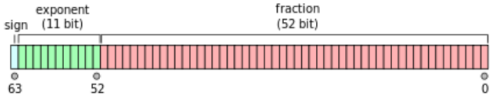
Gleitkommazahlen / Ausgabeformate

Python default: 64-bit Gleitkommazahlen



Genauigkeit ca. 16 Dezimalstellen in der Mantisse, entsprechend einer Rundung auf das letzte Bit der Mantisse.

Genauigkeit von numpy kann eingestellt werden mit:
In: np.set_printoptions(precision=4)
In: np.array([1.2345667e-5])
Out: array([1.2346e-05])

Allgemeine Formatierung:
In: np.format_float_scientific(1372.4828459e-4, precision=4)
Out: '1.3725e-01'

Konsequenzen endlicher Arithmetik
- Darstellungsfehler in Daten: math.pi
- Binäre Darstellung von $0.1 = \frac{1}{10} = \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \ldots : 0.000110011\ldots_{base=2}$, d.h. eine endliche Darstellung hat Fehler
- Rundungsfehler $\left(\frac{1}{3}\right) 3 - \frac{1}{27}$
- Overflow 21024
- Addition und Subtraktion 1+1e-20
- Erklärung zur Addition und Subtraktion: 1 + 1e − 20 = 1e0 + 0.00000000000000000001e0
  -> Mantissen werden addiert, nachdem der kleinere Exponent angeglichen wurde auf den grösseren (durch Kommaverschiebung)

➔ Rundung der Mantissen erfolgt unabhängig vom Exponenten -> folglich nimmt der Abstand der Fliesskommazahlen mit dem Exponenten zu
➔ Abstand zweier benachbarten Gleitkommazahlen mit np.finfo()
➔ Kleine Fehler können bei grosser Anzahl Ausführungen zu grossen Abweichungen führen

Python als Taschenrechner

```
import math
import numpy as np
from matplotlib import pyplot as plt

Potenzieren      2**5
Wurzel           math.sqrt() /
                 math.pow()
```

Vektoren und Matrizen

```
np.array([1,2,3,4])  Out: 1 Zeile
np.array([[-1], [2], [3], [4]]) Out: 4 Zeilen
Bei Rechnen mit Matrizen:
np.array([1],dtype=np.float64) # safe

Transponieren    x.T
Skalarprodukt    np.dot(x,x)
```

Matrixbildende Funktionen:
- eye() Einheitsmatrix
- zeros() Nullmatrix
- ones() Einser-Matrix
- diag() Diagonalmatrix
- random() Zufallsmatrix

Graphische Darstellungen:

```
np.linspace(start, stop, num)

x = np.linspace(0, 2*math.pi ,100)
y = np.sin(x) # math.sin() nicht für Vektoren!
plt.plot(x, y, color='red')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.title('Plot von y = sin(x)')
plt.show()

fig, axs = plt.subplots(2)  # Gestapelte Plots
fig.suptitle('Vertically stacked subplots')
```

Funktionen

```
def log17(x):
        y = np.log(x)/np.log(17)
        return y
```

Ableiten im Python-File -> synpy

Taylorreihe

Funktionen in der Umgebung eines Punktes durch Polynome (Taylorpolynome) annähern.

Oft genug differenzierbare Funktion f an Entwicklungspunkt xo:

$$t(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^k(x_0)}{k!}(x - x_0)^k$$
$$= a_0 + a_{1(x-x_0)} + a_2(x - x_0)^2 + \ldots$$

Fehlerschätzung: Funktionsapproximation durch Taylor-Polynome

Fehlerschätzung per Restglied

Verbleibendes Restglied nach Lagrange mit Zwischenstelle z für exakten Fehler:

$f(x) - t_k(x) = \frac{f^{(k+1)}(z)}{(k+1)!}(x - x_0)^{(k+1)}$ wobei z unbekannt:

Fehlerschranke:

$$|f(x) - t_k(x)| \leq max\ z \left|\frac{f^{(k+1)}(z)}{(k+1)!}(x - x_0)^{(k+1)}\right| = \left|\frac{M}{(k+1)!}(x - x_0)^{(k+1)}\right|$$

```
X = np.linspace(0,2,20)  # Wertebereich x-Achse (von 0 bis 2 in 0.1er Schritten)
x0 = 1              x1 = 0.5

t0 = np.sin(x0) + np.ones(len(x))
t1 = t0 + np.cos(x0)*(x-x0)
t2 = t1 - np.sin(x0)/math.factorial(2)*(x-x0)**2  # mit math.factorial() rechnen !nicht mit Brüchen!
t3 = t2 - np.cos(x0)/math.factorial(3)*(x-x0)**3

plt.figure(1)
plt.plot(x,np.sin(x),x,t0,x,t1,x,t2,x,t3)
plt.legend(['np.sin','t0','t1','t2','t3'])

print( 'Sinus: Fehler bei x =', x1, ':', abs( np.sin(x1) - t3[5]))
   # Fehlerschätzung: Funktionsapproximation durch TP
   # Taylor-Entwicklung exp, x0=0
   x0 = 0
   x = 0.5
   t0 = np.exp(x0)
   t1 = t0 + np.exp(x0) * (x-x0)
   t2 = t1 + np.exp(x0)/np.math.factorial(2) * (x-x0)**2
   t3 = t2 + np.exp(x0)/np.math.factorial(3) * (x-x0)**3

   # Vergleich und Fehlerschranke
   t3
   f = np.exp(x)
   M = np.exp(0.5);
   fehler = abs(t3-f)
   schranke = M/np.math.factorial(4) * (x-x0)**4

   print( 'Fehler:', fehler )
   print( 'Obere Schranke:', schranke )

   # Spezialfall: Linearisierung von Funktionen
   x = np.linspace(0.5,1.5,100)  # Wertebereich x-Achse
   y = np.sqrt(x)
   yy = 1 + 0.5*(x-1)

   fig, axs = plt.subplots(2, 1, constrained_layout=True)
   axs[0].plot( x, y, x, yy )
   axs[0].legend(['Original','Linearisiert'])
   axs[0].set_title( 'Wurzelfunktion' )
```

Abweichung des TP von der Funktion wird begrenzt durch Maximum M der nächsthöheren Ableitung und anderseits der Entfernung vom Entwicklungspunkt

Spezialfall: Linearisierung von Funktionen
Eine komplizierte Funktion f wird durch eine lineare Funktion in der Nähe von x_0 ersetzt. Gesucht: Approximation von x_0
$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$ -> Bsp. $7^{1/3}$ -> 7 ersetzen durch $x_0 = 8$

Beispiel: Linearisierung der Wurzel:

$$(x_0 = 4)\ \sqrt{5} \approx \sqrt{4} + \frac{1}{2\sqrt{4}}(5 - 4) = 2.25$$

Lösen von nicht linearen Funktionen / Suche von Nullstellen

Binäre Suche
Rateversuche, bis Zahl gefunden wird -> so dass stets $f(a) * f(b) < 0$ erfüllt ist, x immer mehr annähern

