

Solutions to Exercises to

Programming Methods in Scientific Computing

David Bauer
Jendrik Stelzner

Letzte Änderung: 5. Februar 2018

Inhaltsverzeichnis

3 Python, the Fundamentals	3
4 Python in Scientific Computation	25
5 Python in Computer Algebra, SymPy	40
7 The language C++	47
8 Matlab	65
9 Maple (done in SymPy)	74

3 Python, the Fundamentals

Exercise 3.1

Wir nutzen den folgenden, leicht abgeänderten Code (für besseren Output):

```
1 print("Calculate machine epsilon:")
2 eps = 1.0
3 i = 0
4 while 1+eps > 1:
5     eps /= 2
6     i += 1
7 eps *= 2
8 print("eps: {}".format(eps))
9 print("iterations: {}".format(i))
10
11 print("With different loop condition:")
12 eps = 1.0
13 i = 0
14 while eps > 0:
15     eps /= 2
16     i += 1
17 eps *= 2
18 print("eps: {}".format(eps))
19 print("iterations: {}".format(i))
```

Wir erhalten den folgenden Output:

```
Calculate machine epsilon:
eps: 2.220446049250313e-16
iterations: 53
With different loop condition:
eps: 0.0
iterations: 1075
```

Der IEEE Standard für Floating-Point-Variablen legt fest, dass bei einer Variable vom Typ double die Mantisse 52 Bit groß ist, und der Exponent 11 Bit. Hierbei ist zu beachten, dass der Exponent in der Darstellung um den Bias von 1023 verschoben wird, falls es sich um eine normalisierte Zahl handelt (Exponent ungleich 0) und um 1022, falls es sich um eine denormalisierte Zahl handelt. Ebenfalls wird der größte Exponent (2047) als `NaN` oder `inf` interpretiert, sodass sich der tatsächliche Exponent der Zahl also im Bereich von -1022 bis $+1023$ befindet.

Das bedeutet, dass die kleinste darstellbare Zahl, die größer als 1 ist, von der Form $(1,00\dots001)_2 = 1 + 2^{-52}$ ist. (Hier steht $(\dots)_2$ für die Binärdarstellung.) Die kleinste darstellbare Zahl, die größer als 0 ist, ist dagegen die Zahl

$$(0,00\dots001)_2 \cdot 2^{-1022} = 2^{-1074}.$$

Der erste Algorithmus rechnet der Reihe nach zunächst die Zahlen 2^{-i} aus und vergleicht dann 1 mit $1 + 2^{-i}$. Bei $i = 52$ ist dies die kleinste Zahl größer als 1, wie oben beschrieben. Im nächsten Durchlauf wird dann `eps` auf 2^{-53} gesetzt. Hier wird nun $1 + \text{eps}$ auf 1 gerundet, die Schleife bricht also ab. Anschließend wird `eps` wieder verdoppelt, sodass sich der folgende Output ergibt:

```
eps: 2.220446049250313e-16
iterations: 53
```

Beim zweiten Algorithmus wird dagegen 2^{-i} mit 0 verglichen. Dies muss bis zum 1074-sten Durchlauf der Schleife nicht gerundet werden. Zu diesem Zeitpunkt hat `eps` den Wert 2^{-1074} , die kleinste positive darstellbare Zahl. Im nächsten Schleifendurchlauf wird `eps` halbiert und auf 0 gerundet. Die Schleife bricht also nach 1075 Durchläufen ab. Danach wird `eps` wieder verdoppelt, hat also den Wert $0 \cdot 2 = 0$. Der Output ist also:

```
eps: 0.0
iterations: 1075
```

Exercise 3.2

Wir erweitern die gegebene Klasse `Polynomial` um eine Methode `derivative` zum Ableiten, sowie eine Methode `antiderivative` zum Bilden einer Stammfunktion. Dabei wählen wir die „Integrationskonstante“ als 0. Wir definieren außerdem Funktionen zum Ausgeben von Polynomen durch die `print`-Funktion.

```
1 class Polynomial:
2     def __init__(self, coeff):
3         self.coeff = coeff
4
5     # convert polynomial to string for printing
6     def __str__(self):
7         s = ""
8         first = True # checks if other terms have already been printed
9         # go from highest coeff to lowest
10        for i in range(len(self.coeff)-1, -1, -1):
11            c = self.coeff[i] # current coefficient
12            # dont print zero coefficient
13            if c == 0:
14                continue
15            prefix = ""
16            power = ""
17            # check for sign of next coefficient
18            if c < 0:
19                if first:
20                    prefix = "_"
21                else:
22                    prefix = " - "
23            else:
24                if not first:
25                    prefix = " + "
```

```

26     # check for power
27     if i >= 2:
28         power = "x^{}".format(i)
29     elif(i == 1):
30         power = "x"
31     # add new term to output string
32     s += prefix + str(abs(c)) + power
33     first = False
34     # check for zero polynomial
35     if s == "":
36         s = "0"
37     return s
38
39 # to get string representation in terminal without printing command
40 def __repr__(self):
41     return str(s)
42
43 # evaluate polynomia at a point
44 def __call__(self, x):
45     s = 0
46     for i in range(len(self.coeff)):
47         s += self.coeff[i]*x**i
48     return s
49
50 # coefficientwise addition of polynomials
51 def __add__(self, other):
52     l = []
53     if len(self.coeff) > len(other.coeff):
54         l += self.coeff # append to list
55         for i in range(len(other.coeff)):
56             l[i] += other.coeff[i]
57     else:
58         l += other.coeff # append to list
59         for i in range(len(self.coeff)):
60             l[i] += self.coeff[i]
61     return Polynomial(l)
62
63 # check for equality coefficientwise
64 def __eq__(self, other):
65     return self.coeff == other.coeff
66
67 # calculates the derivative
68 def derivative(self):
69     coeff = []
70     for i in range(1,len(self.coeff)):
71         coeff.append(i * self.coeff[i])
72     return Polynomial(coeff)
73
74 # calculates the antiderivative with zero constant term
75 def antiderivative(self):
76     coeff = [0]
77     for i in range(len(self.coeff)):
78         coeff.append(self.coeff[i]/(i+1))
79     return Polynomial(coeff)

```

Für das gegebene Polynom $p(x) = 3x^2 + 2x + 1$ testen wir unser Programm mit dem

folgenden Code:

```
1 p = Polynomial([1,2,3])
2 print("The given polynomial p:")
3 print(p)
4
5 print("The derivative of p:")
6 print( p.derivative() )
7
8 print("The antiderivative of p:")
9 print( p.antiderivative() )
10
11 print("Taking antiderivative and then derivative:")
12 print( p.antiderivative().derivative() )
```

Dabei erhalten wir den folgenden Output:

```
The given polynomial p:
3x^2 + 2x + 1
The derivative of p:
6x + 2
The antiderivative of p:
1.0x^3 + 1.0x^2 + 1.0x
Taking antiderivative and then derivative:
3.0x^2 + 2.0x + 1.0
```

Exercise 3.3

Wir definieren direkt Klasse `Matrix`, die über alle Methoden verfügt, die wir in diesem und den späteren Aufgabenteilen nutzen werden.

```
1 class Matrix():
2     # constructor takes list of rows
3     def __init__(self, m, n, entries):
4         self.rows = m
5         self.cols = n
6         self.entries = entries
7
8     # gets the rows via A[i]
9     def __getitem__(self, i):
10        return self.entries[i]
11
12    # sets the rows via A[i]
13    def __setitem__(self, i, r):
14        self.entries[i] = r
15
16    def height(self):
17        return self.rows;
18
19    def width(self):
20        return self.cols;
21
22    # converts matrix to a string for printing
23    # constructs the matrix columnwise, left aligned
```

```

24  def __str__(self):
25      rowstrings = [ "" ]*self.rows
26      for j in range(self.cols): # going through columns
27          numbers = []           # numbers to appear in column j
28          maxlen = 0             # maximal length of a number in column j
29          for i in range(self.rows):
30              s = str(self[i][j])
31              numbers.append(s)
32              maxlen = max(len(s), maxlen)
33          for i in range(self.rows):
34              # pad the entries if they are too short
35              numbers[i] += ")*(maxlen-len(numbers[i]))"
36              # build up column in string
37              rowstrings[i] += numbers[i] + " "
38      # put the row strings together
39      s = ""
40      for r in rowstrings:
41          s += "[" + r[:-1] + "]\n" # remove white space at the end of
42          lines
43      s = s[:-1]                  # remove at newline at the end
44      return s
45
46  def __repr__(self):
47      return str(self)
48
49  # addition of matrices of the same dimensions
50  def __add__(self, other):
51      if self.rows != other.rows:
52          raise ValueError('cannot add matrices of different height')
53      if self.cols != other.cols:
54          raise ValueError('cannot add matrices of different width')
55      result = []
56      for i in range(self.rows):
57          row = []
58          for j in range(other.cols):
59              row.append(self[i][j] + other[i][j])
60          result.append(row)
61      return Matrix(self.rows, self.cols, result)
62
63  # multiplication to matrices of compatible dimensions
64  def __mul__(self, other):
65      if self.cols != other.rows:
66          raise ValueError('cannot multiply matrices of wrong dimensions')
67      # special check for empty matrices
68      if self.cols == 0 or self.rows == 0 or other.rows == 0 or other.cols
69          == 0:
70          return zeromatrix(self.rows, other.cols)
71      result = []
72      for i in range(self.rows):
73          row = []
74          for j in range(other.cols):
75              s = self[i][0] * other[0][j] # makes sure s has right type
76              for k in range(1, self.cols):
77                  s += self[i][k] * other[k][j]
78              row.append(s)
79          result.append(row)

