Methodische und Praktische Grundlagen der Informatik 4

Übungsblatt 7

mpgi4@cg.tu-berlin.de

WiSe 2013/2014

Teil 1: Polynominterpolation

Geben sei eine Menge von Punkten $\{(x_i, y_i) \in \mathbb{R}^2\}_{i=1,\dots,n}$ mit paarweise verschiedenen Stützstellen $x_i \neq x_j$ für $i \neq j$. Gesucht ist ein Polynom

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k = \sum_{j=0}^{k} a_j x^j \tag{1}$$

mit möglichst niedrigem Grad k, welches die gegeben Punkte interpoliert:

$$p(x_i) = y_i \quad \text{für } i = 1, \dots, n \tag{2}$$

Für einen fest gewählten Polynomgrad k definieren die Gleichung 2 ein Gleichungssystem:

$$\begin{aligned} a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_k x_1^k &= y_0 \\ a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_k x_2^k &= y_1 \\ \vdots \qquad\qquad\qquad\qquad & \quad \vdots \\ a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_k x_n^k &= y_n \end{aligned} \tag{3}$$

Dieses lässt sich auch in Matrixnotation schreiben:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_1^k \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^k \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{4}$$

Eine Matrix mit der speziellen Form

$$V(x_1, \dots, x_n) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_1^{n-1} \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix} \tag{5}$$

heißt *Vandermonde-Matrix*. Die Determinate einer Vandermonde-Matrix hat eine besonders einfache Darstellung:

$$\det V(x_1, \dots, x_n) = \prod_{1 \leq i < j \leq n} (x_j - x_i) \tag{6}$$

Wählt man also in Gleichung (4) für den Polynomgrad $k = n - 1$ und sind die x_i paarweise verschieden (dann ist nämlich $x_j - x_i \neq 0$ für $i \neq j$), so hat das Gleichungssystem (und damit auch das Interpolationsproblem) genau eine Lösung. Für die Praxis ist diese Methode allerdings nicht geeignet, da die Vandermonde-Matrix bei ungünstiger Lage der Stützstellen schlecht konditioniert und ihre Berechnung aufwendig ist.

```python
import numpy as np
import matplotlib.pyplot as plt

def intpoly(x, y):
    V = np.vander(x,len(x))
    a = np.linalg.solve(V,y)
    return np.poly1d(a)

points = [ np.array([[ 0., 0.25], [ 1., 0.75]]).T,
           np.array([[ 0., 0.25], [ 0.5, 0.33], [ 1., 0.75]]).T,
           np.array([[0., 0.25], [0.2, 0.33], [0.7, 0.66], [1., 0.75]]).T ]

for index, (x,y) in enumerate(points):
    p = intpoly(x,y)
    tx = np.linspace(-0.25, 1.25, 20)
    ty = p(tx)
    plt.subplot(1, len(points), index+1)
    plt.grid(True)
    plt.plot(x, y, 'ro')
    plt.plot(tx, ty, 'bx-')
    plt.xlim(-0.5,1.5)
    plt.ylim(-0.5,1.5)
    plt.gca().set_aspect('equal')

plt.show()
```

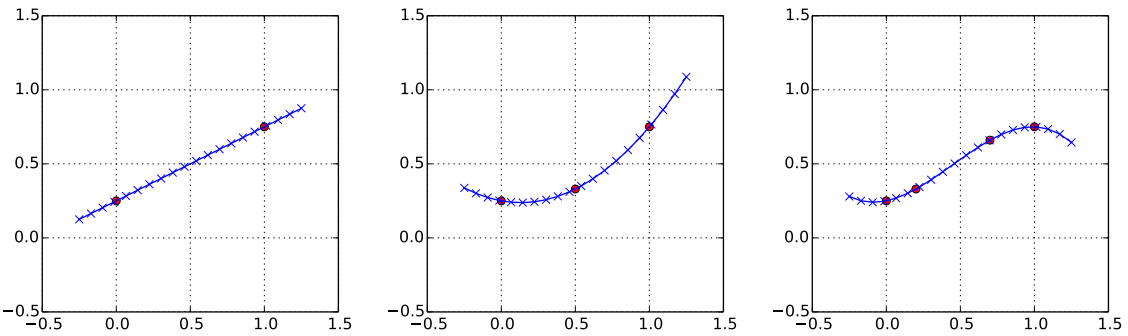

Abbildung 1: Berechnung des Interpolationspolynoms für für zwei, drei und vier gegebene Punkte mit Python.