Gegeben: Obergrenze n
Gesucht: natürliche Zahl z mit 0 < z < n

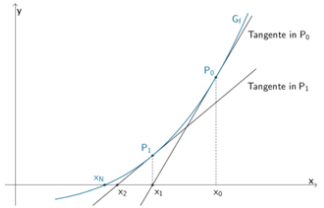
Bisektion, Intervallhalbierung
Anwendung der Idee auf die Nullstellensuche einer stetigen Funktion
Beispiel:
$f(x) := x^2 - 3 \rightarrow$ f ist stetig, d.h. $f(1) < 0\ und\ f(2) > 0$
$\rightarrow Intervall\ (1,2)\ muss\ eine\ Nullstelle\ von\ f\ enthalten$

Zähler Iterationsschritte: $k = 0, \ldots, n$
Startintervall: $k = 0 \rightarrow [a_0; b_0]\ falls\ f(a_0) * f(b_0) < 0$

1. Näherungswert für Nullstelle x_0: $x_0 = \frac{a_0 + b_0}{2}$
   ➔ Wenn $f(a_0) * f(b_0) < 0$: Neues Intervall: $k = 1 \rightarrow [a_1; b_1] = [x_0; b_0]$ *(Neue untere Grenze)*
   ➔ Sonst $k = 1 \rightarrow [a_1; b_1] = [a_0; x_0]$ *(Neue obere Grenze)*
2. Näherungswert für Nullstelle x_1: $x_1 = \frac{a_1 + b_1}{2}$
   ➔ Wenn $f(a_1) * f(b_1) < 0$: Neues Intervall: $k = 2 \rightarrow [a_2; b_2] = [x_1; b_1]$ *(Neue untere Grenze)*
   ➔ Sonst $k = 2 \rightarrow [a_2; b_2] = [a_1; x_1]$ *(Neue obere Grenze)*
3. Weiterführen bis Abbruchkriterium erfüllt: $|f(x_k)| < gewünschte\ Toleranz$ (Konvergenz)
4. Schranke für den absoluten Fehler der Näherung x_k: $|x - x_k| < \frac{b_0 - a_0}{2^{k+1}}$

Newton-Verfahren
Durch ein iteratives Verfahren nähert man sich vom Startpunkt $x0$ mit einer Tangente (Ableitung) an der Nullstelle bzw. an den Schnitt-punkt der x-Achse.

Beispiel siehe Grafik:
Tangenten auf Höhen von xn eingezeichnen bis xN gefunden wird



Voraussetzungen:
f nicht linear, f'(xn) ≠ 0 (Startwert möglichst nahe an realen Nullstelle)

```
# Bisektion zur Intervallhalbierung
def bisektion( f, a, b, tol ):
   k = 0 # zaehlt Schleifendurchlaeufe

   if f(a)*f(b) > 0:
      print( 'Bisektion unmoeglich' )
      return

   xk = []
   while abs(b-a) > tol:
      mid = (a+b)/2  # Intervallmittelpunkt
      fmid = f( mid )
      k = k + 1      # Entscheid ob linkes/rechtes Teilintervall
      if fmid * f(b) < 0:
         a = mid
      elif fmid * f(b)> 0:
         b = mid
      else:
         a = mid
         b = mid

      xk.append( mid )
   return xk, k

def A(x):
   y = 0.01 * x
   return y

def N(x):
   y = x**(-0.2) + x**(-0.4)
   return y
```

➔ Parameter für Funktion bisektion übergeben

```
# Newton-Verfahren
def newton( f, fstrich, x0, tol ):
   k = 0
   err = tol + 1
   xk = [x0]

   # Newton
   while err > tol:
      x0 = x0 - f(x0)/fstrich(x0) # Newtonschritt
      err = abs(x0 - xk[-1])
      k = k + 1
      xk.append( x0 )

   return xk, k

def df(x):
   y = -0.2*x**(-1.2) - 0.4*x**(-1.4) - 0.01
   return y

xn, k = newton( f, df, a, tol )
print( 'b) Preis =', xn[-1], 'in', k, 'Schritten' )
```

Weiteres Beispiel in Python Skript (# Bisektion & Newton Aufgabe Angebot / Nachfrage)

Eigenschaften:
Quadratische Konvergenz: Anzahl der richtigen Nachkommastellen verdoppeln sich mit jedem Schritt
Kleines Residuum ≠ kleiner Fehler
Nicht global konvergent

Allgemeines Newton-Verfahren:
Algorithmus in den einzelnen Schritten: mit Beispiel: $f(x) = x^3 - 2x^2 - x - 8$

1. $x0$ raten, falls nicht angegeben
Man setzt eine beliebige Zahl in die Funktion $f(x)$ ein und schaut wie nahe es an Null ist $f(1) = -5$, $f(2) = 10$ nehmen wir einfach 1 als Startwert x_0

2. $f(x)$ ableiten
$$\rightarrow f'(x) = 3x^2 - 4x - 1$$

3. Iterative Berechnung
Jetzt muss man den Startwert x_0 in die Formel für x_n einsetzen und kann fortan iterieren

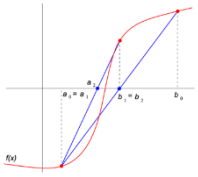
Erste Iteration $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

N-te Iteration (allgemeine Formel) $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

4. Das macht man nun solange bis die gewünschte Genauigkeit erreicht wurde. Bzw. bis $f(x) \approx 0$

Sekanten Verfahren
Anstelle der Nullstelle einer Kurve wird die Nullstelle einer Sekante von zwei Startpunkten aus berechnet.

1. 2 Punkte x_k und x_{k-1} aus dem Intervall [a;b] bestimmen (dabei muss $f(a0) * f(b0) < 0$ gelten!)



2. x_k und x_{k-1} in die Iterationsformel einsetzen:
$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} * f(x_k)$$

Minimierung einer differenzierbaren Funktion
Anwendung des Newton-Verfahrens
Hinweise:
- Konvergenz zu lokalem anstatt zu globalem Minimum
- Stagnation ohne Konvergenz bei einer Funktion mit weiten Plateaus möglich
- Nicht-Konvergenz und Oszillationen, z.B. bei steilen Schluchten möglich
- Verlassen guter Minima hin zu unerwünschten Extremwerten
- Randwerte müssen gesondert geprüft werden
- Spezialisierten Algorithmen existieren

Differentialrechnung von Funktionen zweier Variablen
2D-Linearisierung:
$$t_{1(x,y)} = f(x_0, y_0) + \frac{\partial f}{\partial x}(x_0, y_0)(x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0)(y - y_0)$$
- Beobachtung: Nullstellen der Linearisierung t1(x, y) = 0 sind eine Gerade.
- Um eindimensionale Nullstellensuche und Newton-Verfahren zu verallgemeinern, braucht man zwei Funktionen in zwei Variablen; neue 2D-Newton-Iterierte als Schnittpunkt der zwei Geraden.