```

```

78     return Matrix(self.rows, other.cols, result)
79
80 # compares two matrices entrywise
81 def __eq__(self, other):
82     if self.rows != other.rows or self.cols != other.cols:
83         return False
84     for i in range(self.rows):
85         for j in range(self.cols):
86             if self[i][j] != other[i][j]:
87                 return False
88     return True
89
90 # add row i -> i + c*j
91 def addrowtofrom(self, i, j, c):
92     for k in range(self.cols):
93         # make c responsible for implementing the multiplication
94         self[i][k] = c * self[j][k] + self[i][k]
95
96 # add column i -> i + c*j
97 def addcolumntofrom(self, i, j, c):
98     for k in range(self.rows):
99         # make c responsible for implementing the multiplication
100        self[k][i] = c * self[k][j] + self[k][i]
101
102 # multiply row i -> c*i
103 def multrow(self, i, c):
104     for j in range(self.cols):
105         self[i][j] = c * self[i][j]
106
107 # multiply column j -> c*j
108 def multcolumn(self, j, c):
109     for i in range(self.rows):
110         self[i][j] = c * self[i][j]
111
112 # swap rows i <-> j
113 def swaprows(self, i, j):
114     temp = self[i]
115     self[i] = self[j]
116     self[j] = temp
117
118 # swap columns i <-> j
119 def swapcolumns(self, i, j):
120     for k in range(self.rows):
121         temp = self[k][i]

```

Wir definieren zudem Hilfsfunktionen, die wir im Weiteren nutzen werden:

```

1 ##### AUXILIARY MATRIX FUNCTIONS
2
3 # creates a zero matrix of given height and width
4 def zeromatrix(height, width):
5     entries = []
6     for i in range(height):
7         entries.append([0]*width)
8     return Matrix(height, width, entries)
9
10 # creates an identiy matrix of given square size

```

```

11| def identitymatrix(size):
12|     E = zeromatrix(size, size)
13|     for i in range(size):
14|         E[i][i] = 1
15|     return E
16|
17| # creates the transposed matrix
18| def transpose(A):
19|     m = A.height()
20|     n = A.width()
21|     result = []
22|     for i in range(m):
23|         row = []
24|         for j in range(n):
25|             row.append(A[j][i])
26|         result.append(row)
27|     return Matrix(n, m, result)
28|
29| # creates a new matrix by applying a function entrywise to a given matrix
30| def mapentries(A, f):
31|     m = A.height()
32|     n = A.width()
33|     result = []
34|     for i in range(m):
35|         row = []
36|         for j in range(n):
37|             row.append(f(A[i][j]))
38|         result.append(row)
39|     return Matrix(m, n, result)

```

Die Assoziativität der Matrixmultiplikation testen wir mit dem folgenden Code:

```

1 A = Matrix(3, 2, [[0,1],[1,0],[1,1]])
2 print("A:")
3 print(A)
4
5 B = Matrix(2, 4, [[1,2,3,4],[5,6,7,8]])
6 print("B:")
7 print(B)
8
9 C = Matrix(4, 2, [[1,0],[0,1],[1,0],[0,1]])
10 print("C:")
11 print(C)
12
13 print("Checking if A(BC) == (AB)C:")
14 print(A * (B * C) == (A * B) * C)

```

Wir erhalten wir den folgenden Output:

```

A:
[0 1]
[1 0]
[1 1]
B:
[1 2 3 4]
[5 6 7 8]
C:

```

```
[1 0]
[0 1]
[1 0]
[0 1]
Checking if A(BC) == (AB)C:
True
```

Exercise 3.4

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

```
1 class Rational():
2     def __init__(self, num, denum = 1):    # default denominator is 1
3         if type(num) == Rational:
4             p = num/Rational(denum)
5             self.num = p.num
6             self.denum = p.denum
7         elif type(num) != int:
8             raise TypeError("numerator is no integer")
9         elif type(denum) != int:
10             raise TypeError("denominator is no integer")
11         elif denum == 0:
12             raise ZeroDivisionError("denominator is zero")
13         else:
14             self.num = num
15             self.denum = denum
16
17     # allows printing of rational numbers
18     def __str__(self):
19         return "{} / {}".format(self.num, self.denum)
20
21     def __repr__(self):
22         return str(self)
23
24     # adding to a rational number
25     def __add__(self, other):
26         if type(other) == int:
27             return self + Rational(other)
28         elif type(other) == Rational:
29             return Rational( self.num * other.denum + self.denum * other.num,
30                             self.denum * other.denum )
31         raise TypeError("unsupported operand type(s) for + or add(): 'Rational' and '{}'".format(type(other).__name__))
32
33     # subtracting from a rational number
34     def __sub__(self, other):
35         if type(other) == int:
36             return self - Rational(other)
37         elif type(other) == Rational:
38             return Rational( self.num * other.denum - self.denum * other.num,
39                             self.denum * other.denum )
40         raise TypeError("unsupported operand type(s) for - or sub(): 'Rational' and '{}'".format(type(other).__name__))
```

```

39      # multiply to a rational number
40      def __mul__(self, other):
41          if type(other) == int:
42              return self * Rational(other)
43          elif type(other) == Rational:
44              return Rational( self.num * other.num, self.denum * other.denum )
45          raise TypeError("unsupported operand type(s) for * or mul(): '"
46                          Rational' and '{}'".format(type(other).__name__))
47
48      # divide a rational number
49      def __truediv__(self, other):
50          if type(other) == int:
51              return self / Rational(other)
52          elif type(other) == Rational:
53              if other.num == 0:
54                  raise ZeroDivisionError("division by zero")
55              return Rational( self.num * other.denum, self.denum * other.num )
56          raise TypeError("unsupported operand type(s) for / or truediv(): '"
57                          Rational' and '{}'".format(type(other).__name__))
58
59      # powers of rational number
60      def __pow__(self, n):  # supports only integer powers
61          if not type(n) == int:
62              raise TypeError("unsupported operand type(s) for ** or pow(): '"
63                          Rational' and '{}'".format(type(n).__name__))
64          if n >= 0:
65              return Rational(self.num**n, self.denum**n)
66          return Rational(self.denum, self.num)**(-n)
67
68      # +x for rational x
69      def __pos__(self):
70          return Rational( self.num, self.denum )
71
72      # -x for rational x
73      def __neg__(self):
74          return Rational( -self.num, self.denum )
75
76      # absolute value
77      def __abs__(self):
78          return Rational(abs(self.num), abs(self.denum))
79
80      # equality of rational numbers
81      def __eq__(self, other):
82          if type(other) == int:  # allows comparison to int, used for x == 0
83              return (self == Rational(other))
84          elif type(other) == Rational:
85              return (self.num * other.denum == self.denum * other.num)
86          raise TypeError("unsupported operand type(s) for == or eq(): '"
87                          Rational' and '{}'".format(type(other).__name__))
88
89      # casts to float
90      def __float__(self):
91          return self.num / self.denum

```

Die LU-Zerlegung von passenden Matrix mit ganzzahligen Einträgen kann nun dem

folgenden naiven Algorithmus berechnet werden:

```

1 from matrices import *
2 from rationals import *
3 from copy import deepcopy
4
5 # expects the matrix entries to be comparable to 0 in a sensible way
6 def naive_lu(A):
7     if A.height() != A.width():
8         raise ValueError("matrix is not square")
9     U = deepcopy(A)    # circumvent pass by reference
10    n = U.height()
11    L = identitymatrix(n)
12    # bring U in upper triangular form, change L such that always LU = A
13    for j in range(n):
14        if U[j][j] == 0:
15            raise ValueError("algorithm does not work for this matrix")
16        for i in range(j+1,n):
17            L.addcolumnfrom(j, i, U[i][j]/U[j][j]) # important: change L
18            first
19            U.addrowfrom(i, j, -U[i][j]/U[j][j])
20
21    return (L,U)

```

Wir testen das Programm anhand der gegebenen Matrizen

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 6 & 3 \\ 9 & 10 & 6 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

mit dem folgenden Code:

```

1 A = Matrix(3, 3, [[3,2,1],[6,6,3],[9,10,6]])
2 print("A:")
3 print(A)
4
5 (L,U) = naive_lu(mapentries(A, Rational))
6 print("L:")
7 print(L)
8 print("U:")
9 print(U)
10 print("Check if L*U == A:")
11 print(mapentries(L, Rational) * mapentries(U, Rational) == A)
12
13 B = Matrix(2, 2, [[0,1],[1,0]])
14 print("B:")
15 print(B)
16
17 print("Trying to calculate the LU decomposition of B:")
18 (L,U) == naive_lu(B)

```

Wir erhalten den folgenden Output:

```

A:
[3 2 1]
[6 6 3]
[9 10 6]

```

```

L:
[[9/9  0/18  0]
 [18/9 18/18 0]
 [27/9 36/18 1]
U:
[[3/1   2/1   1/1    ]
 [0/3   6/3   3/3    ]
 [0/162 0/162 162/162]]
Check if L*U == A:
True
B:
[[0 1]
 [1 0]]
Trying to calculate the LU decomposition of B:
Traceback (most recent call last):
  File "exercise_03_04.py", line 69, in <module>
    (L,U) == naive_lu(B)
  File "exercise_03_04.py", line 17, in naive_lu
    raise ValueError("algorithm does not work for this matrix")
ValueError: algorithm does not work for this matrix

```

Da die Matrix B keine LU-Zerlegung besitzt, ist es okay, dass unser Algorithmus diese nicht findet.