Teil 2: Lagrangesche Interpolationsformel

Geben sei eine Menge von Punkten $\{(x_i, y_i) \in \mathbb{R}^2\}_{i=1,\dots,n}$ mit paarweise verschiedenen Stützstellen $x_i \neq x_j$ für $i \neq j$. Dann ist

$$p(x) = \sum_{j=1}^{n} l_j(x)\, y_j\,, \tag{7}$$

mit

$$l_j(x) = \frac{x - x_1}{x_j - x_1} \cdot \ldots \cdot \frac{x - x_{j-1}}{x_j - x_{j-1}} \cdot \frac{x - x_{j+1}}{x_j - x_{j+1}} \cdot \ldots \cdot \frac{x - x_n}{x_j - x_n} = \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}\,, \tag{8}$$

ein Polynom n-ten Grades, welches die gegebenen Punkte interpoliert.

Durch Ausmultiplizieren kann man sehen, dass $l_j(x)$ ein Polynom $(n-1)$-ten Grades ist. Da $p(x)$ eine gewichtete Summe der $l_j(x)$ ist folgt, dass auch $p(x)$ ein Polynom $(n-1)$-ten Grades ist. Es gilt für $i = j$

$$l_j(x_i) = \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x_i - x_k}{x_j - x_k} = \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x_j - x_k}{x_j - x_k} = 1 \tag{9}$$

und für $i \neq j$

$$l_j(x_i) = \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x_i - x_k}{x_j - x_k} = \frac{\overbrace{x_i - x_i}^{=0}}{x_j - x_i} \cdot \prod_{\substack{k=1 \\ k \neq i,j}}^{n} \frac{x_i - x_k}{x_j - x_k} = 0 \tag{10}$$

```python
import numpy as np
import matplotlib.pyplot as plt

def lagrange_basis(x):
    l = []
    for j in range(len(x)):
        lj = 1.0
        for k, xk in enumerate(x):
            if k != j:
                lj *= np.poly1d([1.0, -xk])
                lj /= x[j] - xk
        l.append(lj)
    return l

def lagrange_poly(l, y):
    assert(len(l) == len(y))
    p = 0
    for (lj, yj) in zip(l,y):
        p += lj * yj
    return p

points = [ np.array([[ 0., 0.25], [ 1., 0.75]]).T,
           np.array([[ 0., 0.25], [ 0.5, 0.33], [ 1., 0.75]]).T,
           np.array([[0., 0.25], [0.2, 0.33], [0.7, 0.66], [1., 0.75]]).T ]

for index, (x,y) in enumerate(points):
    n = len(points)
    l = lagrange_basis(x)
    p = lagrange_poly(l, y)
    tx = np.linspace(-0.25, 1.25, 50)
    ty = p(tx)

    plt.subplot(2, n, index+1)
    plt.grid(True)
    plt.plot(tx, ty, '-', color='0.5')
    for (xi,yi) in zip(x,y): plt.plot(xi, yi, 'o')
    plt.xlim(-0.5,1.5)
    plt.ylim(-0.5,1.5)
    plt.gca().set_aspect('equal')

    plt.subplot(2, n, n+index+1)
    plt.grid(True)
    for lj in l: plt.plot(tx, lj(tx), '-')
    plt.xlim(-0.5,1.5)
    plt.ylim(-0.5,1.5)
    plt.gca().set_aspect('equal')

plt.show()
```

