Solutions to Exercises to

Programming Methods in Scientific Computing

David Bauer
Jendrik Stelzner

Letzte Änderung: 21. November 2017

Chapter 3: Python, the Fundamentals

Exercise 3.2

Wir weitern die gegeben Klasse `Polynomial` um Methoden zum Ableiten, sowie Bilden einer Stammfunktion. Dabei wählen wir die „Integrationskonstante“ als 0.

```python
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        l = []
        if len(self.coeff) > len(other.coeff):
            l += self.coeff
            for i in range(len(other.coeff)):
                l[i] += other.coeff[i]
        else:
            l += other.coeff
            for i in range(len(self.coeff)):
                l[i] += self.coeff[i]
        return Polynomial(l)

    def __eq__(self, other):
        return self.coeff == other.coeff

    def derivative(self):
        coeff = []
        for i in range(1,len(self.coeff)):
            coeff.append(i * self.coeff[i])
        return Polynomial(coeff)

    def antiderivative(self):
        coeff = [0]
        for i in range(len(self.coeff)):
            coeff.append(self.coeff[i]/(i+1))
        return Polynomial(coeff)
```

Für das gegebene Polynom $p(x) = 3x^2+2x+1$ erhalten wir die folgenden Ergebnisse:

```python
    >>> p = Polynomial([1,2,3])
    >>> p.derivative().coeff
    [2, 6]
```

```
    >>> p.antiderivative().coeff
    [0, 1.0, 1.0, 1.0]
    >>> p.antiderivative().derivative().coeff
    [1.0, 2.0, 3.0]
```

Exercise 3.3

```
class Matrix():
    def __init__(self, entries):
        m = len(entries)
        if m == 0:
            raise ValueError("height must be positive")
        n = len(entries[0])
        if n == 0:
            raise ValueError("width must be positive")
        for i in range(1, m):
            if len(entries[i]) != n:
                raise ValueError("rows must have the same width")
        self.height = m
        self.width = n
        self.entries = entries

    def __getitem__(self, i):       # allows to get the rows via A[i]
        return self.entries[i]

    def __setitem__(self, i, k):    # allows to set rows via A[i]
        self.entries[i] = k

    def __str__(self):              # allows print(A) for a Matrix A
        rows = ["["]*self.height
        for j in range(self.width): # construct output columnwise, align left
            numbers = []            # numbers to appear in column j
            maxlen = 0              # maximal length of a number in column j
            for i in range(self.height):
                s = str(self[i][j])
                numbers.append(s)
                if len(s) > maxlen:
                    maxlen = len(s)
            for i in range(self.height):
                # pad the entries if they are too short
                rows[i] += numbers[i] + " "*(maxlen-len(numbers[i])) + " "
        s = ""
        for r in rows:
            s += r[:-1] + "]\n" # remove white space at the end of ech line
        s = s[:-1]              # remove empty line at the end
        return s

    def __mul__(self, other):
        if self.width != other.height:
            raise TypeError('matrix dimensions do not match')
        newentries = []
        for i in range(self.height):
            row = []
```

```
            for j in range(other.width):
                s = self[i][0] * other[0][j]    # makes s have the right type
                for k in range(1, self.width):
                    s += self[i][k] * other[k][j]
                row.append(s)
            newentries.append(row)
        return Matrix(newentries)

    def __eq__(self, other):
        if self.height != other.height or self.width != other.width:
                return False
        for i in range(self.height):
            for j in range(self.width):
                if self[i][j] != other[i][j]:
                    return False
        return True
```

Für die Matrixassoziativität erhalten wir das Folgende:

```
    >>> A = Matrix([[0,1],[1,0],[1,1]])
    >>> B = Matrix([[1,2,3,4],[5,6,7,8]])
    >>> C = Matrix([[1,0],[0,1],[1,0],[0,1]])
    >>> print(A * (B * C) == (A * B) * C)
    True
```

