \alpha * R * B \\ \frac{\partial R}{\partial t} = -R + \alpha * R * B \end{cases}$$

DGL-SYSTEME FÜR DGLS HÖHERER ORDNUNG

Um numerisch Differentialgleichung höherer Ordnung zu lösen, ist es einfacher daraus ein DGL-System erster Ordnung zu machen.

$$y'' = 2y + t$$

Dafür können wir y wie folgt umformulieren:

1. $z_1 = y$
2. $z_2 = y'$

Daraus ergibt sich folgendes System:

$$\begin{cases} z_1' = z_2 \\ z_2' = 2z_1 + t \end{cases}$$

Dieses System kann einfach mit numerischen Verfahren gelöst, wie Euler oder Runge-Kutta werden.

➔ Durch Auflösen von z_1' nach z_1 und z_2' nach z_2 bekommen wir y und y'.

MEHRERE DGLS HÖHERER ORDNUNG

Aus dem Gleichungssystem von Differentialgleichungen höherer Ordnung,

$$\begin{cases} \dfrac{\partial^2 x}{\partial t^2} = -\dfrac{\partial x}{\partial t}\sqrt{\left(\dfrac{\partial x}{\partial t}\right)^2 + \left(\dfrac{\partial y}{\partial t}\right)^2} \\ \dfrac{\partial^2 y}{\partial t^2} = -\dfrac{\partial y}{\partial t}\sqrt{\left(\dfrac{\partial x}{\partial t}\right)^2 + \left(\dfrac{\partial y}{\partial t}\right)^2 - 10} \end{cases}$$

$$x(0) = 0, \frac{\partial x}{\partial t}(0) = 10, \qquad y(0) = 0, \frac{\partial y}{\partial t}(0) = 100$$

Machen wir:

$$\begin{cases} z_0 = x \\ z_1 = y \\ z_2 = x' \\ z_3 = y' \end{cases}$$

$$\begin{cases} z'_0 = z_2 \\ z'_1 = z_3 \\ z'_2 = -z_2\sqrt{z_2^2 + z_3^2} \\ z'_3 = -z_3\sqrt{z_2^2 + z_3^2} \end{cases}$$

Auch dieses System kann einfach mit numerischen Verfahren gelöst, wie Euler oder Runge-Kutta werden.

➔ Durch Auflösen von z_0' nach z_0 und z_1' nach z_1, z'_2 nach z_2 und z'_3 $nach$ $z_3$bekommen wir x, y, z' und y'.