Höhere Mathematik

Jil Zerndt, Lucien Perret
January 2025

Python Implementationen

Hilfsfunktionen

Matrixoperationen

```python
def matrix_vector_mult(A, v): # Matrix-Vektor Mult.
    n = len(A)
    result = [0] * n
    for i in range(n):
        result[i] = sum(A[i][j] * v[j] for j in
            range(n))
    return result

def matrix_mult(A, B): # Matrix-Matrix Multiplikation
    m, n = len(A), len(B[0])
    p = len(B)
    C = [[0.0] * n for _ in range(m)]
    for i in range(m):
        for j in range(n):
            C[i][j] = sum(A[i][k] * B[k][j] for k in
                range(p))
    return C

def transpose(A): # Matrix transponieren
    n = len(A)
    return [[A[j][i] for j in range(n)] for i in
        range(n)]

def vector_norm(v): # Euklidische Norm eines Vektors
    return sum(x*x for x in v) ** 0.5

def normalize_vector(v): # Vektor auf L. 1 normieren
    norm = vector_norm(v)
    return [x/norm for x in v] if norm > 0 else v

def copy_matrix(A): # Tiefe Kopie einer Matrix
    return [[A[i][j] for j in range(len(A[0]))] for i
        in range(len(A))]
```

is_diagonally_dominant Diagonaldominanz prüfen

```python
def is_diagonally_dominant(A):
    n = len(A)
    for i in range(n):
        if abs(A[i][i]) <= sum(abs(A[i][j]) for j in
            range(n) if j != i):
            return False
    return True
```

convergence_check Konvergenzkriterien

```python
def convergence_check(x_new, x_old, f_new, f_old,
    tol=1e-6):
    # Absoluter Fehler im Funktionswert
    if abs(f_new) < tol:
        return True, "Funktionswert < tol"
    # Relative Aenderung der x-Werte
    if abs(x_new - x_old) < tol * (1 + abs(x_new)):
        return True, "Relative Aenderung < tol"
    # Relative Aenderung der Funktionswerte
    if abs(f_new - f_old) < tol * (1 + abs(f_new)):
        return True, "Funktionsaenderung < tol"
    # Divergenzcheck
    if abs(f_new) > 2 * abs(f_old):
        return False, "Divergenz detektiert"

    return False, "Noch nicht konvergiert"
```

error_estimate Fehlerabschätzung durch Vorzeichenwechsel

```python
def error_estimate(f, x, eps=1e-5):
    fx_left = f(x - eps)
    fx_right = f(x + eps)

    if fx_left * fx_right < 0:
        return eps  # Nullstelle liegt in (x-eps,
            x+eps)
    return None
```

Numerische Lösung von Nullstellenproblemen

root_finder_with_error Nullstellensuche mit Fehlerabschaetzung

```python
def root_finder_with_error(f, x0, tol=1e-6,
    max_iter=100):
    x_old = x0
    f_old = f(x_old)

    for i in range(max_iter):
        # Iterationsschritt (hier Newton als Beispiel)
        x_new = x_old - f_old/derivative(f, x_old)
        f_new = f(x_new)

        # Pruefe Konvergenzkriterien
        converged, reason = convergence_check(
            x_new, x_old, f_new, f_old, tol)

        if converged:
            # Schaetze finalen Fehler
            error = error_estimate(f, x_new, tol)
            return {
                'root': x_new,
                'iterations': i+1,
                'error_bound': error,
                'convergence_reason': reason
            }

        x_old, f_old = x_new, f_new

    raise ValueError(f"Keine Konvergenz nach
        {max_iter} Iterationen")
```