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

Sekatnenverfahren
```
def sekant( f, x0, x1, tol ):
    n = 0 # Iterationszaehler
    nmax = 50  # Max. Iterationen
    err  = tol + 1 # Fehler
    x = x1

    while err > tol:
        x = x1 - (x1-x0) / (f(x1)-f(x0)) * f(x1)  # Ierationsvorschrift
        err = abs(x-x1)
        x0 = x1
        x1 = x
        n = n+1
        if n > nmax:
            print( 'Keine Konvergenz...' )
            break

    return x, n


# Funktion Newton Minimum
def newton_minimum( f, fstrich, f2strich, a, b, x0 ):
    xpts = np.linspace( a, b, 1000 )
    fig, axs = plt.subplots( 2 )
    axs[0].plot( xpts, f(xpts), xpts, xpts*0 )
    axs[0].set_title('Funktion')
    axs[1].plot( xpts, fstrich(xpts), xpts, xpts*0 )
    axs[1].set_title('Ableitung')

    # Newton-Verfahren
    x = x0
    tol = 1.e-8
    res = abs( fstrich(x) )
    k = 0
    axs[0].plot( x, f(x), marker ='*')
    axs[1].plot( x, fstrich(x), marker ='*')
    plt.pause(2)

    while res > tol:
        x = x - fstrich(x) / f2strich(x)  # Newtonschritt
        axs[0].plot( x, f(x), marker ='*')
        axs[1].plot( x, fstrich(x), marker ='*')
        plt.pause(2)

        res = abs( fstrich(x) )
        k = k + 1
    return x, k

def f(x):            ## ABÄNDERN Stammfunktion
    y = x**2 * np.sin(x)
    return y

def fstrich(x):      ## ABÄNDERN 1. Ableitung
    y = 2*x*np.sin(x) + x**2 * np.cos(x)
    return y

def f2strich(x):     ## ABÄNDERN 2. Ableitung
    y = 2*np.sin(x) + 4*x*np.cos(x) - x**2*np.sin(x)
    return y

x, k = newton_minimum( f, fstrich, f2strich, 2, 7, 5 )
print('x =', x, 'in', k, 'Schritten' )
```

Jaccobi-Matrix
$$f(x_1, x_2) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} 2x_1 + 4x_2 \\ 4x_1 + 8x_2^3 \end{pmatrix}$$

```
x = np.linspace( -5, 5, 200 )
y = np.linspace( -5, 5, 200 )

xv, yv = np.meshgrid(x, y)

f1 = 2*xv+4*yv
f2 = 4*xv+8*yv**3

fig, axs = plt.subplots( 1, 2, subplot_kw={"projection": "3d"} )
surf1 = axs[0].plot_surface( xv, yv, f1, cmap=plt.cm.cool-warm, linewidth=0 )
surf2 = axs[1].plot_surface( xv, yv, f2, cmap=plt.cm.cool-warm, linewidth=0 )
```

Mehrdimensionales Newton-Verfahren
2D Newton
Nullstellen von einer Matrix finden

Gauss-Algorithmus
Lineares Gleichungssystem Ax = B
$A := \begin{bmatrix} 10^{-20} & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, b := \begin{bmatrix} 1 \\ 1000 \\ 1 \end{bmatrix}$ *somit wäre* $x = \begin{bmatrix} 0 \\ 1 \\ 1000 \end{bmatrix}$

- Spalten von A sind linear unabhängig
- Matrix A ist invertierbar
- Lineare Gleichungssystem ist eindeutig lösbar
  ➔ Kleine und exakte Null-Pivots sind für die numerische Umsetzung des Gauss-Algorithmus störend

LA MATRIZEN REGELN!!!

$$\left(\begin{array}{ccc|c} 1 & -2 & -1 & 3 \\ 2 & -1 & 1 & 0 \\ 3 & -6 & -5 & 3 \end{array}\right) \begin{matrix} \text{U} \\ \text{y} \end{matrix}$$

Nutzen der Faktorisierung A = L * U
Für die numerische Problematik wird A = L*U verwendet.
Lösung von Ax = b mit A = L * U

1. Zwischenlösung y aus Ly=b, danach
2. X aus Ux =y

Falls Zeilenvertauschungen notwendig sind, können diese in einer Permutationsmatrix P zusammengefasst werden. -> PA = L * U

LA Matrizen Regeln -> Berechnung von L / U von Hand!

Berechnung von L und U mittels Gauss-Algorithmus
```
def LUFaktoren(A):
# Gauss-Algorithmus zur Berechnung der Dreiecksmatrizen L und U, A = L*U
# L: linke untere Dreiecksmatrix, U ist rechte obere Dreiecksmatrix
    n = len( A )
    L = np.eye( n )
    U = A.copy()
    for k in range( n ):
        if U[k,k] == 0:
            raise Exception( 'Null-Pivot' )
        else:
            L[k+1:n,k] = U[k+1:n,k] / U[k,k]
            for j in range( k+1, n ):
                U[j,:] = U[j,:] - L[j,k] * U[k,:]
    return L, U

A = np.array( [[1, 2, 3], [1, 1, 1], [3, 3, 1]] )

L, U = LUFaktoren(A)

print( 'Die berechneten Dreiecksmatrizen L, U sind:' )
print( L )
print( U )

print('Zur Probe: L*U=: ')
print( L@U )
```

Beurteilung einer Näherungslösung
- Gegeben: lin. Gleichungssystem Ax = b mit theoretisch exakter Lösung $x = A^{-1}b$ sowie errechneter Näherungslösung (Fehler unbekannt)
- Indirekte Fehlermessung per Residuum

```
# 2D Newton-Verfahren
def newton2D( f, Jf, x0 ):
    x = x0
    tol = 1e-8
    k = 0
    y = f(x)
    res = np.linalg.norm(y)

    while res > tol:
        k = k + 1
        J = Jf(x)
        d = np.linalg.solve(J, y)
        x = x - d
        y = f(x)
        res = np.linalg.norm(y)

    return x, k


def f1(x):
    f = ( x[0]**2 + x[1]**2 + 0.6*x[1] - 0.16, x[0]**2 - x[1]**2 + x[0] - 1.6*x[1] - 0.14 )  # x[0] = x, x[1] = y
    return f

def Jf1(x):
    J = np.array([[ 2*x[0], 2*x[1] + 0.6 ], [ 2*x[0] + 1, -2*x[1] - 1.6 ]])
    return J

x, k = newton2D( f1, Jf1, [-1.0,-1.0] )
print('x =', x, 'in', k, 'Schritten' )
```

 # Gauss mit rechter oberer Dreiecksmatrix
Berechnet die Loesung x von U x = y durch Rueckwaerts-Einsetzen, dabei ist U eine rechte obere Dreiecksmatrix
```
def backsubst( U, y ):
    n = len( y )
    x = np.zeros([n,1])
    yy = y.copy()
    for i in range( n-1, -1, -1 ):
        for j in range( i+1, n ):
            yy[i] = yy[i] - U[i,j] * x[j]

        if U[i,i] == 0:
            raise Exception( 'U-Matrix ist singulaer!' )
        else:
            x[i] = yy[i]/U[i,i]
    return x

U = np.array( [[1, -2, -1], [0, 3, 3], [0, 0, -2]] )
y = np.array( [[3], [-6], [-6]] )
x = backsubst( U, y )
print( 'Lösung', x )
r = y - U @ x #Loese U x = y, idealerweise  y - U x = 0
print( 'Residuum\n', r )
```

