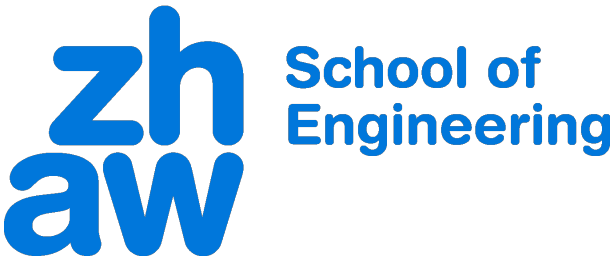
ZHAW Zurich University of Applied Sciences
Winterthur



Zusammenfassung NUM - LibRefManual Studienwochen 1-14

Written by: Severin Sprenger
June 17, 2026
Zf. NUM - LibRefManual SW 1-14

1 Lineare Gleichungssysteme (LGS) & Hilfsfunktionen

1.1 `forward_substitution(L, b)` & `backward_substitution(R, y)`

Konzept: Lösen von gestaffelten LGS, die aus Matrixzerlegungen (LU, Cholesky, QR) resultieren.
Mathematik: Vorwärtseinsetzen löst $Ly = b$ (wobei L eine untere Dreiecksmatrix ist) durch sukzessives Einsetzen von y_1 bis y_n. Rückwärtseinsetzen löst $Rx = y$ (wobei R eine obere Dreiecksmatrix ist) beginnend bei x_n bis x_1.
Verwendung: `y = forward_substitution(L, b)` gefolgt von `x = backward_substitution(R, y)`

1.2 `solve_tridiagonal(a, b, c, d)`

Konzept: Thomas-Algorithmus zur effizienten Lösung von $Ax = d$, wenn A eine Tridiagonalmatrix ist (z.B. bei 1D FDM oder Splines).
Mathematik: Führt eine optimierte Gauss-Elimination ohne Pivotisierung in $\mathcal{O}(n)$ Operationen durch. Es manipuliert die Diagonalenvektoren a, b, c direkt.
Verwendung: `x = solve_tridiagonal(sub_diag, main_diag, sup_diag, rhs)`

1.3 `cholesky_decomposition(A)` & `solve_cholesky(A, b)`

Konzept: Lösung von $Ax = b$ für symmetrisch positiv definite (spd) Matrizen (z.B. in der Ausgleichsrechnung).
Mathematik: Zerlegt $A = LL^T$. Danach wird $Ly = b$ und $L^T x = y$ gelöst.
Verwendung: `L = cholesky_decomposition(A)` oder direkt `x = solve_cholesky(A, b)`

1.4 `solve_qr(A, b)`

Konzept: Stabile Lösung von $Ax = b$ via QR-Zerlegung.
Mathematik: $A = QR$, wobei Q orthogonal ($Q^TQ = I$) und R obere Dreiecksmatrix ist. Aus $QRx = b$ folgt $Rx = Q^Tb$. Die rechte Seite wird mit Q^T multipliziert und dann durch Rückwärtseinsetzen gelöst.
Verwendung: `x = solve_qr(A, b)`

2 Nichtlineare Gleichungssysteme

2.1 `newton_1d(f, df, x0, tol, max_iter)`

Konzept: Iterative Nullstellensuche für skalare Funktionen $f(x) = 0$.
Mathematik: Tangentenverfahren. Iterationsvorschrift: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.
Verwendung: `root = newton_1d(f_func, df_func, 1.0)`

2.2 `newton_multidim(F, J, x0, tol, max_iter)`

Konzept: Finden von Wurzeln mehrdimensionaler nichtlinearer Systeme $\mathbf{F}(\mathbf{x}) = 0$.
Mathematik: Berechnet in jedem Schritt die Jacobi-Matrix $J = D\mathbf{F}(\mathbf{x}_k)$. Löst das lineare Gleichungssystem $J\Delta\mathbf{x} = -\mathbf{F}(\mathbf{x}_k)$ (hier via QR-Zerlegung) und updatet $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}$.
Verwendung: `root_vec = newton_multidim(F_vec_func, Jacobian_func, x0_vec)`

3 Ausgleichsrechnung (Least Squares)

3.1 `linear_least_squares(A, y)`

Konzept: Lösung überbestimmter linearer Systeme zur Minimierung von $\|Ax - y\|_2^2$.
Mathematik: Löst die Normalengleichungen $A^T Ax = A^T y$. Da $A^T A$ spd ist (sofern A vollen Spaltenrang hat), wird die Cholesky-Zerlegung zur Lösung herangezogen.
Verwendung: `params = linear_least_squares(A, y_data)`

3.2 `gauss_newton(r, dr, x0, tol, max_iter)`

Konzept: Fitten von nichtlinearen Modellfunktionen an Datenpunkte.
Mathematik: Minimiert die Summe der quadratischen Residuen. In jeder Iteration wird die Jacobi-Matrix der Residuen J ausgewertet und das LGS $J^T J\Delta\mathbf{x} = -J^T\mathbf{r}(\mathbf{x}_k)$ gelöst (überwiegend mit Cholesky, Fallback auf QR bei schlechter Konditionierung).
Verwendung: `opt_params = gauss_newton(res_func, jac_res_func, initial_guess)`