Exercise 3.5

Wir bestimmen die Cholesky-Zerlegung eintragsweise.

```

1 from matrices import *
2 from copy import deepcopy
3 from math import sqrt
4
5 # expects int or float as matrix entries
6 def cholesky(A):
7     if A.height() != A.width():
8         raise ValueError("matrix is not square")
9     B = deepcopy(A) # circumvent pass by reference
10    n = B.height()
11    L = zeromatrix(n,n)
12    for i in range(n):
13        rowsum = 0
14        for j in range(i):
15            s = 0
16            for k in range(j):
17                s += L[i][k] * L[j][k]
18            L[i][j] = (B[i][j] - s)/L[j][j]
19            rowsum += L[i][j]**2
20        L[i][i] = sqrt( B[i][i] - rowsum )
21

```

Für die gegebenen Matrizen

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 10 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 1.01 \cdot 10^{-2} & 0.705 & 1.42 \cdot 10^{-2} \\ 0.705 & 49.5 & 1 \\ 1.42 \cdot 10^{-2} & 1 & 1 \end{pmatrix}$$

testen wir unser Programm mit dem folgenden Code:

```

1 A = Matrix(3,3,[[1,2,1],[2,5,2],[1,2,10]])
2 print("A:")
3 print(A)
4
5 L = cholesky(A)
6 print("L:")
7 print(L)
8 print("L * L^T:")
9 print(L * transpose(L))
10
11 B = Matrix(3,3,[[1.01E-2, 0.705, 1.42E-2],[0.705,49.5,1],[1.42E-2,1,1]])
12 print("B:")
13 print(B)
14
15 L = cholesky(B)
16 print("L:")
17 print(L)
18 print("L * L^T:")
19 print(L * transpose(L))

```

Wir erhalten den folgenden Output:

```

$ python exercise_03_05.py
A:
[[1 2 1]
 [2 5 2]
 [1 2 10]]
L:
[[1.0 0 0]
 [2.0 1.0 0]
 [1.0 0.0 3.0]]
L * L^T:
[[1.0 2.0 1.0]
 [2.0 5.0 2.0]
 [1.0 2.0 10.0]]
B:
[[0.0101 0.705 0.0142]
 [0.705 49.5 1]
 [0.0142 1 1]]
L:
[[0.1004987562112089 0 0]
 [7.015012190980423 0.5381486415443629 0]
 [0.14129528100981847 0.016374437298272527 0.9898320672556135]]
L * L^T:
[[0.01010000000000001 0.705 0.01420000000000003]
 [0.705 49.5 1.0]
 [0.01420000000000003 1.0 1.0]]

```

Exercise 3.6

Mithilfe elementarer Zeilenumformungen, die in der Klasse `Matrix` implementiert sind, lässt sich nun der Gauß-Algorithmus zum Invertieren von Matrizen implementieren.

```
1 from rationals import *
2 from matrices import *
3 from copy import deepcopy
4
5 # expects the matrix entries to be int or Rational
6 def invert(A):
7     if A.height() != A.width():
8         raise ValueError("matrix is not square")
9     B = deepcopy(A)    # circumvent pass by reference
10    n = B.height()
11    Inv = identitymatrix(n)
12    B = mapentries(B,Rational)      # make all the
13    Inv = mapentries(Inv,Rational)  # entries rational
14    # bring B in lower triangular form
15    for j in range(n):
16        p = -1
17        for i in range(j,n):
18            if B[i][j] != 0:
19                p = i
20                break
21        if p == -1:
22            raise ZeroDivisionError("matrix is not invertible")
23        for i in range(p+1,n):
24            Inv.addrowtofrom(i, p, -B[i][j]/B[p][j])  # import: change
25            B.addrowtofrom(i, p, -B[i][j]/B[p][j])
26    # norm the diagonal entries
27    for i in range(n):
28        Inv.multrow(i, B[i][i]**(-1))    # **(-1) also works for Rational
29        B.multrow(i, B[i][i]**(-1))
30    # bring B into identity form
31    for j in range(n):
32        for i in range(j):
33            Inv.addrowtofrom(i, j, -B[i][j])
34            B.addrowtofrom(i, j, -B[i][j])
35    return Inv
```

Wir testen unser Programm anhand der gegebenen Matrix

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

mit dem folgenden Code:

```
1 A = Matrix(3, 3, [[3,-1,2],[-3,4,-1],[-6,5,-2]])
2 print("A:")
3 print(A)
4
5 B = invert(A)
```

```

6| print("A^(-1) with rationals:")
7| print(B)
8| print("A^(-1) with floats:")
9| print(mapentries(B,float))
10|
11| print("Checking if A*B == I (using rationals):")
12| print(mapentries(A,Rational) * B == identitymatrix(3))

```

Dabei nutzen wir erneut die Klasse `Rational`, um ein genaues Rechnen zu erlauben.
Wir erhalten den folgenden Output:

```

A:
[3 -1 2 ]
[-3 4 -1]
[-6 5 -2]
A^(-1) with rationals:
[-1162261467/3486784401 3099363912/3486784401 -2711943423/3486784401]
[0/43046721 28697814/43046721 -14348907/43046721 ]
[59049/59049 -59049/59049 59049/59049]
A^(-1) with floats:
[-0.3333333333333333 0.8888888888888888 -0.7777777777777778]
[0.0 0.6666666666666666 -0.3333333333333333]
[1.0 -1.0 1.0]
Checking if A*B == I (using rationals):
True

```

Exercise 3.7

Wir Berechnen die QR-Zerlegung einer nicht-singulären Matrix A durch Anwenden des Gram–Schmidt–Verfahrens auf die Spalten von A , von links nach rechts:

```

1| from matrices import *
2| from copy import deepcopy
3| from math import sqrt
4|
5| # assumes the matrix to have integer or float values and to be nonsingular
6| def qrdecomp(A):
7|     if A.height() != A.width():
8|         return ValueError("only square matrices are supported")
9|     n = A.height()
10|    Q = deepcopy(A) # circumvents pass by reference
11|    R = identitymatrix(n)
12|    # orthogonalize Q while maintaining A = QR
13|    for j in range(n):
14|        # make the j-th column of Q orthogonal to the previous columns
15|        for k in range(j):
16|            s = 0 # inner product of j-th and k-th columns
17|            for i in range(n):
18|                s += Q[i][k] * Q[i][j]
19|            Q.addcolumnfrom(j, k, -s)
20|            R.addrowtofrom(k, j, s)
21|    sn = 0 # squared norm of the j-th column
22|    for i in range(n):

```

```

23         sn += Q[i][j]**2
24     # normalize the j-th column
25     Q.multcolumn(j, sqrt(sn)**(-1))
26     R.multrow(j, sqrt(sn))
27     return (Q,R)

```

Für die gegebene Matrix

$$A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}$$

testen wir das Programm mithilfe des folgenden Codes:

```

1 A = Matrix(3,3,[[12,-51,4],[6,167,-68],[-4,24,-41]])
2 print("A:")
3 print(A)
4
5 (Q,R) = qrdecomp(A)
6 print("Q:")
7 print(Q)
8 print("R:")
9 print(R)
10
11 print("Q * Q^T:")
12 print(Q * transpose(Q))
13 print("Q*R:")
14 print(Q*R)

```

Wir erhalten den folgenden Output:

```

A:
[12 -51 4]
[6 167 -68]
[-4 24 -41]
Q:
[0.8571428571428571 -0.3942857142857143 -0.33142857142857124]
[0.42857142857142855 0.9028571428571428 0.03428571428571376 ]
[-0.2857142857142857 0.17142857142857143 -0.9428571428571428 ]
Q * Q^T:
[0.9999999999999998 1.474514954580286e-16 -1.6653345369377348e-16]
[1.474514954580286e-16 0.9999999999999998 4.996003610813204e-16 ]
[-1.6653345369377348e-16 4.996003610813204e-16 1.0]
R:
[14.0 20.99999999999996 -14.000000000000002]
[0.0 175.0 -69.99999999999999 ]
[0.0 0.0 35.0 ]
Q*R:
[12.0 -51.0 4.000000000000002]
[6.0 167.0 -68.0 ]
[-4.0 24.0 -41.0 ]

```

Exercise 3.8

(1)

Alle notwendigen Funktionswerte werden zunächst berechnet und in einer Liste gespeichert, um das mehrfache Berechnen gleicher Funktionswerte zu umgehen.