fixed_point_it Fixpunktiteration

```python
def fixed_point_it(f, x0, tol=1e-6, max_it=100):
    x = x0
    for i in range(max_it):
        x_new = f(x)
        if abs(x_new - x) < tol:
            return x_new, i+1
        x = x_new
    raise ValueError("Keine Konvergenz")

# Optimierte Version mit Fehlerschaetzung
def fixed_point_it_opt(f, x0, tol=1e-6, max_it=100):
    x = x0
    alpha = None  # Schaetzung fuer Lipschitz-Konstante
    for i in range(max_iter):
        x_new = f(x)
        dx = abs(x_new - x)
        # Lipschitz-Konstante schaetzen
        if i > 0 and dx > 0:
            alpha_new = dx / dx_old
            if alpha is None or alpha_new > alpha:
                alpha = alpha_new
        # A-posteriori Fehlerabschaetzung
        if alpha is not None and alpha < 1:
            error = alpha * dx / (1 - alpha)
            if error < tol:
                return x_new, i+1
        x = x_new
        dx_old = dx
    raise ValueError("Keine Konvergenz")
```

newton Newton-Verfahren mit Konvergenzprüfung

```python
def newton(f, df, x0, tol=1e-6, max_iter=100):
    x = x0
    fx = f(x)

    for i in range(max_iter):
        dfx = df(x)
        if abs(dfx) < 1e-10:
            raise ValueError("Ableitung nahe Null")

        dx = fx/dfx
        x_new = x - dx
        fx_new = f(x_new)

        # Konvergenzpruefung
        converged, reason = convergence_check(
            x_new, x, fx_new, fx, tol)
        if converged:
            return {
                'root': x_new,
                'iterations': i+1,
                'residual': abs(fx_new),
                'convergence_reason': reason
            }

        if abs(fx_new) >= abs(fx):
            raise ValueError("Divergenz detektiert")

        x, fx = x_new, fx_new

    raise ValueError("Keine Konvergenz")
```

secant Sekantenverfahren mit Konvergenzprüfung

```python
def secant(f, x0, x1, tol=1e-6, max_iter=100):
    fx0 = f(x0)
    fx1 = f(x1)

    # Stelle mit kleinerem f-Wert als x1
    if abs(fx0) < abs(fx1):
        x0, x1 = x1, x0
        fx0, fx1 = fx1, fx0

    for i in range(max_iter):
        if abs(fx1) < tol:
            return x1, i+1
        if fx1 == fx0:
            raise ValueError("Division durch Null")
        # Sekanten-Schritt
        d = fx1 * (x1 - x0)/(fx1 - fx0)
        x2 = x1 - d

        # Konvergenzpruefungen
        if abs(d) < tol * (1 + abs(x1)): # Relative
             Aenderung
            return {
                'root': x2,
                'iterations': i+1,
                'residual': abs(f(x2))
            }
        fx2 = f(x2)
        if abs(fx2) >= abs(fx1): # Divergenzcheck
            if i == 0:
                raise ValueError("Schlechte
                    Startwerte")
            return {
                'root': x1,
                'iterations': i+1,
                'residual': abs(fx1)
            }

        x0, x1 = x1, x2
        fx0, fx1 = fx1, fx2

    raise ValueError("Keine Konvergenz")
```

Nullstellenverfahren - Praktisches Vorgehen

1. Voraussetzungen prüfen:
   - Existiert Nullstelle? (z.B. Vorzeichenwechsel)
   - Sind Startwerte geeignet?
   - Ist Funktion ausreichend glatt? (für Newton)
2. Verfahren wählen:
   - Newton: Wenn Ableitung verfügbar und Startwert nahe Lösung
   - Sekanten: Wenn keine Ableitung aber zwei Startwerte nahe Lösung
   - Fixpunkt: Wenn Funktion kontraktiv
3. Implementierung:
   - Konvergenzkriterien definieren
   - Maximale Iterationszahl festlegen
   - Fehlerabschätzung einbauen
   - Divergenzschutz implementieren
4. Auswertung:
   - Konvergenzverhalten prüfen
   - Fehler abschätzen
   - Ergebnis validieren