Exercise 3.4

Wir schreiben zunächst eine Klasse `Rational`, die ein genaues Rechnen mit rationalen Zahlen erlaubt.

```
class Rational():
    def __init__(self, num, denum = 1):   # default denumerator is 1
        if type(num) != int:
            raise TypeError("numerator is no integer")
        if type(denum) != int:
            raise TypeError("denumerater is no integer")
        if denum == 0:
            raise ZeroDivisionError("denumerator is zero")
        self.num = num
        self.denum = denum

    def __str__(self):    # allows print(x) for Rational x
        return "{}/{}".format(self.num, self.denum)

    def __add__(self, other):
        return Rational( self.num * other.denum + self.denum * other.num,
            self.denum * other.denum )

    def __sub__(self, other):
        return Rational( self.num * other.denum - self.denum * other.num,
            self.denum * other.num )

    def __neg__(self):
        return Rational( -self.num, self.denum )
```

```

    def __mul__(self, other):
        return Rational( self.num * other.num, self.denum * other.denum )

    def __truediv__(self, other):
        if other.num == 0:
            raise ZeroDivisionError("division by zero")
        return Rational( self.num * other.denum, self.denum * other.num)

    def inverse(self):  # short hand notation
        return Rational(1)/self

    def __eq__(self, other):
        if type(other) == int:  # allows comparison to int, used for x == 0
            return (self == Rational(other))
        return (self.num * other.denum == self.denum * other.num)

    def __float__(self):
        return self.num / self.denum
```

Wir erweitern nun die bisherige `Matrix`-Klasse um weitere Methoden. (Aus Platzgründen kopieren wir den bereits vorhandenen Code nicht noch einmal.)

```
    def mapentries(self, f):       # applies a function to all entries
        A = zeromatrix(self.height, self.width) # zeromatrix is defined below
        for i in range(self.height):
            for j in range(self.width):
                A[i][j] = f(self[i][j])
        return A

    def addrow(self, i, j, c=1):    # add c times row j to row i
        for k in range(self.width):
            self[i][k] = self[i][k] + self[j][k] * c

    def addcolumn(self, i, j, c=1): # add c times column j from colunm i
        for k in range(self.height):
            self[k][i] = self[k][i] + self[k][j] * c
```

Wir nutzen im Folgenden auch einige Hilfsfunktionen, um den Umgang mit Matrizen zu erleichtern. (Dies sind keine zusätzlichen Methoden der Klasse `Matrix`.)

```
def zeromatrix(height, width):  # creates a zero matrix
    entries = []
    for i in range(height):
        entries.append([0]*width)
    return Matrix(entries)

def identitymatrix(size):       # creates an identiy matrix
    E = zeromatrix(size, size)
    for i in range(size):
        E[i][i] = 1
    return E

def copymatrix(A):              # forcefully copies a matrix
    B = zeromatrix(A.height, A.width)
    for i in range(A.height):
```

```
        for j in range(A.width):
            B[i][j] = A[i][j]
    return B
```

Die LU-Zerlegung von passenden Matrix mit ganzzahligen Einträgen kann nun dem folgenden naiven Algorithmus berechnet werden:

```
def naive_lu(A):  # expects integer valued matrix as input
    if A.height != A.width:
        raise ValueError("matrix is not square")
    U = copymatrix(A)   # circumvent pass by reference
    n = U.height
    L = identitymatrix(n)
    U = U.mapentries(Rational)   # make all
    L = L.mapentries(Rational)   # entries rational
    # bring U in upper triangular form, change L such that always LU = A
    for j in range(n):
        if U[j][j] == 0:
            raise ValueError("algorithm does not work for this matrix")
        for i in range(j+1,n):
            L.addcolumn(j, i,  U[i][j]/U[j][j]) # important: change L first
            U.addrow(i, j, -U[i][j]/U[j][j])
    return (L,U)
```