4 Gewöhnliche Differentialgleichungen (DGL / AWP)

4.1 `explicit_euler`, `heun_method`, `runge_kutta_4`, `midpoint_rule`

Konzept: Numerische Integration von Anfangswertproblemen $y' = f(t, y)$ mit $y(t_0) = y_0$.
Mathematik:

- **Euler (p=1):** Lineare Extrapolation $y_{n+1} = y_n + hf(t_n, y_n)$.
- **Mittelpunktregel (p=2):** Nutzt halben Euler-Schritt zur Steigungsauswertung in der Intervallmitte.
- **Heun (p=2):** Mittelung der Steigungen am Anfang und Ende des Intervalls.
- **RK4 (p=4):** Gewichtete Mittelung von 4 Steigungen (Anfang, 2x Mitte, Ende).

Verwendung: `t_vec, y_vec = explicit_euler(f, t0, y0, h, n_steps)` (funktioniert auch für Vektoren y_0 bei DGL-Systemen).

4.2 `implicit_euler_scalar(f, df_dy, t0, y0, h, n_steps)`

Konzept: A-stabiles Verfahren für steife Differentialgleichungen (nur skalar implementiert).
Mathematik: $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$. Da y_{n+1} beidseitig auftritt, wird in jedem Zeitschritt intern das eindimensionale Newton-Verfahren gestartet, um die Nullstelle der Funktion $F(y_{next}) = y_{next} - y_n - hf(t_{n+1}, y_{next})$ zu finden.
Verwendung: `t_vec, y_vec = implicit_euler_scalar(f, df_dy, t0, y0, h, n_steps)`

5 Partielle Differentialgleichungen (FDM)

5.1 `fdm_1d_poisson(f, a, b, alpha, beta, N)`

Konzept: Lösen des 1D-Randwertproblems $-u''(x) = f(x)$ mit Dirichlet-Randbedingungen $u(a) = \alpha, u(b) = \beta$.
Mathematik: Diskretisiert die zweite Ableitung mit zentralen Differenzen $\mathcal{O}(h^2)$. Baut das System $\frac{1}{h^2}(-u_{i-1} + 2u_i - u_{i+1}) = f_i$ auf. Dies resultiert in einer Tridiagonalmatrix, die über den Thomas-Algorithmus effizient gelöst wird.
Verwendung: `x_grid, u_sol = fdm_1d_poisson(f_func, 0.0, 1.0, u_a, u_b, 100)`

5.2 `create_2d_poisson_matrix(Nx, Ny)`

Konzept: Aufbau der Matrix für die 2D Poisson-Gleichung $\Delta u = f$.
Mathematik: Verwendet den 5-Punkte-Stern $(4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1})$. Erzeugt die charakteristische schwach besetzte Block-Tridiagonalmatrix. Das entstehende LGS $Ax = b$ muss dann gelöst werden.
Verwendung: `A = create_2d_poisson_matrix(Nx, Ny)` gefolgt von `solve_cholesky(A, f_vec)`.

6 Spline-Interpolation

6.1 `cubic_spline_natural(x, y)` & `evaluate_spline(...)`

Konzept: Stabile, stückweise polynomiale Interpolation von Datenpunkten ohne Oszillationen (wie bei Runge-Polynomen höherer Ordnung).
Mathematik: Löst das Momentengleichungssystem (Tridiagonalsystem der 2. Ableitungen M_i) unter der "natürlichen" Annahme $M_0 = M_n = 0$. Daraus werden die Koeffizienten a_i, b_i, c_i, d_i für jedes Intervall $[x_i, x_{i+1}]$ berechnet.
Verwendung: `a, b, c, d = cubic_spline_natural(x, y)` und zur Auswertung `y_val = evaluate_spline(x_query, x, a, b, c, d)`.

6.2 `levenberg_marquardt(r, dr, x0, tol, max_iter, initial_lam, factor)`

Konzept: Robuste Optimierung für nichtlineare Ausgleichsprobleme. Dient als sicherer Fallback, wenn Gauss-Newton aufgrund von schlecht konditionierten Jacobi-Matrizen divergiert oder oszilliert.
Mathematik: Kombiniert Gauss-Newton (Konvergenz in der Nähe des Minimums) und Gradientenabstieg (Stabilität weit weg vom Minimum). Löst das modifizierte Normalengleichungssystem $(J^T J + \lambda I)\Delta \mathbf{x} = -J^T \mathbf{r}(\mathbf{x}_k)$. Der Dämpfungsparameter λ wird dynamisch angepasst: Verkleinert (Faktor), wenn der Schritt den Fehler reduziert (Richtung Gauss-Newton); vergrössert, wenn der Schritt den Fehler verschlechtert (Richtung Gradientenabstieg).
Verwendung: `opt_params = levenberg_marquardt(res_func, jac_res_func, initial_guess)`