Numerische Lösung linearer Gleichungssysteme

gauss_elimination Gauss-Elimination mit Spaltenpivotisierung

```python
def gauss_elimination(A, b, tol=1e-10):
    n = len(A)
    M = copy_matrix(A) # Tiefe Kopie von A und b
    x = [0] * n
    b = b.copy()

    # Vorwaertselimination mit Pivotisierung
    for i in range(n):
        # Pivotisierung
        pivot_row = i
        for j in range(i+1, n):
            if abs(M[j][i]) > abs(M[pivot_row][i]):
                pivot_row = j
        if pivot_row != i:
            M[i], M[pivot_row] = M[pivot_row], M[i]
            b[i], b[pivot_row] = b[pivot_row], b[i]
        # Pruefe auf singulaere Matrix
        if abs(M[i][i]) < tol:
            raise ValueError("Matrix (fast) singulaer")
        # Elimination
        for j in range(i+1, n):
            factor = M[j][i] / M[i][i]
            for k in range(i, n):
                M[j][k] -= factor * M[i][k]
            b[j] -= factor * b[i]

    # Rueckwaertssubstitution
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - sum(M[i][j] * x[j]
                for j in range(i+1, n))) / M[i][i]
    return {
        'solution': x, 'matrix': M,
        'condition': estimate_condition(A)
    }
```

lr_decomposition LR-Zerlegung mit Zeilenvertauschung

```python
def lr_decomposition(A, tol=1e-10):
    n = len(A)

    # Initialisiere L, R und P
    R = copy_matrix(A)
    L = [[1.0 if i == j else 0.0 for j in range(n)]
         for i in range(n)]
    P = [[1.0 if i == j else 0.0 for j in range(n)]
         for i in range(n)]

    for k in range(n-1):
        # Pivotisierung
        pivot = k
        for i in range(k+1, n):
            if abs(R[i][k]) > abs(R[pivot][k]):
                pivot = i

        if abs(R[pivot][k]) < tol:
            raise ValueError("Matrix (fast) singulaer")

        # Zeilenvertauschung falls noetig
        if pivot != k:
            R[k], R[pivot] = R[pivot], R[k]

            # L und P anpassen fuer Zeilen < k
            for j in range(k):
                L[k][j], L[pivot][j] = L[pivot][j],
                    L[k][j]
            P[k], P[pivot] = P[pivot], P[k]

        # Elimination
        for i in range(k+1, n):
            factor = R[i][k] / R[k][k]
            L[i][k] = factor
            for j in range(k, n):
                R[i][j] -= factor * R[k][j]

    return {
        'L': L,
        'R': R,
        'P': P
    }
```

solve_lr LGS mit LR-Zerlegung lösen

```python
def solve_lr(L, R, P, b):
    n = len(L)
    # Pb berechnen
    pb = matrix_vector_mult(P, b)

    # Vorwaertseinsetzen Ly = Pb
    y = [0] * n
    for i in range(n):
        y[i] = pb[i] - sum(L[i][j] * y[j] for j in
            range(i))

    # Rueckwaertseinsetzen Rx = y
    x = [0] * n
    for i in range(n-1, -1, -1):
        x[i] = (y[i] - sum(R[i][j] * x[j]
                for j in range(i+1, n))) / R[i][i]

    return x
```

householder_vector Householder-Vektor zu x berechnen

```python
def householder_vector(x):
    n = len(x)
    v = x.copy()
    sigma = sum(v[i]*v[i] for i in range(1, n))

    if sigma == 0 and x[0] >= 0:
        return [0] * n, 0
    elif sigma == 0 and x[0] < 0:
        return [2] + [0]*(n-1), -2
    mu = (x[0]*x[0] + sigma)**0.5
    if x[0] <= 0:
        v[0] = x[0] - mu
    else:
        v[0] = -sigma/(x[0] + mu)
    beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
    return normalize_vector(v), beta
```

qr_decomposition QR-Zerlegung mittels Householder-Transformationen

```python
def qr_decomposition(A):
    m = len(A)
    n = len(A[0])
    R = copy_matrix(A)
    Q = [[1.0 if i == j else 0.0 for j in range(m)]
         for i in range(m)]

    for k in range(min(m-1, n)):
        # Extrahiere k-te Spalte ab k-ter Zeile
        x = [R[i][k] for i in range(k, m)]
        # Berechne Householder-Transformation
        v, beta = householder_vector(x)

        # Wende Householder auf R an
        for j in range(k, n):
            # w = beta * (v^T * R_j)
            w = beta * sum(v[i-k]*R[i][j]
                           for i in range(k, m))
            for i in range(k, m): # Update R
                R[i][j] -= v[i-k] * w
        for j in range(m): # Update Q
            w = beta * sum(v[i-k]*Q[j][i+k]
                           for i in range(len(v)))
            for i in range(len(v)):
                Q[j][k+i] -= v[i] * w

    Q = transpose(Q) # Transponiere Q am Ende
    return {
        'Q': Q,'R': R
    }
```