Für die gegebene Matrix

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 6 & 3 \\ 9 & 10 & 6 \end{pmatrix}$$

erhalten wir das folgende Ergebnis:

```
>>> A = Matrix([[3,2,1],[6,6,3],[9,10,6]])
>>> (L,U) = naive_lu(A)
>>> print(L)
[9/9  0/18  0/1]
[18/9 18/18 0/1]
[27/9 36/18 1/1]
>>> print(U)
[3/1   2/1   1/1    ]
[0/3   6/3   3/3    ]
[0/162 0/162 162/162]
>>> print( (L*U).mapentries(float) )
[3.0 2.0  1.0]
[6.0 6.0  3.0]
[9.0 10.0 6.0]
```

Für die gegebene Matrix

$$B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(welche keine LU-Zerlegung besitzt) bricht das Programm mit einem Fehler ab.

```
>>> B = Matrix([[0,1],[1,0]])
>>> (L,U) == naive_lu(B)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in naive_lu
ValueError: algorithm does not work for this matrix
```

Exercise 3.5

Wir erweitern die Klasse `Matrix` um eine Methode zum Berechnen der Transponierten, um im Folgenden die Ergebnisse überprüfen zu können.

```
    def transpose(self):
        T = zeromatrix(self.width, self.height)
        for i in range(self.height):
            for j in range(self.width):
                T[j][i] = self[i][j]
        return T
```

Wir bestimmen die Cholesky-Matrix eintragsweise.

```
def cholesky(A):
    from math import sqrt
    if A.height != A.width:
        raise ValueError("matrix is not square")
    B = copymatrix(A)
    n = B.height
    L = zeromatrix(n,n)
    for i in range(n):
        rowsum = 0
        for j in range(i):
            s = 0
            for k in range(j):
                s += L[i][k] * L[j][k]
            L[i][j] = (B[i][j] - s)/L[j][j]
            rowsum += L[i][j]**2
        L[i][i] = sqrt( B[i][i] - rowsum )
    return L
```

Für die gegebene Matrix

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 10 \end{pmatrix}$$

erhalten wir das folgende Ergebnis:

```
>>> A = Matrix([[1,2,1],[2,5,2],[1,2,10]])
>>> L = cholesky(A)
>>> print(L)
[1.0 0   0  ]
[2.0 1.0 0  ]
[1.0 0.0 3.0]
>>> print(L * L.transpose())
[1.0 2.0 1.0 ]
[2.0 5.0 2.0 ]
[1.0 2.0 10.0]
```

Für die gegebene Matrix

$$B = \begin{pmatrix} 1.01 \cdot 10^{-2} & 0.705 & 1.42 \cdot 10^{-2} \\ 0.705 & 49.5 & 1 \\ 1.42 \cdot 10^{-2} & 1 & 1 \end{pmatrix}$$

erhalten wir das folgende Ergebnis:

```
>>> B = Matrix([[1.01E-2, 0.705, 1.42E-2],[0.705,49.5,1],[1.42E-2,1,1]])
>>> L = cholesky(B)
>>> print(L)
[0.1004987562112089  0                     0                 ]
[7.015012190980423   0.5381486415443629   0                 ]
[0.14129528100981847 0.016374437298272527 0.9898320672556135]
>>> print(L * L.transpose())
[0.010100000000000001 0.705 0.014200000000000003]
[0.705                49.5  1.0                 ]
[0.014200000000000003 1.0   1.0                 ]
```

Exercise 3.6

Wir nutzen die Klassen `Rational` und `Matrix` aus Exercise 3.4, sowie die dort definierten Hilfsfunktionen. Zudem erweitern wir die Klasse `Matrix` um zwei weitere Methoden zum Durchführen elementarer Zeilenumformungen.

```
    def multrow(self, i, c):        # multiply row i with c
        for j in range(self.width):
            self[i][j] *= c

    def swaprows(self, i, j):       # swap rows i and j
        if i > self.height or j > self.height:
            raise ValueError("swap nonexistent rows")
        l = self[i]
        self[i] = self[j]
        self[j] = l
```