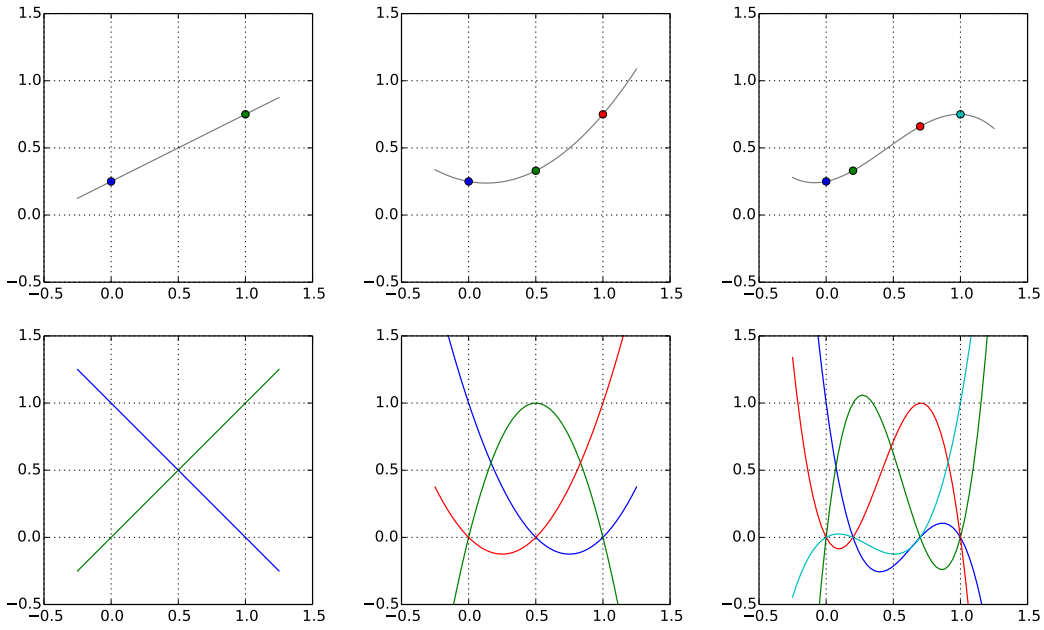

Abbildung 2: Berechnung des Interpolationspolynoms mit der Lagrangesche Interpolationsformel für zwei, drei und vier gegebene Punkte mit Python. Die unteren Plots zeigen die Lagrange Basisfunktionen l_j der jeweiligen Stützstellen.

```python
import numpy as np
import matplotlib.pyplot as plt

def intpoly(x, y):
    V = np.vander(x,len(x))
    a = np.linalg.solve(V,y)
    return np.poly1d(a)

x = np.linspace(0, 4.0, 13)
y = np.sin(x * np.pi * 2.5) * np.exp(-x**2/8.0)

for k in range(1,4):
    plt.subplot(1,3,k)
    plt.plot(x, y, 'ko', zorder=0)
    for i in range(0, len(x)-k, k):
        p = intpoly(x[i:i+k+1], y[i:i+k+1])
        tx = np.linspace(x[i], x[i+k], 20)
        ty = p(tx)
        plt.plot(tx, ty, '-')

    plt.grid(True)
    plt.xlim(-0.1,4.1)
    plt.ylim(-1.1,1.1)
    plt.gca().set_aspect('equal')

plt.show()
```

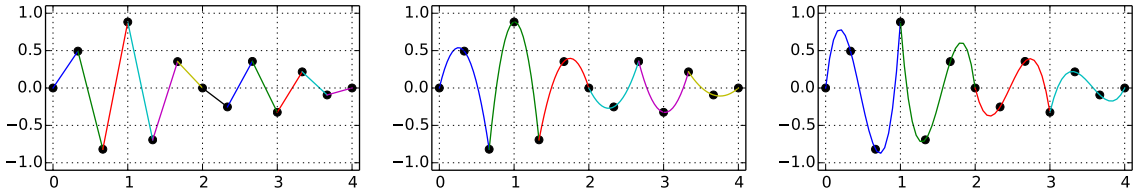

Abbildung 3: Stückweise lineare, quadratische und kubische Polynominterpolation.