#Gauss-Algorithmus mit linker unterer Dreiecksmatrix
Loese L y = b durch Vorwaerts-Einsetzen (L: linke untere Dreiecksmatrix)
```
def forwardsubst( L, b ):
    n = len( b )
    y = np.zeros([n,1])
    bb = b.copy()
    for i in range( n ):
        for j in range( i ):
            bb[i] = bb[i] - L[i,j] * y[j]

        if L[i,i] == 0:
            raise Exception( 'L-Matrix ist singulaer!' )
        else:
            y[i] = bb[i]/L[i,i]
    return y

L = np.array( [[1, 0, 0], [2, 1, 0], [3, 0, 1]] )
U = np.array( [[1, -2, -1], [0, 3, 3], [0, 0, -2]] )
b = np.array( [[3], [0], [3]] )

y = forwardsubst( L, b )
x = backsubst( U, y )
print( 'LÃ¶sung\n', x )
r = b - (L @ U) @ x
print( 'Residuum\n', r )
```

```
A = np.array( [[1, 2, 3], [3, 4, -2], [7, -3, 5]] )
b = np.array( [[1.2], [2.3], [3.4]] )
xSchlange = np.linalg.solve(A,b)
print('berechnete L¨osung x=', xSchlange )
r = b - A @ xSchlange # Residuum
print( 'Residuum r =', r )
print( 'Konditionszahl =', np.linalg.cond(A,2))
e = np.linalg.cond(A,2) * np.linalg.norm(r) / np.linalg.norm(b)
print( 'Schranke fuer relativen Fehler Gleichung (1):', e )
```

```
# Inverse einer Matrix berechnen
A = np.array( [[0.2, -5, 3, 0.4, 0],
               [-0.5, 1, 7, -2, 0.3],
               [0.6, 2, -4, 3, 0.1],
               [3, 0.8, 2, -0.4, 3],
               [0.5, 3, 2, 0.4, 1]] )

n = len(A)

Ainv, L, U = sp.lu(A)
Ainv = Ainv.T # Ainv = P.T

for k in range( n ):
   y = forwardsubst( L, Ainv[:,k] ) # Funktion forwardsubst
   print( backsubst( U, y ) ) # & backsubst einlesen
   Ainv[:,k] = backsubst( U, y )[:,0]

print( 'Inverse' )
print( Ainv )
```

Ausgleichsrechnung (Regression)

Lineare Regression – Vergleich Interpolation und Regression

Interpolation:

Konstruiert Näherungswerte einer Funktion zwischen Punkten, an denen die Funktion exakt bekannt ist.

Regression:

Nimmt an, dass die bekannten Funktionswerte Fehler enthalten. Regression findet in der gewählten Modellklasse (z.B. lin. Funktionen) das Modell, welches eine Fehlerfunktion (r^2) minimiert.

Ausgleichspolynome aus Minimierung der Fehlerquadrate

Gesucht: Das beste Polynom p_n vom Höchstgrad n sodass die Fehlerquadratsumme minimiert wird

$|A * p - y|^2 \leq |A * q - y|^2 \ für\ alle\ q \in R^{n+1}$

P minimiert also die Länge des Residuenvektors r = y - A*p

Intuition:

Falls nur ein Parameter p: nach oben geöffnete Parabel / eindeutiges Minimum

Optimalitätsbedingung: $\nabla F = 0 \rightarrow (A^T * A) * p = (A^T * y)$

- Die Matrix ($A^T \cdot A$) ist quadratisch und symmetrisch.
- ($A^T \cdot A$) invertierbar falls Spalten von A linear unabhängig
- Falls ($A^T \cdot A$) invertierbar: Normalengleichung eindeutig lösbar

Mit der Normalgleichung: $\boldsymbol{A^T A\lambda = A^T y}$

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) \\ f_1(x_2) & f_2(x_2) \\ f_1(x_3) & f_2(x_3) \\ f_1(x_4) & f_2(x_4) \end{pmatrix} = \begin{pmatrix} f_1(1) & f_2(1) \\ f_1(2) & f_2(2) \\ f_1(3) & f_2(3) \\ f_1(4) & f_2(4) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{pmatrix}$$

$$\Rightarrow A^T A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{pmatrix} = \begin{pmatrix} 30 & 10 \\ 10 & 4 \end{pmatrix},$$

$$A^T y = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 6 \\ 6.8 \\ 10 \\ 10.5 \end{pmatrix} = \begin{pmatrix} 91.6 \\ 33.3 \end{pmatrix}$$

$$\begin{pmatrix} 30 & 10 \\ 10 & 4 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 91.6 \\ 33.3 \end{pmatrix} \quad \Rightarrow \quad a = 1.67,\ b = 4.15$$

Die gesuchte Ausgleichsgerade lautet also $f(x)$ = 1.67x + 4.15 es ist also die gleiche wie vorher.

Verbesserte Berechnung von p (Orthogonalität)

Normalengleichung $(A^T * A) * p = (A^T * y)$ durch Distributivgesetz und mit Residuum r: $0 = A^T(y - A * p) = A^T * r$

Interpretation über Skalarprodukt (minimales Residuum ist senkrecht zu den Spalten von A)