Mithilfe elementarer Zeilenumformungen lässt sich nun der Gauß-Algorithmus zum Invertieren von Matrizen implementieren.

```
def invert(A):      # allowed input are integer matrices
    if A.height != A.width:
        raise ValueError("matrix is not square")
    B = copymatrix(A)   # circumvent pass by reference
    n = B.height
    Inv = identitymatrix(n)
    B = B.mapentries(Rational)      # make all
    Inv = Inv.mapentries(Rational)  # entries rational
    # bring B in lower triangular form
    for j in range(n):
        p = -1
        for i in range(j,n):
            if B[i][j] != 0:
```

```
                p = i
                break
        if p == -1:
            raise ZeroDivisionError("matrix is not invertible")
        for i in range(p+1,n):
            Inv.addrow(i, p, -B[i][j]/B[p][j])  # import: change inverse first
            B.addrow(i, p, -B[i][j]/B[p][j])
    # norm the diagonal entries
    for i in range(n):
        Inv.multrow(i, B[i][i].inverse())
        B.multrow(i, B[i][i].inverse())
    # bring B into identity form
    for j in range(n):
        for i in range(j):
            Inv.addrow(i, j, -B[i][j])
            B.addrow(i, j, -B[i][j])
    return Inv
```

Für die gegebene Matrix

$$A = \begin{pmatrix} 3 & -1 & 2 \\ -3 & 4 & -1 \\ -6 & 5 & -2 \end{pmatrix}$$

erhalten wir das folgende Ergebnis:

```
>>> A = Matrix([[3,-1,2],[-3,4,-1],[-6,5,-2]])
>>> B = invert(A)
>>> A = A.mapentries(Rational)
>>> print(A*B == identitymatrix(3))
True
>>> print(B.mapentries(float))
[-0.3333333333333333 0.8888888888888888 -0.7777777777777778]
[0.0                 0.6666666666666666 -0.3333333333333333]
[1.0                 -1.0               1.0                ]
```

Exercise 3.8

(1)

Alle notwendigen Funktionswerte werden zunächst berechnet und in einer Liste gespeichert, um mehrfaches Berechnen des gleichen Funktionswertes zu umgehen.

```
def trapeze(f,a,b,n):
    values = [f(a + (k/n)*(b-a)) for k in range(n+1)]
    integral = 0
    for i in range(len(values)-1):
        integral += values[i] + values[i+1]
    integral = (b-a)*integral/n/2
    return integral
```

(2)

Damit erhalten wir für $\int_0^\pi \sin(x)\,\mathrm{d}x$ die folgende Approximation:

```
>>> from math import sin, pi
>>> n = 1
>>> s = 0
>>> while 2 - s >= 1.E-6:   # sin is concave on [0,pi] -> estimate too small
...     n += 1              # can skip n = 1 because it results in 0
...     s = trapeze(sin, 0, pi, n)
...

>>> print(s)
1.9999990007015205
```

Exercise 3.9

(1)

Wir definieren eine neue Funktion `Trapeze`:

```
def trapeze(f, a, b, mmax):
    integrals = []              # list of the approximations
    values = [f(a), f(b)]       # list of the calculated values
    for m in range(1,mmax+1):
        n = 2**m
        for k in range(1,n,2):
            values.insert(k, f(a + (k/n)*(b-a)))    # add new values
        integral = 0
        for i in range(len(values)-1):
            integral += values[i] + values[i+1]
        integral = (b-a)*integral/n/2
        integrals.append(integral)                  # add new approx.
    return integrals
```