Folglich ist also

$$l_j(x_i) = \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{x_i - x_k}{x_j - x_k} = \delta_{ij} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases}, \tag{11}$$

und daher

$$p(x_i) = \sum_{j=1}^{n} l_j(x_i) y_j = \underbrace{l_i(x_i)}_{=1} y_i + \sum_{\substack{j=1 \\ j\neq i}}^{n} \underbrace{l_j(x_i)}_{=0} y_j = y_i \tag{12}$$

für $i = 1, \ldots, n$. Das Polynom $p(x)$ interpoliert somit die Werte y_i an den Stützstellen x_i.

Teil 3: Stückweise Polynominterpolation

Sind n Datenpunkte gegeben, so beträgt der maximale Grad des Interpolationspolynoms $n - 1$. Zwei Datenpunkte können zum Beispiel mittels einer linearen Funktion, drei Datenpunkte mittels eines quadratischen Polynoms und vier Datenpunkte mittels eines kubischen Polynoms interpoliert werden. Ist eine größere Anzahl von Datenpunkten gegeben, so liefert das Interpolationspolynom in der Regel keine geeignete Interpolationsfunktion der Datenpunkte, da starke Oszillationen auftreten (vgl. Hausaufgabe).

Eine einfache Möglichkeit eine größere Anzahl Datenpunkte zu interpolieren ist die stückweise Polynominterpolation. Dabei werden die Datenpunkte in einzelne Abschnitte unterteilt und dann die Datenpunkte von jedem Abschnitt mittels Polynominterpolation interpoliert (Abbildung 3). Auf diese Weise erhält man eine stetige Interpolationsfunktion, die jedoch in der Regel nur stückweise glatt ist. Dies liegt daran, dass die Interpolationspolynome zweier benachbarter Abschnitte zwar am Ende des einen und am Start des anderen übereinstimmen, dort normaler Weise jedoch unterschiedliche Steigungen haben. Interessant wäre es also an den Stützstellen neben den Werten auch die Steigungen vorgeben zu können.

```python
import numpy as np
import matplotlib.pyplot as plt

def pcubic(x, y):
    A = np.array([
        [ x[0]**3, x[0]**2, x[0],  1. ],
        [ x[1]**3, x[1]**2, x[1],  1. ],
        [ 3*x[0]**2, 2*x[0], 1., 0. ],
        [ 3*x[1]**2, 2*x[1], 1., 0. ]
    ])
    a = np.linalg.solve(A, y)
    return np.poly1d(a)

for dy in np.linspace(-2.0, 2.0, 5):
    x = np.array([0., 1.])
    y = np.array([0.75, -0.25, dy, -0.25])
    p = pcubic(x, y)
    tx = np.linspace(-1.0, 2.0, 100)
    ty = p(tx)
    plt.plot(x, y[:2], 'ko', zorder=0)
    plt.plot(tx, ty, '-')

plt.grid(True)
plt.xlim(-0.5,1.5)
plt.ylim(-0.5,1.5)
plt.gca().set_aspect('equal')

plt.show()
```

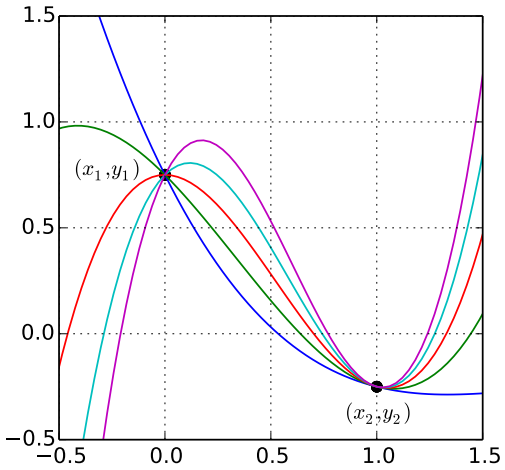

Abbildung 4: Kubische Interpolationspolynome für zwei Stützstellen mit vorgegebenen Werten und Steigungen. In der ersten Stützstelle wurde die Steigungen variiert.