solve_qr Lösen von QRx = b

```python
def solve_qr(Q, R, b):
    # Berechne Q^T * b
    y = matrix_vector_mult(transpose(Q), b)
    # Rueckwaertseinsetzen
    n = len(R)
    x = [0] * n
    for i in range(n-1, -1, -1):
        x[i] = (y[i] - sum(R[i][j] * x[j] for j in
             range(i+1, n))) / R[i][i]
    return x
```

Iterative Löser für lineare Gleichungssysteme

jacobi_method Jacobi-Verfahren

```python
def jacobi_method(A, b, tol=1e-6, max_iter=100):
    n = len(A)
    # Pruefe Diagonaldominanz
    if not is_diagonally_dominant(A):
        print("Warnung: Matrix nicht diagonaldominant")
    # Initialisiere mit Nullvektor
    x = [0.0] * n
    iterations = []
    residuals = []
    for iter in range(max_iter):
        x_new = [0.0] * n
        # Jacobi-Iteration
        for i in range(n):
            sum_term = sum(A[i][j] * x[j]
                           for j in range(n) if j != i)
            x_new[i] = (b[i] - sum_term) / A[i][i]
        # Berechne Residuum
        res = max(abs(sum(A[i][j] * x_new[j]
                  for j in range(n)) - b[i])
                  for i in range(n))
        # Konvergenzcheck
        diff = max(abs(x_new[i] - x[i]) for i in
             range(n))

        iterations.append(x_new.copy())
        residuals.append(res)
        if diff < tol:
            return {
                'solution': x_new,
                'iterations': iterations,
                'residuals': residuals,
                'iteration_count': iter + 1
            }
        x = x_new.copy()
    raise ValueError("Keine Konvergenz nach
         {max_iter}, Iterationen\nLetztes Residuum:
         {res}")
```

gauss_seidel_method Gauss-Seidel-Verfahren

```python
def gauss_seidel_method(A, b, tol=1e-6, max_iter=100):
    n = len(A)
    iterations = []
    residuals = []

    # Pruefe Diagonaldominanz
    if not is_diagonally_dominant(A):
        print("Warnung: Matrix nicht diagonaldominant")
    x = [0.0] * n
    # Gauss-Seidel-Iteration
    for iter in range(max_iter):
        x_old = x.copy()
        for i in range(n):
            sum1 = sum(A[i][j] * x[j] for j in
                 range(i))
            sum2 = sum(A[i][j] * x_old[j]
                       for j in range(i+1, n))
            x[i] = (b[i] - sum1 - sum2) / A[i][i]
        # Berechne Residuum und relative Aenderung
        res = max(abs(sum(A[i][j] * x[j]
                  for j in range(n)) - b[i])
                  for i in range(n))
        diff = max(abs(x[i] - x_old[i]) for i in
             range(n))

        iterations.append(x.copy())
        residuals.append(res)
        if diff < tol:
            return {
                'solution': x,
                'iterations': iterations,
                'residuals': residuals,
                'iteration_count': iter + 1
            }

    raise ValueError(f"Keine Konvergenz nach
         {max_iter} "
                     f"Iterationen\nLetztes Residuum:
                         {res}")
```