Wir berechnen die Werte für $m = 1, \ldots, 10$, also für $m_{\max} = 10$, und geben diese, zusammen mit der jeweiligen Abweichung vom exakten Ergebnis, in Tabellenform aus:

```
>>> from math import sin, pi
>>> m = 10
>>> results = trapeze(sin, 0, pi, m)
>>> for i in range(m):
...     print("m={:2d}\t{}\t{:.20f}".format(i+1, results[i], 2-results[i]))
...
m= 1    1.5707963267948966      0.42920367320510344200
m= 2    1.8961188979370398      0.10388110206296019555
m= 3    1.9742316019455508      0.02576839805444919307
m= 4    1.9935703437723395      0.00642965622766045186
m= 5    1.9983933609701445      0.00160663902985547224
m= 6    1.9995983886400386      0.00040161135996141795
m= 7    1.9998996001842035      0.00010039981579645918
m= 8    1.9999749002350518      0.00002509976494824429
m= 9    1.9999937250705773      0.00000627492942273378
m=10    1.9999984312683816      0.00000156873161838433
```

(2)

Es fällt auf, dass sich der Fehler in jedem Schritt etwa geviertelt wird. Bezeichnet a_n die n-te Approximation, so gilt $a_0 \leq a_1 \leq \cdots \leq a_n$; deshalb ist diese Beobachtung äquivalent dazu, dass die Quotienten $(a_i - a_{i+1})/(a_{i+1} - a_{i+2})$ ungefähr 4 sind.

```
>>> for i in range(m-2):
...     print( (results[i] - results[i+1])/(results[i+1] - results[i+2]) )
...
4.164784400584785
4.039182316416593
4.009677144752887
4.002411992937073
4.00060254408483
4.000150607761501
4.000037649528035
4.000009414842847
```

(3)

Wir erhalten die folgenden Approximationen:

```
>>> m = 10
>>> f = (lambda x : 3**(3*x-1))
>>> results = trapeze( f, 0, 2, m)
>>> for i in range(m):
...     print("m={:2d}\t{:24.20f}".format(i+1, results[i]))
...
m= 1    130.66666666666665719276
m= 2     89.58204463929762084717
m= 3     77.74742639121230070032
m= 4     74.66669853961546721166
m= 5     73.88840395800384897029
m= 6     73.69331521665949935596
m= 7     73.64451070980437918934
m= 8     73.63230756098684537392
m= 9     73.62925664736960129630
m=10     73.62849391106399821183
```

Da f konvex ist, sind die Approximationen b_n monoton fallend. Die Vermutung lässt sich erneut durch das Betrachten der Quotienten $(b_i - b_{i+1})/(b_{i+1} - b_{i+2})$ überprüfen:

```
>>> for i in range(m-2):
...     (results[i] - results[i+1])/(results[i+1] - results[i+2])
...
3.471562932248868
3.841500716121706
3.958305665211694
3.9894387356667425
3.9973509398114637
3.9993371862347957
3.9998342622879792
3.999958563440565
```

Unsere Vermutung scheint sich zu bestätigen.

Exercise 3.10

Für die Funktion $f(x) = e^{x^2}$ gilt $f''(x) = (4x^2+2)e^{x^2}$. Da $f''(x) > 0$ auf $[0,1]$ monoton steigend ist, gilt für alle $0 \le a \le b \le 1$, dass

$$|\mathrm{E}(f,a,b)| \le \frac{(b-a)^3}{12} \max_{a\le x\le b} |f''(x)| \le \frac{(b-a)^3}{12} f''(b)\,.$$

Für alle $n \ge 1$ und $0 \le k \le n-1$ gilt deshalb

$$\left| E\left(f, \frac{k}{n}, \frac{k+1}{n}\right)\right| \le \frac{1}{12n^3}\left(4\left(\frac{k+1}{n}\right)^2 + 2\right)\underbrace{e^{((k+1)/n)^2}}_{\le e \le 4} \le \frac{1}{12n^3}(4+2)\cdot 4 \le \frac{2}{n^3}\,.$$