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

(2)

Wir testen unser Programm anhand des gegebenen Integrals $\int_0^\pi \sin(x) dx$ mit dem folgenden Code:

```
1 from math import sin, pi
2 n = 1
3 s = 0
4 while 2 - s >= 1.E-6:    # sin is concave on [0,pi] -> estimate too small
5     n += 1                  # can skip n = 1 because it results in 0
6     s = trapeze(sin, 0, pi, n)
7
8 print("Estimate for integral of sin from 0 to pi using trapeze:")
9 print(s)
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_08.py
Estimate for integral of sin from 0 to pi using trapeze:
1.9999990007015205
```

Exercise 3.9

(1)

Wir definieren eine neue Funktion powertrapeze, welche das angegebene Verfahren implementiert:

```
1 def powertrapeze(f, a, b, mmax):
2     integrals = []          # list of the approximations
3     values = [f(a), f(b)]    # list of the calculated values
4     for m in range(1,mmax+1):
5         n = 2**m
6         for k in range(1,n,2):
7             values.insert(k, f(a + (k/n)*(b-a))) # add new values
8             integral = 0
```

```

9     for i in range(len(values)-1):
10        integral += values[i] + values[i+1]
11        integral = (b-a)*integral/n/2
12        integrals.append(integral) # add new approx.
13    return integrals

```

Hiermit berechnen die Approximationen für $\int_0^\pi \sin(x) dx$ für $m = 1, \dots, 10$ mit dem folgenden Code:

```

1 from math import sin, pi
2 m = 10
3 results = powertrapeze(sin, 0, pi, m)
4 print("Calculate trapeze estimate for sin(x) from 0 to pi, 2^m intervals:")
5 print(" m \testimate \t\terror")
6 for i in range(m):
7     print("{:2d}\t{}\t{:.20f}".format(i+1, results[i], 2-results[i]))

```

Wir erhalten den folgenden Output:

Calculate trapeze estimate for sin(x) from 0 to pi, 2^m intervals:		
m	estimate	error
1	1.5707963267948966	0.42920367320510344200
2	1.8961188979370398	0.10388110206296019555
3	1.9742316019455508	0.02576839805444919307
4	1.9935703437723395	0.00642965622766045186
5	1.9983933609701445	0.00160663902985547224
6	1.9995983886400386	0.00040161135996141795
7	1.9998996001842035	0.00010039981579645918
8	1.9999749002350518	0.00002509976494824429
9	1.9999937250705773	0.00000627492942273378
10	1.9999984312683816	0.00000156873161838433

(2)

Es fällt auf, dass der Fehler in jedem Schritt etwa geviertelt wird. Bezeichnet a_n die n -te Approximation, so gilt $a_0 \leq a_1 \leq \dots \leq a_n$, da \sin auf $[0, \pi]$ konkav ist. Deshalb ist die Vermutung äquivalent dazu, dass die Quotienten $(a_i - a_{i+1})/(a_{i+1} - a_{i+2})$ näherungsweise 4 sind. Dies testen wir mit dem folgenden weiteren Code:

```

1 print("Quotients of any two subsequent differences of estimates:")
2 for i in range(m-2):
3     q = (results[i] - results[i+1]) / (results[i+1] - results[i+2])
4     print(q)

```

Wir erhalten den folgenden Output:

Quotients of any two subsequent differences of estimates:
4.164784400584785
4.039182316416593
4.009677144752887
4.002411992937073
4.00060254408483
4.000150607761501
4.000037649528035
4.000009414842847

Es fällt auf, dass das Verhältnis tatsächlich gegen 4 zu gehen scheint.

(3)

Wir berechnen die Approximationen für $\int_0^2 3^{3x-1} dx$ für $m = 1, \dots, 10$ mit dem folgenden Code:

```

1 m = 10
2 f = (lambda x : 3** (3*x-1))
3 results = powertrapeze( f, 0, 2, m)
4 print("Calculate trapeze estimate for 3^(3x-1) from 0 to 2, 2^m intervals:")
5 print(" m \testimate")
6 for i in range(m):
7     print("{:2d}\t{:24.20f}".format(i+1, results[i]))

```

Wir erhalten den folgenden Output:

```

Calculate trapeze estimate for int. of 3^(3x-1) from 0 to 2, 2^m intervals:
    m      estimate
1      130.66666666666665719276
2      89.58204463929762084717
3      77.74742639121230070032
4      74.66669853961546721166
5      73.88840395800384897029
6      73.69331521665949935596
7      73.64451070980437918934
8      73.63230756098684537392
9      73.62925664736960129630
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. Hierfür nutzen wir (erneut) den folgenden Code:

```

1 print("Quotients of any two subsequent differences of estimates:")
2 for i in range(m-2):
3     q = (results[i] - results[i+1])/(results[i+1] - results[i+2])
4     print(q)

```

Wir erhalten den folgenden Output:

```

Quotients of any two subsequent differences of estimates:
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 $0 \leq a \leq b \leq 1$, dass

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

Für alle $n \geq 1$ und $0 \leq k \leq n-1$ gilt deshalb

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

Der gesamte Fehler bei einer Unterteilung von $[0, 1]$ in n Intervalle lässt sich deshalb durch

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

abschätzen. Dabei gilt

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

Für das verbesserte Trapezverfahren aus Exercise 3.9 gilt mit $n = 2^m$, dass $n > 1500$ für $m \geq 11$. Wir nutzen nun den folgenden Code, um die entsprechenden Approximationen für $m = 1, \dots, 11$ zu bestimmen:

```

1 from math import exp
2 f = (lambda x: exp(x**2))
3 m = 11
4 results = powertrapeze(f, 0, 1, m)
5 print("Calculate trapeze estimate for e^(x^2) from 0 to 1, 2^m intervals:")
6 for i in range(m):
7     print("m = {:2d}\t{:24.20f}".format(i+1, results[i]))

```

Wir erhalten den folgenden Output:

```

Calculate trapeze estimate for e^(x^2) from 0 to 1, 2^m intervals:
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 der Funktion $f(x) = e^x$ an der Entwicklungsstelle 0, so gilt für das Restglied $R_n(x) := e^x - T_n(x)$, dass es für jedes $x \in \mathbb{R}$ ein ξ zwischen 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 $x \geq 0$ gilt dann $e^\xi 3^m$ für $m := \lceil x \rceil$, und somit

$$|R_n(x)| \leq \frac{3^m x^n}{(n+1)!}.$$

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

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

Dies führt zu dem folgenden Code:

```

1 from math import ceil
2
3 def exp_approx(x):
4     y = 1          # current approx
5     d = 6          # number of digits
6     n = 1          # current iteration
7     fac = 1        # n!
8     m = ceil(x)
9     if x >= 0:
10         while fac < (3**m) * (x**(n+1)) * 10**d:
11             y += x**n / fac
12             n += 1
13             fac *= n
14     if x < 0:
15         while fac < ((-x)**(n+1)) * 10**d:
16             y += x**n / fac
17             n += 1
18             fac *= n
19     return y

```

Wir testen die Genauigkeit des Programms mit dem folgenden Code:

```

1 from math import exp
2
3 print("Comparison of exp_approx(x) and exp(x) up to 7 digits.")
4 print("{:>3s}  {:>22s}  {:>22s}  {:>13s}" .format("x", "approximation", "exact",
5           "difference (10 digits)"))
6 for x in range(-30, 31):
7     approx = exp_approx(x)
8     exact = exp(x)
9     print("{:3d}  {:22.7f}  {:22.7f}  {:13.10f}" .format(x, approx, exact,
10           exact-approx))

```

Wir erhalten den folgenden (gekürzten) Output:

Comparison of exp_approx(x) and exp(x) up to 7 digits.		exact	difference (10 digits)
x	approximation		
)			
-30	-0.0000855	0.0000000	0.0000855145
-29	0.0000551	0.0000000	-0.0000550745
-28	0.0000050	0.0000000	-0.0000050079
-27	-0.0000045	0.0000000	0.0000044619
-26	-0.0000014	0.0000000	0.0000013633
-25	-0.0000006	0.0000000	0.0000006464
-24	-0.0000003	0.0000000	0.0000002671
-23	-0.0000000	0.0000000	0.0000000403
-22	-0.0000000	0.0000000	0.0000000071
-21	-0.0000000	0.0000000	0.0000000192
[...]			
21	1318815734.4832141	1318815734.4832146	0.0000004768
22	3584912846.1315928	3584912846.1315918	-0.0000009537
23	9744803446.2489052	9744803446.2489033	-0.0000019073
24	26489122129.8434715	26489122129.8434715	0.0000000000
25	72004899337.3858795	72004899337.3858795	0.0000000000
26	195729609428.8387451	195729609428.8387756	0.0000305176
27	532048240601.7988281	532048240601.7986450	-0.0001831055
28	1446257064291.4738770	1446257064291.4750977	0.0012207031
29	3931334297144.0424805	3931334297144.0419922	-0.0004882812
30	10686474581524.4667969	10686474581524.4628906	-0.0039062500

Für etwa $x \geq 23$ und $x \leq -26$ hat unsere Approximation nicht mehr die gewünschten Genauigkeit, da die aufzuaddierenden Summanden $x^n/n!$ dann zu klein werden.

Exercise 3.12

(1)

Wir definieren zunächst eine Klasse `TimeOutError`, um ggf. eine passende Fehlermeldung ausgeben zu können.

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

Wir implementieren das Newton-Verfahren mit der gewünschten Genauigkeit:

```
1 def newton(f, f_prime, x):
2     eps = 1.E-7 # when to stop
3     xold = x      # current position
4     n = 1        # current iteration
5     while n <= 100:
6         d = f_prime(xold)
7         if d == 0:
8             raise ZeroDivisionError("derivative vanishes at {}".format(xold))
9         xnew = xold - f(xold)/d
10        if abs(xnew - xold) < eps:
11            return xnew
12        xold = xnew
```

```
13|         n += 1
14|     raise TimeOutError("the calculation takes too long")
```

(2)

Wir testen unser Programm anhand der gegebenen Funktion $f(x) = x^2 - 2$ mit dem folgenden Code:

```
1 f = (lambda x: x**2 - 2)
2 fprime = (lambda x: 2*x)
3 print("Calculating an approximation of sqrt(2):")
4 print( newton(f, fprime, 1) )
```

Wir erhalten den folgenden Output:

```
Calculating an approximation of sqrt(2):
1.4142135623730951
```

Dabei stimmen die ersten 15 Nachkommestellen mit dem exakten Ergebnis überein.

4 Python in Scientific Computation

Exercise 4.13

Wir geben den beiden Funktionen ein zusätzliches Argument `err`, mithilfe dessen entschieden wird, wann die Funktion abbricht; standardmäßig ist dieser Wert bei 10^{-6} .

```
1 def bisect_it(f, a, b, err=1e-6):
2     x = (a+b)/2
3     while abs(f(x)) >= err:
4         # f(a) < 0 < f(b) by assumption
5         if f(x) > 0:
6             b = x
7         else:
8             a = x
9         x = (a+b)/2
10    return x
11
12 def bisect_rec(f, a, b, err=1e-6):
13     x = (a+b)/2
14     if abs(f(x)) < err:
15         return x
16     if f(x) > 0:
17         return bisect_rec(f, a, x, err)
18     else:
19         return bisect_rec(f, x, b, err)
```

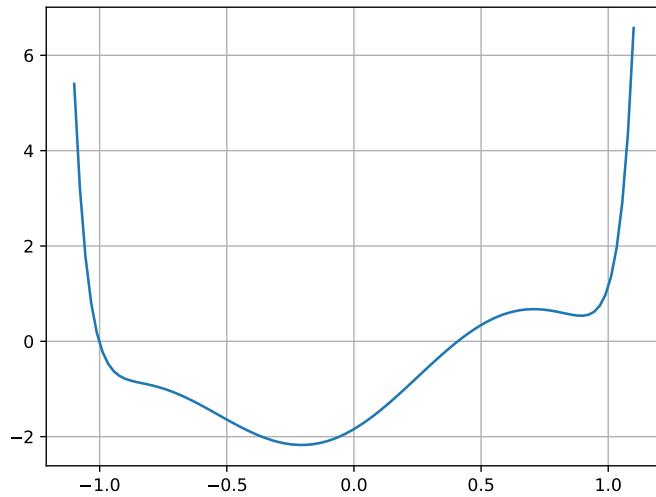
Wir testen die beiden Funktionen an der gegebenen Funktion

$$f(x) = \sin(4x - 1) + x + x^{20} - 1.$$

Hierfür plotten wir die Funktion zunächst mithilfe des folgenden Codes:

```
1 from scipy import linspace, sin
2 import matplotlib.pyplot as plt
3
4 def f(x): return sin(4*x-1)+ x + x**20 - 1
5
6 x = linspace(-1.1, 1.1, 100)
7 plt.clf()
8 plt.plot(x, f(x))
9 plt.grid()
10 plt.show()
```

Wir wählen das Intervall $[-1.1, 1.1]$, da die Funktion f außerhalb dieses Intervalls zu groß wird. Wir erhalten den folgenden Graphen:



Anhand des Graphen sind zwei Nullstellen zu erkennen, jeweils in der Nähe von -1 und 0.4 . Wir berechnen die Nullstellen nun mit dem folgenden Code:

```

1 xit1, xit2 = bisect_it(f, 0, -1.5), bisect_it(f, 0, 1)
2 print("The roots with bisect_it are {} and {}".format(xit1, xit2))
3
4 xrec1, xrec2 = bisect_rec(f, 0, -1.5), bisect_rec(f, 0, 1)
5 print("The roots with bisect_rec are {} and {}".format(xrec1, xrec2))

```

Wir erhalten den folgenden Output:

```

The roots with bisect_it are -1.002246916294098 and 0.4082937240600586.
The roots with bisect_rec are -1.002246916294098 and 0.4082937240600586.

```

Exercise 4.14

Wir nutzen die bisherige `newton`-Methode, kombiniert mit einer Approximation für Ableitungen:

```

1 ### newton method from exercise 3.12
2
3 class TimeOutError(Exception):
4     pass
5
6 def newton(f, f_prime, x):
7     eps = 1.E-7 # when to stop
8     xold = x      # current position
9     n = 1          # current iteration
10    while n <= 100:

```

```

11     d = f_prime(xold)
12     if d == 0:
13         raise ZeroDivisionError("derivative vanishes at {}".format(xold))
14     xnew = xold - f(xold)/d
15     if abs(xnew - xold) < eps:
16         return xnew
17     xold = xnew
18     n += 1
19     raise TimeOutError("the calculation takes too long")
20
21 ### new Newton method
22
23 def prime(f, h=1e-6):
24     return (lambda x: (f(x+h)-f(x))/h)
25
26 def newton_ext(f, x):
27     fprime = prime(f)
28     return newton(f, fprime, x)

```

Wir plotten nun zunächst die gegeben Funktion

$$f(x) = e^x + 2x$$

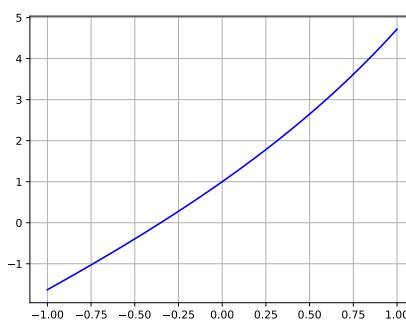
mit dem folgenden Code:

```

1 if __name__ == "__main__":
2
3     from scipy import exp, linspace
4     import matplotlib.pyplot as plt
5
6     def f(x): return exp(x) + 2*x
7
8     # plotting the function
9     x = linspace(-1, 1, 100)
10    plt.clf()
11    plt.plot(x, f(x), color="b")
12    plt.grid()
13    plt.show()

```

Wir erhalten den folgenden Graphen:



Wir wählen nun den Startwert $x_0 = 1$, und berechnen die Nullstelle von f sowohl mit der bisherigen `newton`-Funktion, als auch mit der neuen `newton_ext`-Funktion:

```

1 x0 = -0.25
2 def fprime(x): return exp(x) + 2
3 print("With the exact derivative we get a root at      {}".format(newton
4           (f, fprime, x0)))
5 print("With an approximate derivative we get a root at {}".format(
6           newton_ext(f, x0)))

```

Wir erhalten in der Konsole den folgenden Output:

```
With the exact derivative we get a root at      -0.35173371124919584.
With an approximate derivative we get a root at -0.35173371124919584.
```

Exercise 4.15

(1)

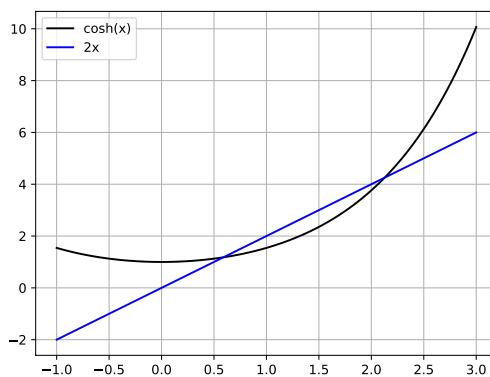
Wir plotten zunächst die beiden Funktionen:

```

1 from scipy import cosh, linspace
2 import matplotlib.pyplot as plt
3
4 # plot the graphs of cosh(x) and 2x
5 x = linspace(-1, 3, 100)
6 plt.clf()
7 plt.plot(x, cosh(x), color="k", label="cosh(x)")
8 plt.plot(x, 2*x, color="b", label="2x")
9 plt.grid()
10 plt.legend()
11 plt.show()

```

Wir erhalten den folgenden Graphen:



Wir bestimmen nun die beiden Nullstellen mit dem folgenden Code:

```
1 from exercise_04_14 import prime, newton_ext
2
3 def f(x): return cosh(x) - 2*x
4 x1 = 0.5 # starting value for left intersection
5 x2 = 2   # starting value for right intersection
6 print( "The intersections are at {} and {}".format(newton_ext(f, x1),
    newton_ext(f, x2)) )
```

Wir erhalten den folgenden Output:

```
The intersections are at 0.5893877634693506 and 2.1267998926782568.
```

(2)

Die gegebene Funktion f ist konvex; für einen beliebigen Startwert x_0 mit $f'(x_0) \neq 0$ konvergiert daher das Newton-Verfahren gegen eine der beiden Nullstellen. Der einzige kritische Punkt ist daher der eindeutige Wert $x \in \mathbb{R}$ mit $f'(x) = 0$. Wir bestimmen diesen Wert näherungsweise:

```
1 fprime = prime(f)
2 print("The newton method cannot start at {}".format(newton_ext(fprime, 1)))
```

Wir erhalten das folgende Ergebnis:

```
The newton method cannot start at 1.4436349751811388.
```

Exercise 4.16

Wir passen zunächst die bisherige `newton`-Methode an, um mit `scipy.array` zu arbeiten:

```
1 from scipy import *
2 from scipy.linalg import *
3
4 class TimeOutError(Exception):
5     pass
6
7 def newton(f, Df, x):
8     eps = 1.E-6 # when to stop
9     xold = x      # current value
10    n = 1        # current iteration
11    while n <= 100:
12        D = Df(xold)
13        xnew = xold - inv(D) @ f(xold)
14        if norm(xnew - xold) < eps:
15            return xnew
16        xold = xnew
17        n += 1
18    raise TimeOutError("the calculation takes too long")
```

Anschließend bestimmen wir die gesuchte Nullstelle:

```

1 def f(p):
2     (x,y,z) = p
3     xnew = 9*x**2 + 36*y**2 + 4*z**2 - 36
4     ynew = x**2 - 2*y**2 - 20*z
5     znew = x**2 - y**2 + z**2
6     return (xnew, ynew, znew)
7
8 def J(p):
9     (x,y,z) = p
10    D = array([[18*x, 72*y, 8*z], [2*x,-4*y,-20],[2*x,-2*y,2*z]])
11    return D
12
13 x0 = [(1,1,0), (1,-1,0), (-1,1,0), (-1,-1,0)]
14 print("initial value\root")
15 for i in range(4):
16     print( "{}\t{}.{format(x0[i], newton(f, J, x0[i]))} )
```

Wir erhalten den folgenden Output:

initial value	root
(1, 1, 0)	[0.89362823 0.89452701 -0.04008929].
(1, -1, 0)	[0.89362823 -0.89452701 -0.04008929].
(-1, 1, 0)	[-0.89362823 0.89452701 -0.04008929].
(-1, -1, 0)	[-0.89362823 -0.89452701 -0.04008929].

Exercise 4.17

Wir nutzen die Methoden `quad` und `romberg` mit den Standardoptionen, und die Methoden `trapz` und `simps` mit einer Unterteilung der jeweiligen Intervalle in 1000 gleichmäßige Teilintervalle:

```

1 from scipy import sin, exp, pi, linspace
2 from scipy.integrate import quad, romberg, trapz, simps
3
4 # defining the functions
5 f = sin
6 def g(x): return 3**(3*x-1)
7 def h(x): return exp(x**2)
8
9 # defining the intervals
10 x1 = linspace(0, pi, 1000)
11 x2 = linspace(0, 2, 1000)
12 x3 = linspace(0, 1, 1000)
13
14 # creating the output
15 print( "sin(x) from 0 to pi ", "3^(3x-1)" )
16 print( "from 0 to 2", "e^(x^2) from 0 to 1" )
17 print( "quad:   {:19.17f} {:21.17f} {:20.17f}" .format( quad(f,0,pi)[0], quad(g,0,2)[0], quad(h,0,1)[0] ) )
18 print( "romberg: {:19.17f} {:21.17f} {:20.17f}" .format( romberg(f,0,pi),
19             romberg(g,0,2), romberg(h,0,1) ) )
20 print( "trapz:   {:19.17f} {:21.17f} {:20.17f}" .format( trapz(f,x1), x1,
21             trapz(g,x2), trapz(h,x3) ) )
```

```

19 print( "simps:   {:19.17f} {:21.17f} {:20.17f}" .format( simps(f(x1), x1),
    simps(g(x2), x2), simps(h(x3), x3) ) )

```

Wir erhalten den folgenden Output:

	$\sin(x)$ from 0 to pi	$3^{(3x-1)}$ from 0 to 2	$e^{(x^2)}$ from 0 to 1
quad:	2.0000000000000000	73.62823966492641148	1.46265174590718150
romberg:	2.00000000000132117	73.62823966494875094	1.46265174591010316
trapz:	1.99999835177085195	73.62850679530863829	1.46265219986153205
simps:	1.99999999999701172	73.62824054651866845	1.46265174667154541

Exercise 4.18

Wir nutzen die folgende Funktion, um eine gegebene quadratische Matrix A als $A = L + D + U$ wie in der Aufgabenstellung zu zerlegen:

```

1 def ldu(A):
2     (n, m) = A.shape
3     if n != m:
4         return ValueError("matrix is not square")
5     L = zeros((n,n))
6     D = zeros((n,n))
7     U = zeros((n,n))
8     for i in range(n):
9         # splitting up the i-th row
10        L[i,:i] = A[i,:i]
11        D[i,i] = A[i,i]
12        U[i, i+1:] = A[i, i+1:]
13    return(L,D,U)

```

Wir bestimmen nun zunächst die exakte Lösung mithilfe des folgenden Codes:

```

1 A = array([[4,3,0],[3,4,-1],[0,-1,4]])
2 print("A:")
3 print(A)
4 b = array([24,30,-24])
5 print("b:")
6 print(b)
7
8 exact = solve(A, b)
9 print("The solution to Ax = b is:")
10 print(exact)

```

Anschließend berechnen wir mithilfe des Gauß-Seidel-Algorithmus eine approximative Lösung:

```

1 L,D,U = ldu(A)
2 n = 3 # iterations steps
3 x = array([3,3,3]) # starting point
4 for i in range(n+1):
5     x = inv(D + L) @ (b - U @ x)
6
7 print("An approximate solution to Ax = b is:")
8 print(x)

```

```

9| print("The difference is:")
10| print(exact - x)

```

Wir erhalten den folgenden Output:

```

A:
[[ 4  3  0]
 [ 3  4 -1]
 [ 0 -1  4]]
b:
[ 24  30 -24]
The solution to Ax = b is:
[ 3.  4. -5.]
An approximate solution to Ax = b is:
[ 2.57885742  4.35095215 -4.91226196]
The difference is:
[ 0.42114258 -0.35095215 -0.08773804]

```

Exercise 4.19

Wir interpolieren die gegebenen Werte mit einem Polynom p vom Grad 6. Hierfür nutzen wir die Funktion `KroghInterpolator` aus dem Paket `scipy.interpolate`; mit dieser lassen sich die Ableitungen $p^{(n)}(0)$ bestimmen, aus denen sich dann die Koeffizienten bestimmen lassen:

```

1 from scipy import *
2 from scipy.misc import factorial
3 from scipy.interpolate import KroghInterpolator
4
5 # the interpolation itself
6
7 xarr = linspace(0,3,7)
8 f = [1,1,0,0,3,1,2]
9 p = KroghInterpolator(xarr, f)
10
11 # getting the coefficients
12
13 deriv = p.derivatives(0)
14 coeff = []
15 for n in range(len(deriv)):
16     coeff.append(deriv[n]/factorial(n))
17
18 print("The coefficients (via interpolation):")
19 print(coeff)

```

Zur Überprüfung unserer Ergebnisses bestimmen wir die Koeffizienten anschließend noch einmal durch ein entsprechendes lineares Gleichungssystem:

```

1 # comparing to a different approach
2
3 from scipy.linalg import solve
4 A = zeros((7,7))
5 for i in range(len(xarr)):

```

```

6|     for j in range(len(f)):
7|         A[i,j] = xarr[i]**j

```

Wir erhalten den folgenden Output:

```

The coefficients (via interpolation):
[1.0, -13.66666666666666, 65.46666666666664, -110.66666666666664,
 81.333333333331, -26.66666666666666, 3.1999999999999993]
The coefficients (via linear equations) are:
[ 1.          -13.6666667   65.4666667 -110.6666667   81.3333333
 -26.6666667    3.2        ]

```

Beide Ergebnisse stimmen überein.

Exercise 4.20

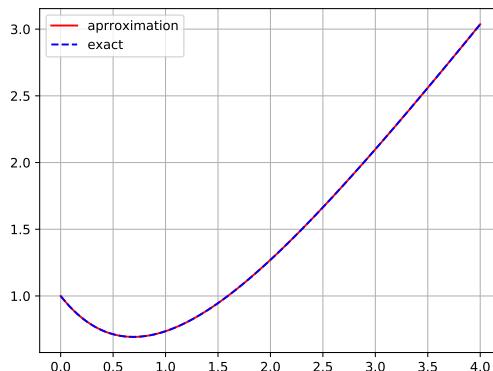
Wir nutzen den folgenden Code:

```

1 from scipy import *
2 from scipy.integrate import odeint
3 import matplotlib.pyplot as plt
4
5 def du(u, x): return x - u # the ode
6 x = linspace(0,4,100)      # interval [0,4]
7 u0 = 1.0                   # left boundary condition
8 y = odeint(du, u0, x)      # solve ode
9
10 # plotting the graphs
11 plt.clf()
12 plt.plot(x, y, "r-", label="apprroximation")
13 plt.plot(x, x - 1 + 2*exp(-x), "b--", label="exact")
14 plt.grid()
15 plt.legend()
16 plt.show()

```

Wir erhalten damit das folgende Bild, in denen die beiden Graphen praktisch nicht unterscheidbar sind:

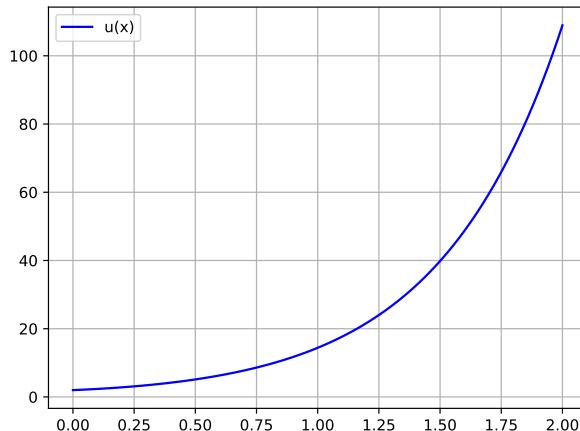


Exercise 4.21

Wir nutzen den folgenden Code:

```
1 from scipy import *
2 from scipy.integrate import odeint
3 import matplotlib.pyplot as plt
4
5 def f(y, x): # the ode
6     (u, uprim) = y
7     dy = (uprim, (3*x+2)/(3*x-1)*uprim + (6*x-8)/(3*x-1)*u)
8     return dy
9 u0 = (2, 3) # initial values
10
11 # solving the ode on the interval [0,2]
12 xval = linspace(0, 2, 100)
13 sol = odeint(f, u0, xval)
14 yval = sol[:,0]
15
16 # plotting the solution
17 plt.clf()
18 plt.plot(xval, yval, color='b', label="u(x)")
19 plt.grid()
20 plt.legend()
```

Wir erhalten den folgenden Graphen:



Exercise 4.22

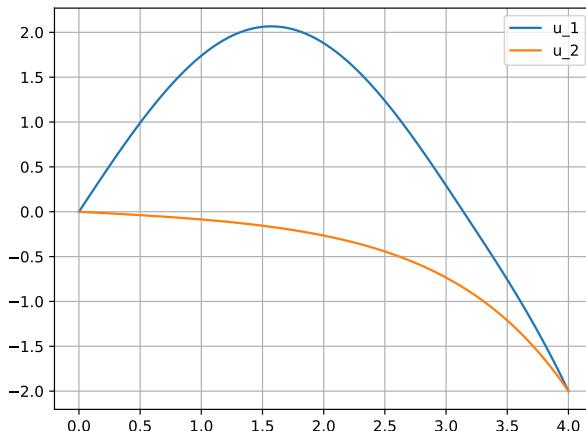
In der ursprünglichen Version der Aufgabe war fehlerhaft $u' = -|u|$ gefordert. Der Vollständigkeit halber betrachten wir beide Formulierungen.

Version $u'' = -|u|$

Wir nutzen den folgenden Code:

```
1 from scipy import *
2 from scipy.integrate import solve_bvp
3 import matplotlib.pyplot as plt
4
5 def fun(x,y): return [y[1], -abs(y[0])] # differential equation
6 def bc(yl, yr): return [yl[0], yr[0]+2] # boundary condition
7
8 x = linspace(0,4,2); # interval for solving
9
10 y_1 = zeros((2, len(x))) # first initial guess
11 y_1[1][0] = 1 #  $u'(0) > 0$ 
12 y_2 = zeros((2, len(x))) # second initial guess
13 y_2[1][0] = -1 #  $u'(0) < 0$ 
14
15 res_1 = solve_bvp(fun, bc, x, y_1) # solve with first guess
16 res_2 = solve_bvp(fun, bc, x, y_2) # solve with second guess
17
18 x = linspace(0, 4, 100) # interval for plotting
19 u_1 = res_1.sol(x)[0] # y values for  $u_1$ 
20 u_2 = res_2.sol(x)[0] # y values for  $u_2$ 
21
22 # plotting
23 plt.clf()
24 plt.plot(x, u_1, label='u_1')
25 plt.plot(x, u_2, label='u_2')
26 plt.grid()
27 plt.legend()
28 plt.show()
```

Wir erhalten damit die folgenden Lösungsgraphen:



Version $u' = -|u|$

Das gegebene Randwertproblem

$$u' = -|u(x)|, \quad u(0) = 0, \quad u(4) = -2$$

für $u \in C^1[0, 1]$ hat keine Lösung:

Die Funktion u wäre monoton fallend, da $u'(x) = -|u(x)| \leq 0$ für alle $x \in [0, 4]$ gilt.
Daher ist

$$I := \{x \in [0, 4] \mid u(x) < 0\}$$

ein Intervall mit Randpunkt 4. Wegen der Stetigkeit von u ist I halboffen; es gibt also $x_0 \in [0, 4]$ mit $I = (x_0, 4]$. Da $u(0) = 0$ gilt, ist dabei $x_0 > 0$.

Auf dem Intervall I gilt nun $-|u| = u$. Also erfüllt u auf I die Differenzialgleichung $u' = u$. Somit gibt es eine Konstante $c \in \mathbb{R}$ mit $u(x) = ce^x$ für alle $x \in I$. Es gilt

$$-2 = u(4) = ce^4$$

und somit $c = -2e^{-4}$. Dann gilt aber

$$u(x_0) = \lim_{x \downarrow x_0} u(x) = \lim_{x \downarrow x_0} -2e^{x-4} = -2e^{x_0-4} < 0,$$

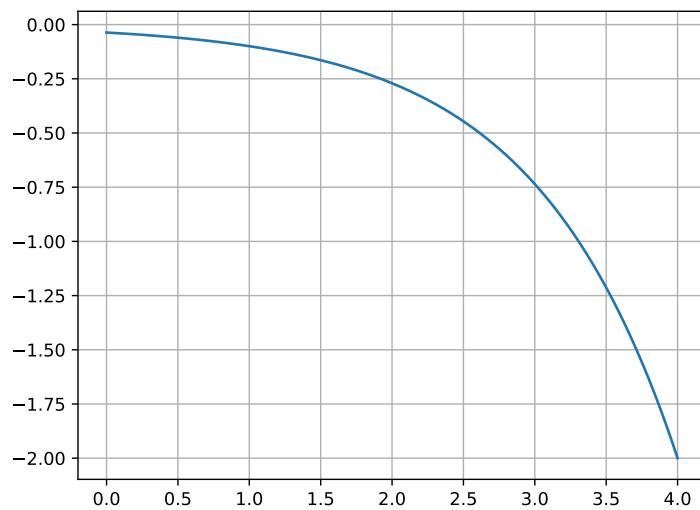
im Widerspruch zu $x_0 \notin I$.

Unsere obige Argumentation spiegelt sich auch in dem Verhalten von `scipy` wieder:
Das folgende Programm würde eine Lösung der Differenzialgleichung liefern und plotten:

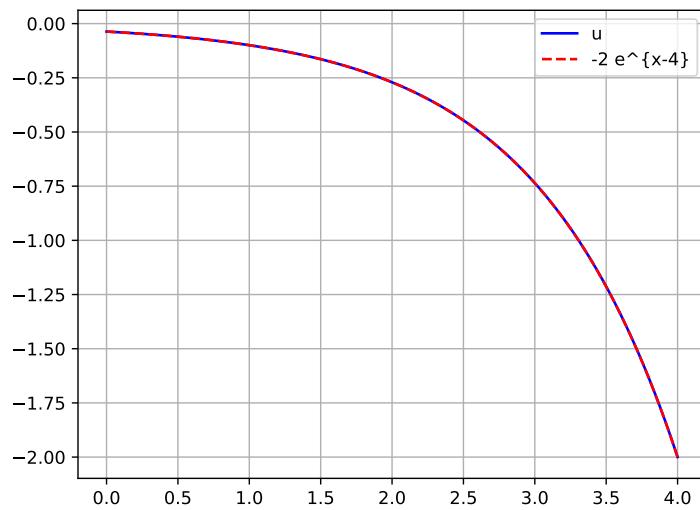
```