analyze_convergence Konvergenzanalyse

```python
def analyze_convergence(method_name, iterations,
     residuals):
    """Analysiert Konvergenzverhalten eines iterativen
         Verfahrens"""
    n = len(residuals)
    if n < 2:
        return {
            'method': method_name,
            'converged': False,
            'reason': 'Zu wenige Iterationen'
        }

    # Schaetze Konvergenzrate
    rates = [abs(residuals[i]/residuals[i-1])
             for i in range(1, n)]
    avg_rate = sum(rates) / len(rates)

    # Schaetze asymptotische Konvergenzrate
    asymp_rate = rates[-1] if rates else None

    # Berechne empirische Konvergenzordnung
    if n > 2:
        q = [abs(log(residuals[i+1]/residuals[i]) /
                 log(residuals[i]/residuals[i-1]))
             for i in range(n-2)]
        avg_order = sum(q) / len(q) if q else None
    else:
        avg_order = None

    return {
        'method': method_name,
        'converged': residuals[-1] < residuals[0],
        'iterations': n,
        'final_residual': residuals[-1],
        'avg_rate': avg_rate,
        'asymp_rate': asymp_rate,
        'conv_order': avg_order
    }
```

Implementation iterativer Verfahren

1. Wählen Sie Startvektor $x^{(0)}$
2. Wählen Sie Abbruchkriterien:
   - Maximale Iterationszahl k_{max}
   - Toleranz ϵ für Änderung $\|x^{(k+1)} - x^{(k)}\|$
   - Toleranz für Residuum $\|Ax^{(k)} - b\|$
3. Führen Sie Iteration durch bis Kriterien erfüllt

Eigenwerte und Eigenvektoren

complex_operations Komplexe Zahlen

```python
def complex_operations(z1, z2):
    """Grundlegende Operationen mit komplexen Zahlen."""
    # Basisfunktionen
    def to_polar(z):
        r = (z.real**2 + z.imag**2)**0.5
        phi = math.atan2(z.imag, z.real)
        return r, phi

    def from_polar(r, phi):
        return r * (math.cos(phi) + 1j*math.sin(phi))

    try:
        # Addition und Subtraktion
        z_add = z1 + z2
        z_sub = z1 - z2
        # Multiplikation und Division
        z_mul = z1 * z2
        z_div = z1 / z2 if z2 != 0 else None
        # Polarform
        r1, phi1 = to_polar(z1)
        r2, phi2 = to_polar(z2)
        # Exponentialform
        z1_exp = from_polar(r1, phi1)
        z2_exp = from_polar(r2, phi2)

        return {
            'addition': z_add,
            'subtraktion': z_sub,
            'multiplikation': z_mul,
            'division': z_div,
            'polar_z1': (r1, phi1),
            'polar_z2': (r2, phi2)
        }

    except Exception as e:
        print(f"Fehler bei Berechnung: {e}")
        return None
```

Determinante

```python
def det_2x2(matrix):
    return matrix[0][0]*matrix[1][1] -
        matrix[0][1]*matrix[1][0]

def det_3x3(matrix):
    det = 0
    # Entwicklung nach erster Zeile
    for i in range(3):
        minor = []
        for j in range(1,3):
            row = []
            for k in range(3):
                if k != i:
                    row.append(matrix[j][k])
            minor.append(row)
        det += ((-1)**i) * matrix[0][i] *
            det_2x2(minor)
    return det
```

characteristic_polynomial

Charakteristisches Polynom einer 2x2 oder 3x3 Matrix

```python
def characteristic_polynomial(A):
    n = len(A)
    if n == 2:
        a, d = A[0][0], A[1][1]
        # det(A - lambda*I) = lambda^2 - tr(A)*lambda
            + det(A)
        return [1, -(a + d), det_2x2(A)]
    elif n == 3:
        # Hier nur Koeffizienten, keine
            Polynomauswertung
        trace = sum(A[i][i] for i in range(3))
        det = det_3x3(A)
        return [1, -trace, None, det]  # Mittlerer
            Koeff. kompliziert
    else:
        raise ValueError("Nur fuer 2x2 oder 3x3
            Matrizen")
```

find_eigenvalues_2x2 Eigenwerte einer 2x2 Matrix

```python
def find_eigenvalues_2x2(A, tol=1e-10):
    coeff = characteristic_polynomial(A)
    # Quadratische Formel
    p, q = -coeff[1], coeff[2]
    disc = p*p/4 - q
    if abs(disc) < tol:  # Doppelte Eigenwerte
        return [-p/2, -p/2]
    elif disc > 0:  # Reelle Eigenwerte
        root = disc**0.5
        return [-p/2 + root, -p/2 - root]
    else:  # Komplexe Eigenwerte
        root = (-disc)**0.5
        return [-p/2 + 1j*root, -p/2 - 1j*root]
```