Teil 3.1: Kubische Hermiteinterpolation

Dies wollen wir uns nun an einem einfachen Beispiel anschauen (Abbildung 4). Gegeben seinen zwei Datenpunkte (x_1, y_1) und (x_2, y_2) mit vorgegeben Steigungen y_1', y_2'. Gesucht ist ein kubisches Polynom

$$p(x) = ax^3 + bx^2 + cx + d\,, \tag{13}$$

welches an den Stützstellen x_1, x_2 die Funktionswerte y_1, y_2 und Steigungen y_1', y_2' hat. Die Steigung von p erhalten wir durch Ableiten und ist gegeben durch:

$$p'(x) = \frac{\mathrm{d}p}{\mathrm{d}x} = 3ax^2 + 2bx + c \tag{14}$$

Zusammengefasst erhalten wir vier Gleichungen:

$$\begin{aligned} p(x_1) &= ax_1^3 + bx_1^2 + cx_1 + d = y_1 \\ p(x_2) &= ax_2^3 + bx_2^2 + cx_2 + d = y_2 \\ p'(x_1) &= 3ax_1^2 2bx_1^2 + c = y_1' \\ p'(x_1) &= 3ax_2^2 2bx_2^2 + c = y_2' \end{aligned} \tag{15}$$

Oder in Matrixnotation:

$$\begin{pmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ 3x_1^2 & 2x_1 & 1 & 0 \\ 3x_2^2 & 2x_2 & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_1' \\ y_2' \end{pmatrix} \tag{16}$$

Teil 3.2: Quadratischer Spline mit periodischen Randbedingungen

Oft kennt man allerdings die Steigungen in den Datenpunkten nicht. Anstatt jeweils für jeden Abschnitt das Interpolationspolynom unabhängig von den anderen Abschnitten zu bestimmen, wollen wir nun quadratische Interpolationspolynome

$$p_j(x) = a_j x^2 + b_j x + c_j, \quad j = 1, \ldots, n-1, \tag{17}$$

für eine gegebene Menge von Datenpunkten $\{(x_i, y_i)\}_{i=1,\ldots,n}$ bestimmen, welche an den inneren Stützstellen die gleiche Steigung haben (Abbildung 5). Dies ergibt dann die folgenden Interpolationsbedingungen:

$$\begin{aligned} p_j(x_j) &= y_j && \text{für } j = 1, \ldots, n-1 \\ p_j(x_{j+1}) &= y_{j+1} && \text{für } j = 1, \ldots, n-1 \\ p_j'(x_{j+1}) &= p_{j+1}'(x_{j+1}) && \text{für } j = 1, \ldots, n-2 \end{aligned} \tag{18}$$

Dabei bezeichnet p_j' die erste Ableitung von p_j, welche gegeben ist durch:

$$p_j'(x) = 2a_j x + b_j \tag{19}$$

Dies sind $3(n-1)-1$ Gleichungen, jedoch mit $3(n-1)$ Unbekannten. Unter der Annahme $y_1 = y_n$ wollen wir daher als zusätzliche Bedingung fordern, dass die Steigung auch an der ersten und letzten Stützstelle übereinstimmt:

$$p_1'(x_1) = p_{n-1}'(x_n) \tag{20}$$

Zusammengefasst erhalten wir die folgenden $3(n-1)$ Gleichungen mit $3(n-1)$ Unbekannten:

$$\begin{aligned} p_j(x_j) &= a_j x_j^2 + b_j x_j + c_j &&= y_j \\ p_j(x_{j+1}) &= a_j x_{j+1}^2 + b_j x_{j+1} + c_j &&= y_{j+1} \\ p_j'(x_{j+1}) - p_{j+1}'(x_{j+1}) &= 2a_j x_{j+1} + b_j - 2a_{j+1} x_{j+1} + b_{j+1} &&= 0 \\ p_1'(x_1) - p_{n-1}'(x_n) &= 2a_1 x_1 + b_1 - 2a_{n-1} x_n - b_{n-1} &&= 0 \end{aligned} \tag{21}$$