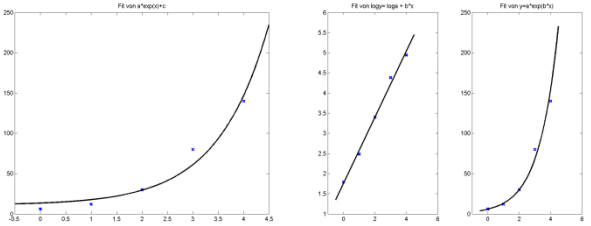
Orthogonalprojektion von y auf Spalten von A äquivalent, aber numerisch vorteilhafter

```
Regressionsmodell
x = np.linspace( -2.5, 2.5, 50 )
y = np.tanh(x).reshape(-1,1) # Generiere Beispielfunktion

# Ansatz: p3(x) = a + b*x + c*x^2 + d*x^3, p=(a,b,c,d)^T
A = np.array([ np.ones(len(x)), x, x**2, x**3 ]).T # Berechnung von Zahlen für A

# Normalengleichung loesen:
p = np.linalg.solve( A.T@A, A.T@y ) # Lösen der Gleichung
yy = p[0] + p[1]*x + p[2]*x**2 + p[3]*x**3 # Werte der Funktion (zum Einzeichnen im Plot)

# Graphische Darstellung
plt.figure(2)
plt.plot( x, y, 'b-.' ,x ,yy , 'k-' )

# Vergleich mit polyfit
pp = np.polyfit( x, y, 3 )

plt.plot( x, np.polyval( pp, x ), 'r-' )
plt.show()

Ausrechnen von p
x = np.linspace( -2.5, 2.5, 50 )
y = np.tanh(x).reshape(-1,1)

grad = 3
V = np.zeros( ( len(x), grad+1))
# Generiere rechteckige Vandermonde-Typ-Matrix
V[:,grad] = np.ones( len(x) )
for j in range( grad-1, -1, -1 ):
  V[:,j] = x * V[:,j+1]

# Anstatt Normalengleichung: direkte Python-Lsg
p = np.linalg.solve( V.T@V, V.T@y )
print( p )
print()

# Vergleich mit den Koeffizienten berechnet durch Numpy's polyfit
pp = np.polyfit(x,y,grad)
print( pp )

Nichtlineare Ausgleichsprobleme
x = np.array([0, 1, 2, 3, 4]) # Gegebene Punkte x-Werte
y = np.array([6, 12, 30, 80, 140]).reshape(-1,1) # Y-Werte
ylog = np.log(y) # logarithmierte Daten

# Generiere rechteckige Vandermonde-Typ-Matrix
V = np.zeros( ( len(x), 2) )
V[:,1] = np.ones( len(x) )
V[:,0] = x

# Stelle lineares Gleichungssystem der Normalengleichungen auf
N = V.T @ V
# logarithmierte Datenpunkte als rechte Seite!
blog = V.T @ ylog
p = np.linalg.solve( N, blog )

# rekonstruiere Parameter fuer Originaldaten
a = np.exp( p[1] )
b = p[0]

# Auswertung und Vergleichsplot
xx = np.linspace( x[0]-0.5, x[-1]+0.5, 100 )

plt.figure(4)
yy = p[1] + p[0]*xx
plt.plot( x, ylog, 'bx ', xx, yy, 'k' )
plt.title('Fit von logy= loga + b*x')

plt.figure(5)
yy = a * np.exp( b*xx )
plt.plot( x, y, 'bx ', xx, yy, 'k' )
plt.title('Fit von y=a*exp(b*x)')
```

Perspektive: Nichtlineare Ausgleichsprobleme

Mit Hilfe der Logarithmusgesetzte berechnen:

$y = a * \exp(b * x) \rightarrow \ln(y) = \ln(a) + b * x$



Aufgabenblatt 8 anschauen

Polynom-Interpolation als lineares Gleichungssystem

Lineare Interpolation

$$y = f(x_0) + \frac{x - x_0}{x_1 - x_0} * (f(x_1) - f(x_0)$$

1. x raten -> Wert sollte zwischen x_0 und x_1 liegen
2. x in obige Formel einsetzten und somit erhält man y

Vandermonde-Matrix

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

Dieses Polynom vom Grad n hat n+1 Koeffizienten

Interpolationsaufgabe:

Für gegebene x,y-Paare sind Koeffizienten zu bestimmen so dass: $p(x_i) = y_i, i = 0, \ldots, n$. Dies lässt sich durch das Lösen des linearen Gleichungssystems Vx = y erreichen, wobei V der Vandermonde-Matrix entspricht.

$$V(x_1, x_2, \ldots, x_n) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}$$

Beispiel:

Gegeben:

$p(x) = a_4 * x^4 + a_3 * x^3 + a_2 * x^2 + a_1 * x + a_0,$

$Punkte\ (0,1)\ ,(0.5,1)\ ,(1,0)\ ,(1.5,0)\ ,(2,3)$

Gesucht: Koeffizienten

Lösung Code: →

Polynom-Interpolation nach Lagrange

Messdaten anhand der Lagrange Interpolation zu einer Funktion (Polynom) approximieren.

Konstruktion des Interpolationspolynoms aus n+1 Grundpolynomen :

$$\ell_i(x_k) = \begin{cases} 0, & k \neq i \\ 1, & k = i \end{cases}$$

$\rightarrow\ p(x) = y_0 * \ell_0(x) + y_1 * \ell_1(x) + \cdots + y_n * \ \ell_n(x)$

Lösung durch das Grundpolynom nach Lagrange vom Grad n:

$$\ell_i(x) = \frac{(x - x_0) * \ldots * (x - x_{i-1}) * (x - x_{i+1}) * \ldots * (x - xn)}{(x_i - x_0) * \ldots * (x_i - x_{i-1}) * (x_i - x_{i+1}) * \ldots * (x_i - xn)}$$

Beispiel:

```
# Interpolation mit Vandermatrix
# Interpolationsdaten
x = np.linspace( 0, 2, 5 )
y = np.array( [1, 1, 0, 0, 3] ) # Y-Werte aus den gegebenen Punkten

# Berechnung der Koeffizienten des Interpolationspolynoms
V = np.vander(x)
p = np.linalg.solve( V, y )

# Vergleichsplot
xx = np.linspace( -1, 3, 100 )
yy = np.polyval( p, xx )
plt.plot( x, y, 'b*', xx, yy, 'r' )
plt.title( 'Interpolationspolynom' )
plt.show()

Konditionszahl:
print( np.linalg.cond( np.vander( np.linspace( 0, 2, 5 )), 2 ))
# Mit zunehmender Dimension nimmt Konditionszahl zu.

Stützstellen berechnen:
x = np.array([1,3,5]) # Bereich
y = 3 * x**2 - 4 * x – 1 # Gegebene Funktion

V = np.vander(x)
result = np.linalg.lstsq( V, y, rcond=None )
p = result[0]
print( p )


# Interpolation nach Lagrange
def lagrangepoly( n, x, y, xx ):
   nn = len( xx )
   # L speichert in jeder Zeile die Auswertung eines Lagrangepolynoms
   L = np.ones((n + 1, nn))
   for k in range( n+1 ):
      for kk in range( n+1 ):
         if kk != k:
            L[kk,:] = L[kk,:] * (xx - x[k]) / (x[kk]-x[k])
   return y@L, L

x = np.array( [1, 2, 3, 4] ) # x0, x1, x2, x3
n = len(x) - 1 # Polynomgrad
y = 1 / x  # Funktion
xx = np.linspace( 0.5, 4.5, 40 ) # Auswertungsppunkte
yy, L = lagrangepoly( n, x, y, xx )
plt.figure(0)
plt.plot( xx, L.T )
plt.title('Lagrange Polynome')
plt.figure(1)
plt.plot( x, y, 'b*', xx, yy, 'r--', xx, 1/xx, 'g-' )
plt.legend(['Punkte','Interpolation','Funktion y=f(x)'])
plt.title('Interpolationspolynom')

# Verwendung: yy=stueckweise_linear(n,x,y,xx)
# Interpoliert stueckweise linear zwischen den Punkten (x[i],y[i])
# Eingabe: n  - Anzahl der Stuetzstellen (x,y), n>=2
#          x  - Stuetzstellen, paarweise verschieden
```

Lagrange Interpolation

Messdaten:

$(x_0, y_0) = (-1, 4)$ $\quad L_0 = \frac{x+0}{-1+0} \cdot \frac{x+3}{-1+3} \cdot \frac{x+5}{-1+5} = -\frac{1}{8}\,x(x-1)(x-3)$

$(x_1, y_1) = (0, 3)$ $\quad L_1 = \frac{x+1}{0+1} \cdot \frac{x+3}{0-1} \cdot \frac{x+5}{0-3} = \frac{1}{5}\,(x+1)(x-1)(x-3)$

$(x_2, y_2) = (1, -2)$ $\quad L_2 = \frac{x+1}{1+1} \cdot \frac{x-0}{1-0} \cdot \frac{x+5}{1+5} = -\frac{1}{4}\,x(x+1)(x-3)$

$(x_3, y_3) = (3, 8)$ $\quad L_3 = \frac{x+1}{3+1} \cdot \frac{x-0}{3-0} \cdot \frac{x-1}{3-1} = \frac{1}{24}\,x(x+1)(x-1)$

$\rightarrow P(x) = 4 \cdot L_0 + 3 \cdot L_1 + (-2) \cdot L_2 + 8 \cdot L_3$

Nachteile: Grosser Rechenaufwand, bei einer Stützstelle mehr, startet der Rechenaufwand von Neuem

Spline-Interpolation

Stückweise lineare Interpolation

Wenn Polynome zwischen zwei Stützstellen teilweise stark schwanken kann eine stückweise Interpolation vorgenommen werden.
Die Spline-Interpolation lässt sich mit geringem linearen Aufwand berechnen, liefert aber im Vergleich zur Polynominterpolation eine geringere Konvergenzordnung.

Stückweise Definition:

$$s(x) = \begin{cases} s_0(x) & x \in [x_0, x_1] \\ s_1(x) & x \in [x_1, x_2] \\ \vdots & \vdots \\ s_{n-1}(x) & x \in [x_{n-1}, x_n] \end{cases}$$

Stückweise lineare Interpolation bestimmt s_i eindeutig: $\quad s_i(x_i) = f_i$, $\quad s_i(x_{i+1}) = f_{i+1}, \quad i = 0, \ldots, n-1.$

Formeln:

GLS für n = 5 => n-1 X n-1 Matrix

$$\begin{pmatrix} 2(h_0+h_1) & h_1 & 0 & 0 \\ h1 & 2(h_1+h_2) & h_2 & 0 \\ 0 & h_2 & 2(h_2+h_3) & h_3 \\ 0 & 0 & h_3 & 2(h_3+h_4) \end{pmatrix} * \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \end{pmatrix} = \begin{pmatrix} c_1 - h_0 * M_0 \\ c_2 \\ c_3 \\ c_4 - h_4 * M_5 \end{pmatrix}$$

$$c_i = \frac{6}{h_i} * (y_{i+1} - y_i) - \frac{6}{h_{i-1}} * (y_i - y_{i-1})$$

Stückweise kubische Splines

Nachteil der stückweise linearen Interpolation: Funktion hat «Spitzen»
Um dem entgegen zu wirken kann eine kubische Fukntion auf jedem Teilintervall angewandt werden.

Praktikum W9

```
#       y - Stuetzwerte
#       xx - Auswertungspunkte
# Ausgabe: yy - Werte der Interpolation an den jeweiligen
xx
def stueckweise_linear( n, x, y, xx ):
  if (x != np.sort(x)).all():
    print( 'Stuetzstellen muessen aufsteigend geordnet
sein' )
  else:
    yy = np.zeros([len(xx)])
    for i in range( len(xx) ):
      # Finde Stuetzstellen xl,xr sodass xl <= xi <= xr
      if (xx[i] < x[0]) or (xx[i] > x[-1]):
        print( 'Auswertungspunkt ausserhalb des erlaub-
ten Bereichs' )
        return None
      else:
        for j in range( n - 1 ):
          if (xx[i] >= x[j] ) and (xx[i] <= x[j+1]):
            xl, yl = x[j], y[j]
            xr, yr = x[j+1], y[j+1]
            break
      # Werte die dem Intervall [xl,xr] entsprechende line-
are Funktion
      # an der Stelle xi aus
      yy[i] = yl + (xx[i]-xl) * (yr-yl)/(xr-xl)

  return yy

x = np.linspace(1, 8, 8)
y = np.zeros([len(x)])
y[3] = 0.3
# Punkte zum Plotten
xx = np.linspace(1, 8, 29)
# Stueckweise lineare Interpolation an den xx
yy = stueckweise_linear( len(x), x, y, xx )
plt.figure(5)
plt.plot( x, y, 'bo ', xx, yy, 'g-' )
plt.show()

# Kubische Splines – besseres Beispiel?
x = np.array( [0, 1, 2] )
y = np.array( [0, math.pi, 0] )
A = np.array( [[0, 0, 0, 1, 0, 0, 0, 0],
               [1, 1, 1, 1, 0, 0, 0, 0], # Interpolation s1
               [0, 0, 0, 0, 1, 1, 1, 1],
               [0, 0, 0, 0, 8, 4, 2, 1], # Interpolation s2
               [3, 2, 1, 0, -3, -2, -1, 0],
               [6, 2, 0, 0, -6, -2, 0, 0], # stetige 1.,2. Ableitungen
               [0, 2, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 12, 2, 0, 0]] # %Zusatzbed. s''(0)=s''(2)=0
              )
b = np.array( [0, math.pi, math.pi, 0, 0, 0, 0, 0] )
result = np.linalg.lstsq( A, b, rcond=None )
p = result[ 0 ] # %s1 mit Koeffizienten 1-4, s2 mit Koeffi-
zienten 5-8

xx1 = np.linspace(0, 1, 100)
s1 = p[3] + p[2]*xx1 + p[1]*xx1**2 + p[0]*xx1**3
xx2 = np.linspace(1, 2, 100)
s2 = p[7] + p[6]*xx2 + p[5]*xx2**2 + p[4]*xx2**3
plt.figure(6)
plt.plot( x ,y ,'bo ', xx1, s1, 'g-', xx2, s2, 'r-' )
plt.show()
```

Numerische Integration

Numerische Quadratur nach Newtion-Cotes

$$\int_a^b f(x)\,dx$$

1. Lösung ist nicht analytisch, d.h. Berechnung einer Stammfunktion ist nicht nötig. Gesucht ist eine Approximation des Flächeninhalts unter f.
2. Stammfunktion muss nicht einmal existieren, wie z.B. bei $\exp(-x^2/2)$ aus der Normalverteilung.
3. Gilt auch fur Funktionen f ohne bekannte geschlossene Form, d.h. für f nur in Form einer Wertetabelle vorliegend.
4. Ober- und Untersummen aus der Analysis i. a. nicht nützlich, da lokale Min./Max. von f unbekannt.

```
def mittelpunktsregel( a, b, f ):
  return (b-a) * f((a+b)/2)

def trapezregel( a, b, f ):
  return (b-a)/2 * f(a) + (b-a)/2 * f(b)

def simpsonregel( a, b, f ):
  return (b-a)/6*f(a) + 4*(b-a)/6*f((a+b)/2) + (b-a)/6*f(b)

# Gewichte auf Standardintervall [0,1]
A = np.array([[1, 1, 1], [0, 1/3, 2/3, 1], [0, 1/9, 4/9, 1], [0,
1/27, 8/27, 1]])
b = np.array([1, 1/2, 1/3, 1/4])

w = np.linalg.solve(A, b)
print('w=',w)
# Gewichte auf [a,b] = (b-a)*w
```