find_eigenvector Eigenvektor zu gegebenem Eigenwert

```python
def find_eigenvector(A, eigenval, tol=1e-10):
    n = len(A)
    # A - lambda*I
    M = [[A[i][j] - (eigenval if i==j else 0)
          for j in range(n)] for i in range(n)]

    # Loese homogenes System (A - lambda*I)x = 0
    # Nehme eine Komponente als 1 an
    for i in range(n):
        if abs(M[i][i]) > tol:
            vec = [0] * n
            vec[i] = 1
            for j in range(n):
                if j != i:
                    s = sum(-M[j][k]*vec[k] for k in
                        range(n) if k != j)
                    if abs(M[j][j]) > tol:
                        vec[j] = s/M[j][j]
            return normalize_vector(vec)

    raise ValueError("Kein Eigenvektor gefunden")
```

power_iteration Von-Mises-Iteration für grössten Eigenwert

```python
def power_iteration(A, tol=1e-10, max_iter=100):
    n = len(A)
    # Starte mit Einheitsvektor
    v = normalize_vector([1] + [0]*(n-1))
    lambda_old = 0

    for _ in range(max_iter):
        # Matrix-Vektor-Multiplikation
        w = matrix_vector_mult(A, v)
        # Normiere neuen Vektor
        v = normalize_vector(w)

        # Rayleigh-Quotient
        lambda_k = sum(v[i] * A[i][j] * v[j]
                       for i in range(n)
                       for j in range(n))

        if abs(lambda_k - lambda_old) < tol:
            return {
                'eigenvalue': lambda_k,
                'eigenvector': v
            }

        lambda_old = lambda_k

    raise ValueError("Keine Konvergenz")
```

inverse_iteration Inverse Iteration für Eigenwert nahe μ

```python
def inverse_iteration(A, mu, tol=1e-10, max_iter=100):
    n = len(A)
    # Starte mit Einheitsvektor
    v = normalize_vector([1] + [0]*(n-1))
    lambda_old = 0

    # (A - mu*I) berechnen
    M = [[A[i][j] - (mu if i==j else 0)
          for j in range(n)] for i in range(n)]

    for _ in range(max_iter):
        # Loese (A - mu*I)w = v
        w = gauss_elimination(M, v)['solution']
        # Normiere neuen Vektor
        v = normalize_vector(w)

        # Rayleigh-Quotient
        lambda_k = sum(v[i] * A[i][j] * v[j]
                       for i in range(n)
                       for j in range(n))

        if abs(lambda_k - lambda_old) < tol:
            return {
                'eigenvalue': lambda_k,
                'eigenvector': v
            }

        lambda_old = lambda_k

    raise ValueError("Keine Konvergenz")
```

qr_algorithm QR-Algorithmus für alle Eigenwerte

```
def qr_algorithm(A, tol=1e-10, max_iter=100):
    n = len(A)
    A_k = copy_matrix(A)

    for k in range(max_iter):
        # QR-Zerlegung
        qr = qr_decomposition(A_k)
        Q, R = qr['Q'], qr['R']

        # Neue Iteration A_(k+1) = RQ
        A_k = matrix_mult(R, Q)

        # Pruefe ob Diagonalelemente konvergiert sind
        if all(abs(A_k[i][j]) < tol
               for i in range(1, n)
               for j in range(i)):
            return {
                'eigenvalues': [A_k[i][i] for i in
                    range(n)],
                'iterations': k+1,
                'final_matrix': A_k
            }

    raise ValueError("Keine Konvergenz")

def deflation(A, eigenval, eigenvec):
    """Deflation nach Hotelling"""
    n = len(A)
    v = normalize_vector(eigenvec)

    # Berechne A - lambda*vv^T
    vvt = [[v[i]*v[j] for j in range(n)] for i in
        range(n)]
    return [[A[i][j] - eigenval*vvt[i][j]
             for j in range(n)] for i in range(n)]
```