In Matrixnotation erhalten wir:

$$\begin{pmatrix} x_1^2 & x_1 & 1 & & & & & & & & & \\ x_2^2 & x_2 & 1 & & & & & & & & & \\ 2x_2 & 1 & 0 & -2x_2 & -1 & 0 & & & & & & \\ & & & x_2^2 & x_2 & 1 & & & & & & \\ & & & x_3^2 & x_3 & 1 & & & & & & \\ & & & 2x_3 & 1 & 0 & -2x_3 & -1 & 0 & & & \\ & & & & & & & & \ddots & & & \\ & & & & & & & & & x_{n-1}^2 & x_{n-1} & 1 \\ & & & & & & & & & x_n^2 & x_n & 1 \\ 2x_1 & 1 & 0 & & & & & & & -2x_n & -1 & 0 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ \vdots \\ b_{n-2} \\ c_{n-2} \\ a_{n-1} \\ b_{n-1} \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ 0 \\ y_2 \\ y_3 \\ 0 \\ \vdots \\ y_{n-1} \\ y_n \\ 0 \end{pmatrix} \tag{22}$$

Teil 3.3: Kubischer Spline

Als weiteres Beispiel wollen wir nun kubische Interpolationspolynome

$$p_j(x) = a_j x^3 + b_j x^2 + c_j x + d_j, \quad j = 1, \ldots, n-1, \tag{23}$$

für eine gegebene Menge von Datenpunkten $\{(x_i, y_i)\}_{i=1,\ldots,n}$ bestimmen, welche an den inneren Stützstellen in erster und zweiter Ableitung übereinstimmen (Abbildung 6). Dies ergibt die folgenden Interpolationsbedingungen:

$$\begin{aligned} p_j(x_j) &= y_j && \text{für } j = 1, \ldots, n-1 \\ p_j(x_{j+1}) &= y_{j+1} && \text{für } j = 1, \ldots, n-1 \\ p_j'(x_{j+1}) &= p_{j+1}'(x_{j+1}) && \text{für } j = 1, \ldots, n-2 \\ p_j''(x_{j+1}) &= p_{j+1}''(x_{j+1}) && \text{für } j = 1, \ldots, n-2 \end{aligned} \tag{24}$$

```
import numpy as np
import matplotlib.pyplot as plt

n = 8
x1 = 0.
xn = 4*np.pi
rx = np.linspace(x1, xn, 100)
ry = np.sin(rx)
plt.plot(rx, ry, '--', color='0.4', zorder=0)

x = np.linspace(x1, xn, n)
y = np.sin(x)
plt.plot(x, y, 'ko')

A = np.zeros((3*(n-1), 3*(n-1)))
b = np.zeros((3*(n-1),))

for i in range(n-1):
    k = 3*i
    A[k:k+2, k:k+3] = np.array([
        [ x[i]**2, x[i], 1. ],
        [ x[i+1]**2, x[i+1], 1. ]])
    if i < n-2:
        A[k+2, k:k+6] = np.array([2.*x[i+1], 1., 0., -2.*x[i+1], -1., 0.])
    b[k+0] = y[i]
    b[k+1] = y[i+1]
A[-1,0:3] = np.array([2.*x[0], 1., 0.])
A[-1,-3:] = np.array([-2.*x[-1], -1., 0.])

c = np.linalg.solve(A,b)
P = []
for i in range(n-1):
    p = np.poly1d(c[i*3:(i+1)*3])
    P.append(p)
    tx = np.linspace(x[i], x[i+1], 20)
    ty = p(tx)
    line = plt.plot(tx, ty, '-')
    plt.plot(tx+xn, ty, '-', color=line[0].get_color())

plt.grid(True)
plt.xlim(x1-0.2,2*xn+0.2)
plt.ylim(-1.3,1.3)
plt.subplots_adjust(left=0.05, right=0.98, top=0.98, bottom=0.05)
plt.show()
```