Der gesamte Fehler für eine Unterteilung von $[0,1]$ in n Intervalle lässt sich deshalb ingesamt durch

$$n \cdot \frac{2}{n^3} = \frac{2}{n^2}$$

abschätzen. Dabei gilt

$$\frac{2}{n^2} < 10^{-6} \iff n^2 > 2\cdot 10^6 \iff n > \sqrt{2}\cdot 10^3 \impliedby n > 1500\,.$$

Indem wir das verbesserte Trapezverfahren aus Exercise 3.9 nutzen, so gilt mit $n = 2^m$, dass $n > 1500$ für $m \ge 11$.

```
>>> from math import exp
>>> f = (lambda x: exp(x**2))
>>> m = 11
>>> results = trapeze(f, 0, 1, m)
>>> for i in range(m):
...     print("m={:2d}\t{:24.20f}".format(i+1, results[i]))
...
m= 1      1.57158316545863208091
m= 2      1.49067886169885532865
m= 3      1.46971227642966528748
m= 4      1.46442031014948170764
m= 5      1.46309410260642858148
m= 6      1.46276234857772702291
m= 7      1.46267939741858832292
m= 8      1.46265865883777390621
m= 9      1.46265347414312651964
m=10      1.46265217796637525538
m=11      1.46265185392199392744
```

Exercise 3.11

Ist $T_n(x) = \sum_{k=0}^{n} x^k/k!$ das k-te Taylorpolynom für f an der Entwicklungsstelle 0, so gilt für das Restglied $R_n(x) \coloneqq e^x - T_n(x)$, dass es für jedes $x \in \mathbb{R}$ ein ξ zwschen 0 und x gibt, so dass

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\xi^n = \frac{e^{\xi}\xi^n}{(n+1)!}\,.$$

Für alle $x \geq 0$ gilt $e^\xi \leq e^x \leq 3^x$, und somit gilt

$$|R_n(x)| \leq \frac{3^x x^n}{(n+1)!} \qquad \text{für alle } x \geq 0.$$

Für alle $x \leq 0$ gilt $e^\xi \leq e^0 = 1$, und somit

$$|R_n(x)| \geq \frac{(-x)^n}{(n+1)!}.$$

Dies führt zu dem folgenden Code:

```
def exp_approx(x):
    y = 1       # current approx
    d = 6       # number of digits
    n = 1
    fac = 1     # n!
    if x >= 0:
        while fac < (3**x) * (x**(n+1)) * 10**d
            y += x**n / fac
            n += 1
            fac *= n
    if x < 0:
        while fac < ((-x)**(n+1)) * 10**d:
            y += x**n / fac
            n += 1
            fac *= n
    return y
```

Für etwa $x \geq 23$ und $x \leq -26$ führt funktioniert diese Approximation allerdings nicht mehr mit der gewünschten Genauigkeit, da die aufzuaddierenden Summanden $x^n/n!$ dann betragsmäßig zu klein werden.

Exercise 3.12

(1)

Wir definieren eine Klasse `TimeOutError`:

```
class TimeOutError(Exception):
    pass
```

Wir implementieren das Newton-Verfahren für mit der gewünschten Genauigkeit:

```
def newton(f, f_prime, x):
    n = 1
    xold = x
    xnew = x
    while n <= 100:
        d = f_prime(xold)
        if d == 0:
            raise ZeroDivisionError("derivative vanishes at {}".format(xold))
        xnew = xold - f(xold)/d
```

```
        if 0 <= xnew - xold <= 1.E-7 or 0 <= xold - xnew <= 1.E-7:
            return xnew
        xold = xnew
        n += 1
    raise TimeOutError("the calculation takes too long")
```

(2)

Wir erhalten das folgende Ergebnis:

```
>>> f = (lambda x: x**2 - 2)
>>> fprime = (lambda x: 2*x)
>>> print( newton(f, fprime, 1) )
1.4142135623730951
```

Die ersten 15 Nachkommestellen stimmen mit dem exakten Ergebnis überein.