Mittelpunktregel (Rechtecksregel):
n=0, Interpolation mit Konstante

$$M_0 := (b-a) * f\left(\frac{a+b}{2}\right)$$

Trapezregel:
n=1, Interpolation mit Gerade

$$T_1 := \frac{b-a}{2} * f(a) + \frac{b-a}{2} * f(b)$$

Simpsonregel:
n=2, Interpolation mit Parabel

$$S_2 := \frac{b-a}{6} * f(a) + 4 * \frac{b-a}{6} * f\left(\frac{a+b}{2}\right) + \frac{b-a}{6} * f(b)$$

Konstruktion von Quadraturformeln

Unterteilung des Integrals -> siehe Lagrange
Folgende Approximation verwenden:

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = \sum_{j=0}^{n} y_j \underbrace{\int_a^b \ell_j(x)dx}_{=:w_j}$$

Beispiele:

Trapezregel ($n = 1$, Interpolation mit Gerade):

$$T_1 := \frac{b-a}{2} \cdot f(a) + \frac{b-a}{2} \cdot f(b)$$

Berechnung der Gewichte:

$$\begin{aligned} \int_a^b \ell_0(x)dx &= \int_a^b \frac{x-b}{a-b}dx \\ &= \frac{1}{a-b}\left[\frac{1}{2}x^2 - bx\right]_a^b = \frac{1}{a-b}\left[\frac{b^2-a^2}{2} - (b^2-ab)\right] = \frac{b-a}{2} \\ \int_a^b \ell_1(x)dx &= \int_a^b \frac{x-a}{b-a}dx \\ &= \frac{1}{b-a}\left[\frac{1}{2}x^2 - ax\right]_a^b = \frac{1}{b-a}\left[\frac{b^2-a^2}{2} - (ab-a^2)\right] = \frac{b-a}{2} \end{aligned}$$

Beispiel Simpsonregel ($n = 2$):

$$S_2 := \frac{b-a}{6} \cdot f(a) + 4 \cdot \frac{b-a}{6} \cdot f\left(\frac{a+b}{2}\right) + \frac{b-a}{6} \cdot f(b)$$

Berechnung der Gewichte: Geschickte lineare Substitution $x = \phi(s)$ mit $h = \frac{b-a}{n}$ und $\phi : [0, n] \to [a, b],\ s \mapsto \phi(s) := a + h \cdot s$ ergibt

$$w_j = \int_a^b \ell_j(x)dx = \int_a^b \ell_j(\phi(s))\phi'(s)ds = h\int_0^n \tilde{\ell}_j(s)ds$$

$$\begin{aligned} \int_a^b \ell_0(x)dx &= h\int_0^2 \frac{(hs-h)\cdot(hs-2h)}{(-h)\cdot(-2h)}ds = \frac{h}{2}\int_0^2 (s^2-3s+2)ds \\ &= \frac{h}{2}\left[\frac{1}{3}s^3 - \frac{3}{2}s^2 + 2s\right]_0^2 = \frac{h}{2}\left[\frac{8}{3} - \frac{12}{2} + 4\right] = \frac{h}{2}\cdot\frac{2}{3} = \frac{b-a}{6} \\ \int_a^b \ell_1(x)dx &= h\int_0^2 \frac{(hs-0)\cdot(hs-2h)}{(h)\cdot(-h)}ds = -h\int_0^2 (s^2-2s)ds \\ &= -h\left[\frac{1}{3}s^3 - s^2\right]_0^2 = -h[\frac{8}{3} - 4] = -h\cdot\frac{-4}{3} = 4\frac{b-a}{6} \end{aligned}$$

Alternative Konstruktion der Gewichte:
Alternativ kann man die Gewichte auch aus einem linearen Gleichungssystem ausrechnen, w = A^{-1}b.
Man benutzt, dass alle Interpolationspolynome vom Höchstgrad n mit n+ 1 Interpolationsknoten exakt integriert werden.

```
# NewtonCotes Zusammengesetzt
def NewtonCotesZusammengesetzt( f, a, b, n ):

  Aufruf:  M,T,S = NewtonCotesZusammengesetzt(f,a,b,n)
  Beispiel: M,T,S = NewtonCotesZusammengesetzt(lambda
            x: np.sin(x),0,np.pi,2)

  Eingabe: f:  die zu integrierende Funktion,
           a,b: untere/obere Integrationsgrenzen,
           n:  Anzahl Teilintervalle

Ausgabe: M: Zusammengesetzte Mittelpunktregel
         T: Zusammengesetzte Trapezregel
         S: Zusammengesetzte Simpsonregel

  h = (b-a) / n
  M, T, S = 0, 0, 0

  for i in range( 1, n+1 ):
    left = a + (i-1)*h
    right = a + i*h
    mid = a + (2*i-1)*h/2
    fleft = f(left)
    fright = f(right)
    fmid = f(mid)

    # Zusammengesetzte Mittelpunktregel, intervallweise
                aufaddiert
    M = M + h*fmid

    # Zusammengesetzte Trapezregel, intervallweise auf
                addiert
    T = T + h/2 * (fleft + fright)

    # Zusammengesetzte Simpsonregel, intervallweise auf
                addiert
    S = S + h/6 * (fleft + 4*fmid + fright)

  return M, T, S

# TeÃ®l a)
a, b = 0, math.pi
nmax = 50
lexakt = 2

def f(x):
  return np.sin(x)

def g(x):
  return 2 / ( 1 + x**2 )

M = np.zeros( nmax )
T = np.zeros( nmax )
S = np.zeros( nmax )

for n in range( 1, nmax + 1 ):
  M[n-1], T[n-1], S[n-1] = NewtonCotesZusammengesetzt(
f, a, b, n )

plt.figure(0)
x = np.linspace( 1, nmax+1, nmax )
plt.plot(x, M, 'b', x, T, 'r',x, S, 'k' )
plt.legend(['M(h)','T(h)','S(h)','Location','BestOutside'])
plt.xlabel('n')
plt.title('Aufgabe 2: Konvergenz zusammengesetzte New-
ton-Cotes Formeln')

plt.figure(1)
plt.semilogy(x, abs(M-lexakt), 'b', x, abs(T-lexakt), 'r',
        x, abs(S-lexakt), 'k' )
plt.legend(['M(h)','T(h)','S(h)','Location','BestOutside'])
plt.title( 'Aufgabe 2: Fehler im Vergleich zu exaktem Wert '
+ str(lexakt))
```