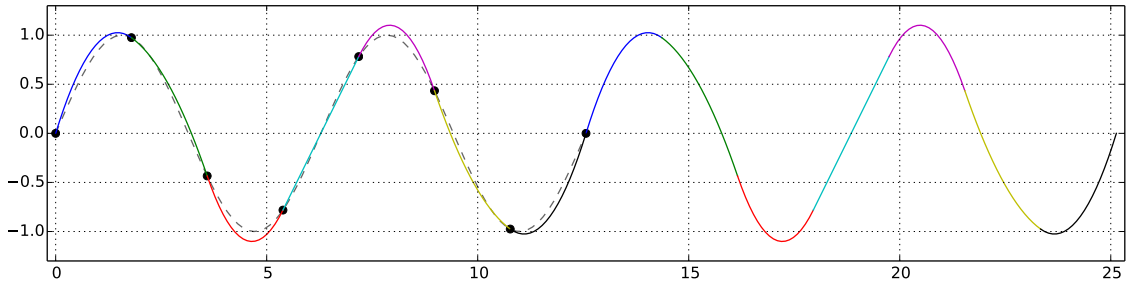

Abbildung 5: Einmal stetig differenzierbare stückweise quadratische Polynominterpolation mit periodischen Randbedingungen.

```python
import numpy as np
import matplotlib.pyplot as plt

np.set_printoptions(precision=3, linewidth=256, threshold=50, edgeitems=20)

n = 13
x = np.linspace(0, 4.0, n)
y = np.sin(x * np.pi * 2.5) * np.exp(-x**2/8.0)

A = np.zeros((4*(n-1), 4*(n-1)))
b = np.zeros((4*(n-1),))
for i in range(n-1):
    k = 4*i
    A[k:k+2, k:k+4] = np.array([
        [ x[i]**3, x[i]**2, x[i], 1. ],
        [ x[i+1]**3, x[i+1]**2, x[i+1], 1. ]])
    if i < n-2:
        A[k+2:k+4, k:k+8] = np.array([
            [ 3.*x[i+1]**2, 2.*x[i+1], 1., 0., -3.*x[i+1]**2, -2.*x[i+1], -1., 0. ],
            [ 6*x[i+1], 2., 0., 0., -6.*x[i+1], -2., 0., 0. ]])
    b[k+0] = y[i]
    b[k+1] = y[i+1]
A[4*(n-1)-2,0:4] = np.array([6*x[0], 2., 0., 0.])
A[4*(n-1)-1,4*(n-2):4*(n-1)] = np.array([6*x[n-1], 2., 0., 0.])

c = np.linalg.solve(A,b)
P = []
for i in range(n-1):
    p = np.poly1d(c[i*4:(i+1)*4])
    P.append(p)
    tx = np.linspace(x[i], x[i+1], 20)
    ty = p(tx)
    plt.plot(tx, ty, '-')

plt.plot(x, y, 'ko', zorder=0)
plt.grid(True)
plt.xlim(-0.1,4.1)
plt.ylim(-1.1,1.5)
plt.gca().set_aspect('equal')

plt.show()
```

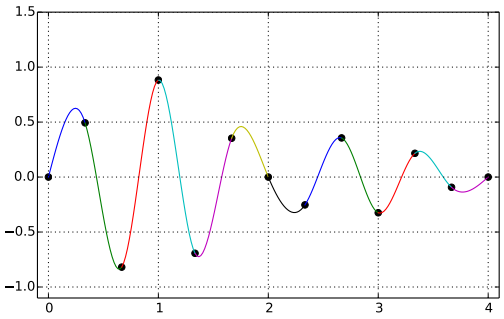

Abbildung 6: Zweimal stetig differenzierbare stückweise kubische Polynominterpolation.