Beispiel Trapezregel, diese hat die Form

$$T_1 := w_0 \cdot f(a) + w_1 \cdot f(b).$$

Wenn die Polynome 1 und x exakt integriert werden sollen, so muss gelten

$$\begin{bmatrix} 1 & 1 \\ a & b \end{bmatrix} \cdot \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} \int_a^b 1dx \\ \int_a^b xdx \end{pmatrix} = \begin{pmatrix} b-a \\ b^2/2 - a^2/2 \end{pmatrix}$$

Man findet dann durch Lösen

$$T_1 := \frac{b-a}{2} \cdot f(a) + \frac{b-a}{2} \cdot f(b).$$

Die Simpsonregel hat die Form

$$S_2 := w_0 \cdot f(a) + w_1 \cdot f\left(\frac{a+b}{2}\right) + w_2 \cdot f(b).$$

Wenn die Polynome 1, x und x^2 exakt integriert werden sollen, so muss gelten

$$\begin{bmatrix} 1 & 1 & 1 \\ a & \frac{a+b}{2} & b \\ a^2 & (\frac{a+b}{2})^2 & b^2 \end{bmatrix} \cdot \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} \int_a^b 1dx \\ \int_\beta^b xdx \\ \int_a^b x^2dx \end{pmatrix} = \begin{pmatrix} b-a \\ b^2/2 - a^2/2 \\ b^3/3 - a^3/3 \end{pmatrix}$$

und daraus

$$S_2 := \frac{b-a}{6} \cdot f(a) + 4 \cdot \frac{b-a}{6} \cdot f\left(\frac{a+b}{2}\right) + \frac{b-a}{6} \cdot f(b).$$

Zusammengesetzte Newton-Cotes Formeln

Stückweise Integration

Idee: Zusammengesetzte Mittelpunkt- und Trapezregel

- Je kleiner die Teilintervalle, desto besser stimmen Interpolationspolynome und Originalfunktion überein
- Interpolationspolynome können zwischen zwei Stützpunkten stark schwanken

Formeln:

Unterteilung des Intervalls [a, b] in n Teilintervalle der Grösse $h = \frac{b-a}{n}$

Zusammengesetzte Mittelpunktregel:

$$M_0(h) = h\left[f\left(\frac{a+(a+h)}{2}\right) + f\left(\frac{(a+h)+(a+2h)}{2}\right) + \cdots + f\left(\frac{(a+(n-1)h)+b}{2}\right)\right]$$

Zusammengesetzte Trapezregel:

$$T_1(h) = \frac{h}{2} * [f(a) + 2f(a+h) + \cdots + 2f(a+(n-1)h) + f(b)]$$

Zusammengesetzte Simpson

$$S_2(h) = \frac{h}{6} * \left[f(a) + 4f\left(\frac{a+(a+h)}{2}\right) + 2f(a+h) + \cdots + f(b)\right]$$

Praktikum W10

Gewöhnliche Differentialgleichungen

Numerische Lösung

Diskretes und kontinuierliches exponentielles Wachstum

Beispiel einer linearen Differentialgleichung: $x_{k+1} = a * x_k$

d.h.: $x_{k+1} - x_k = (a-1) * x_k$

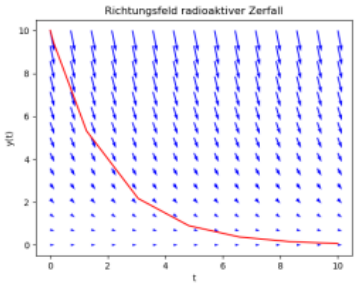
$\rightarrow$ *bei positivem Startwert* $x_0 > 0$ & $a > 0$: $a^x = e^{x*\ln(a)}$

also in diskretem Fall: exponentielles Wachstum so bald a > 1 (also ln(a) > 0)

Zerfallsgesetz und (explizites) Euler-Verfahren

- Radioaktiver Zerfall: Bestand verringert sich pro Zeiteinheit um einen festen Prozentsatz der Anzahl Teilchen $y'(t) = -a * y(t)$
- Lösung von DGL beschreibt für a > 0 exponentielle Abnahme eines positiven Bestands $y(t) = y(0) * e^{-a*t}$
- $y(t + h) = y(t) + h * (-a * y(t))$

Interpretation der numerischen Diskretisierung



DGL-Systeme

Zerfallskette

Aquivalenz von einer DGL höherer Ordnung / System erster Ordnung

$$y'' + a \cdot y' + b \cdot y = 0$$

1. $a^2 > 4b$: *Kriechfall*
2. $a^2 = 4b$: *aperiodischer Grenzfall*
3. $a^2 < 4b$: *Schwingfall*

Verwendung von Scipy-Verfahren

Automatische Wahl der Schrittweite

```
def zerfall( y0, T, n ):
    h = T/(n-1) # Schrittweite
    t = np.linspace( 0, T, n ) # Zeitpunkt-Gitter anlegen
    y = np.zeros( len(t) )
    y[0] = y0  # Startwert

    for k in range( len(t) - 1 ): # durchlaufe Gitterpunkte
        y[k+1] = y[k] + h*f(y[k]) # Euler-Verfahren: berechne neuen Punkt

    return t, y

    Aufruf:
      t, y = zerfall( y0, T, n )
    Inputs:
     y0: Startwert, Teilchenzahl zur Zeit t = 0
     T: maximaler Zeitpunkt
     n: Anzahl der Zeitschritte
    Outputs:
      t: Zeitschritte
      y: Vektor mit Loesung zu jedem Zeitschritt


# Unterfunktion mit rechter Seite von y'=f(y)
def f(y):
    return -a*y

a = 0.5 -> Je kleiner desto näher an der exakten Lösung
y0 = 100
T = 10
n = 50
t, y_numerisch = zerfall( y0, T, n )
y_exakt = y0*np.exp(-a*t)
plt.figure(0)
plt.plot( t, y_exakt, t, y_numerisch, 'xr ')
plt.legend(['exakt','numerisch'])
plt.xlabel('t')
plt.ylabel('y')
plt.title('Vergleich numerische und exakte Lsg')
plt.show()

Richtungsfeld
def f( t, yalt ):
    return -0.5*yalt

y0 = 10
t = np.linspace(0, 10, 15)
y = np.linspace(0, y0, 15)
dy = np.ones((len(y),len(t)))
dt = np.ones((len(y),len(t)))

for i in range ( len(y) ):
    for j in range ( len(t) ):
        dy[i,j] = f(t[i],y[i])
        dt[i,j] = 1

plt.quiver( t, y, dt, dy, color='b' )

# Benutzt anstelle von Euler ein scipy-Verfahren
result = integrate.solve_ivp(f, [0,10], [y0] )
tr = result['t']
yr = result['y'][0,:]

plt.figure(1)
plt.plot( tr, yr, 'r-' )
plt.xlabel('t')
plt.ylabel('y(t)')
plt.title( 'Richtungsfeld radioaktiver Zerfall' )
plt.show()
```