Dabei bezeichnen p'_j und p''_j die ersten und zweiten Ableitungen von p_j, welche gegeben sind durch:

$$p'_j(x) = 3a_j x^2 + 2b_j x + c_j \tag{25}$$
$$p''_j(x) = 6a_j x + 2b_j \tag{26}$$

Zusammengefasst erhalten wir die folgenden $4(n-1)-2$ Gleichungen mit $4(n-1)$ Unbekannten:

$$\begin{aligned}
p_j(x_j) &= a_j x_j^3 + b_j x_j^2 + c_j x_j + d_j &&= y_j \\
p_j(x_{j+1}) &= a_j x_{j+1}^3 + b_j x_{j+1}^2 + c_j x_{j+1} + d_j &&= y_{j+1} \\
p'_j(x_{j+1}) - p'_{j+1}(x_{j+1}) &= 3a_j x_{j+1}^2 + 2b_j x_{j+1} + c_j - 3a_{j+1} x_{j+1}^2 - 2b_{j+1} x_{j+1} - c_{j+1} &&= 0 \\
p''_j(x_{j+1}) - p''_{j+1}(x_{j+1}) &= 6a_j x_{j+1} + 2b_j - 6a_{j+1} x_{j+1} - 2b_{j+1} &&= 0
\end{aligned} \tag{27}$$

Dieses Gleichungssystem ist unterbestimmt. Für eine eindeutige Lösung fehlen zwei Gleichungen. Diese kann man durch Randbedingungen erhalten. Man unterscheidet hierbei drei verschiedene Ansätze:

a) **Natürliche Randbedingungen:** Die zweiten Ableitungen am Anfangs- und Endpunkt des Splines werden Null gesetzt. Dies hat zur Folge, dass der Spline eine minimale Gesamtkrümmung hat:

$$p_1''(x_1) = p_{n-1}''(x_n) = 0 \tag{28}$$

b) **Vorgabe der Steigungen:** Die ersten Ableitungen am Anfangs- und Endpunkt des Splines werden vorgegeben:

$$p_1'(x_1) = y_1' \qquad p_{n-1}'(x_n) = y_n' \tag{29}$$

c) **Periodizitätsforderung:** Die Funktionswerte, ersten Ableitungen und zweiten Ableitungen sollen am Anfangs- und Endpunkt des Splines übereinstimmen:

$$p_1(x_1) = p_{n-1}(x_n) \qquad p_1'(x_1) = p_{n-1}'(x_n) \qquad p_1''(x_1) = p_{n-1}''(x_n) \tag{30}$$

Gleichungen (27) und (28) ergeben dann zusammen in Matrixnotation:

$$\begin{pmatrix}
x_1^3 & x_1^2 & x_1 & 1 & & & & & & & & \\
x_2^3 & x_2^2 & x_2 & 1 & & & & & & & & \\
3x_2^2 & 2x_2 & 1 & 0 & -3x_2^2 & -2x_2 & -1 & 0 & & & & \\
6x_2 & 2 & 0 & 0 & -6x_2 & -2 & 0 & 0 & & & & \\
 & & & & x_2^3 & x_2^2 & x_2 & 1 & & & & \\
 & & & & x_3^3 & x_3^2 & x_3 & 1 & & & & \\
 & & & & & & & & \ddots & & & \\
 & & & & & & & & x_{n-1}^3 & x_{n-1}^2 & x_{n-1} & 1 \\
 & & & & & & & & x_n^3 & x_n^2 & x_n & 1 \\
6x_1 & 2 & 0 & 0 & & & & & & & & \\
 & & & & & & & & 6x_n & 2 & 0 & 0
\end{pmatrix}
\begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ \\ \vdots \\ \\ a_{n-1} \\ b_{n-1} \\ c_{n-1} \\ d_{n-1} \end{pmatrix}
=
\begin{pmatrix} y_1 \\ y_2 \\ 0 \\ 0 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \\ 0 \\ 0 \end{pmatrix} \tag{31}$$