1 from scipy import *
2 from scipy.integrate import solve_bvp
3 import matplotlib.pyplot as plt
4
5 def f(x,y): return -abs(y) # the differential equation
6 def bc(yl, yr): return abs(yl) + abs(yr+2) # boundary conditions
7
8 I = linspace(0, 4) # interval for solving
9
10 y = zeros((1, len(I))) # initial guess
11 y[0][0] = 1 # u(0) = 0
12
13 res = solve_bvp(f, bc, I, y) # solving the ode
14
15 x = linspace(0, 4, 100) # interval for plotting
16 u = res.sol(x)[0] # yval of solution
17
18 # plotting solution
19 plt.clf()
20 plt.plot(x, u)
21 plt.grid()
22 plt.show()
```

Wir erhalten den folgenden Graphen:



Die Randbedingung $u(4) = -2$ wird zwar beachtet, die Randbedingung $u(0) = 0$ hingegen nicht. Der eingezeichnete Graph ist, wie auch aus der obigen Argumentation hervorgeht, der Graph von $-2e^{x-4}$:



Exercise 4.23

(1)

Wir plotten L durch den folgenden zusätzlichen Code:

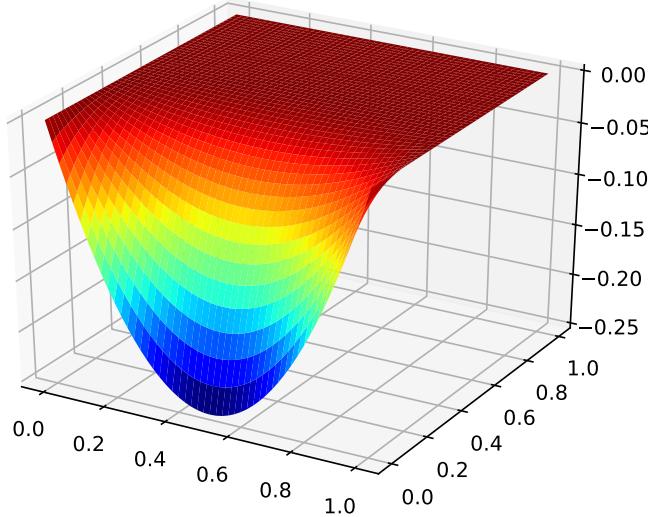
```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
```

```

3| from matplotlib import cm
4|
5| fig = plt.figure()
6| ax = fig.gca(projection='3d')
7| x, y = meshgrid(x, y)
8| surf = ax.plot_surface(x, y, L, rstride=1, cstride=1, cmap=cm.jet, linewidth
9| =0)
| plt.show()

```

Wir erhalten hierdurch den folgende Graphen:



(2)

Das Programm berechnet für $\Omega = (0, 1) \times (0, 1)$ eine approximative Lösung der Differentialgleichung

$$\Delta u = 0 \text{ auf } \Omega \quad \text{und} \quad u = g \text{ auf } \partial\Omega.$$

wobei

$$g(x, y) = \begin{cases} y(y-1) & \text{falls } x = 0, \\ 0 & \text{sonst.} \end{cases}$$

Dabei soll für alle für alle $i, j = 0, \dots, 50$ der Eintrag $L[i, j]$ approximativ dem Funktionswert $u(i/50, j/50) =: u_{ij}$ entsprechen. Die aus

$$u = g \text{ auf } \partial\Omega$$

folgenden Randbedingungen werden durch die folgende Zeile des Codes festgelegt:

```

1| L[0, :] = y*(y-1)

```

Man bemerke, dass die Randwerte $L[i,j]$ (mit $i = 0, 50$ und $j = 0, 50$) im Laufe des Programmes unverändert bleiben. Die Gleichung

$$\Delta u = 0 \text{ auf } \Omega$$

wird mit

$$\frac{1}{h^2}(-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1}) = f(ih, jh) \stackrel{!}{=} 0$$

(siehe Abschnitt 4.8, Seite 66 im Skript) durch die folgende Gleichung des Codes implementiert:

```
1 L[i,j] = (Lt[i+1,j] + Lt[i-1,j] + Lt[i,j+1] + Lt[i,j-1]) / 4
```

Der Code funktioniert also dadurch, dass auf dem Rand mit den exakten Werten für u_{ij} begonnen wird, und diese dann mithilfe der Gleichung

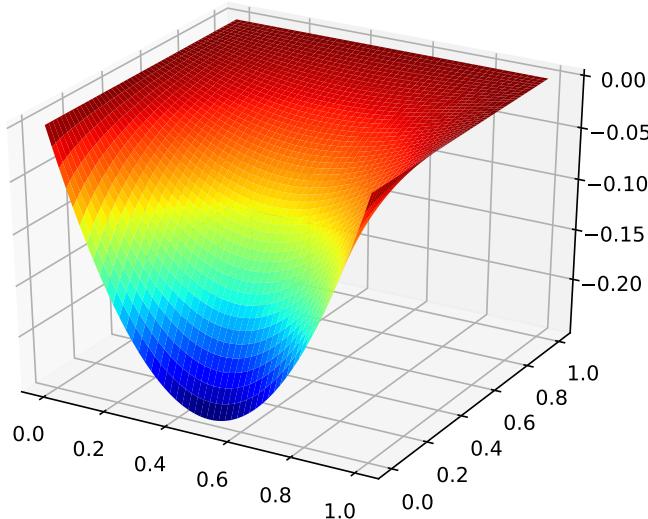
$$u_{ij} = \frac{u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}}{4}$$

auch nach innen hin propagiert werden.

Um unsere Vermutung zu überprüfen, bestimmen wir mithilfe der gegebenen Methode `poisson_solver` die exakte Lösung, und plotten diese:

```
1 from poisson solver import *
2 f = (lambda x,y: 0)
3 def g(x,y):
4     if x == 0:
5         return y*(y-1)
6     return 0
7 poisson_solver(f, g, 51)
```

Wir erhalten den folgenden Graphen:



Der Vergleich mit dem obigen Graphen bestätigt unsere Vermutung.

5 Python in Computer Algebra, SymPy

Exercise 5.24

(1)

Wir testen, für welche Werte von α die Determinante von A verschwindet:

```
1 from sympy import *
2
3 alpha, beta, x, y, z = symbols('alpha beta x y z')
4
5 A = Matrix([[1,1,1],[1,0,-1],[alpha,-1,1]])
6 b = Matrix([3,beta,-3])
7 print( "The kernel is nonzero for the following values of alpha:" )
8 print( solveset(det(A)) )
```

Wir erhalten den folgenden Output:

```
The kernel is nonzero for the following values of alpha:
{-3}
```

Somit hat das lineare Gleichungssystem $Ax = 0$ nur für $\alpha = -3$ nicht-triviale Lösungen.

(2)

Die Spalten von A sind genau linear abhängig, wenn $\det A = 0$ gilt. Nach dem vorherigen Aufgabenteil gilt dies nur für $\alpha = -3$.

(3)

Für $\alpha \neq -3$ ist die Matrix invertierbar, sodass das Gleichungssystem $Ax = b$ dann für jedes $\beta \in \mathbb{R}$ eine eindeutige Lösung hat. Für $\alpha = 3$ hat die Matrix A immer noch Rang 2, weshalb sich die Werte für β in diesem Fall wie folgt bestimmen lässt:

```
1 (v,w) = A.subs(alpha,-3).columnspace() # column space is 2-dimensional
2 B = v.col_insert(1, w).col_insert(2, b) # put v,w,b into a matrix
3 print("For alpha = -3 there exists a solution for the following values of
      beta:")
4 print( solveset(det(B)) )
```

Wir erhalten den folgende Output:

```
For alpha = -3 there exists a solution for the following values of beta:
{0}
```

(4)

Für $\alpha = -3$, $\beta = 0$ berechnen wir die Lösungen für $Ax = b$ mit `linsolve`:

```
1 print("For alpha = -3, beta = 0 the solutions are given as follows:")
2 print( linsolve( [Eq(x+y+z,3), Eq(x-z,0), Eq(-3*x-y+z,-3) ], [x,y,z] ) )
```

Wir erhalten den folgenden Output:

```
For alpha = -3, beta = 0 the solutions are given by the following set:
{(z, -2*z + 3, z)}
```

Exercise 5.25

Wir bestimmen zunächst die Eigenwerte und zugehörigen Eigenvektoren:

```
1 from sympy import *
2
3 eps = symbols('eps')
4 A = Matrix([[1 + eps*cos(2/eps), -eps*sin(2/eps)], [-eps*sin(2/eps), 1 + eps*
    cos(2/eps)]])
5
6 eig = A.eigenvecs()
7 lambda1 = simplify(eig[0][0]) # first eigenvector
8 lambda2 = simplify(eig[1][0]) # second eigenvalue
9 phi1 = simplify(eig[0][2][0]) # first eigenvector
10 phi2 = simplify(eig[1][2][0]) # second eigenvector
11
12 print("The eigenvalues are:")
13 print(lambda1)
14 print(lambda2)
15 print("The corresponding eigenvectors are:")
16 print(phi1)
17 print(phi2)
```

Wir erhalten den folgenden Output:

```
The eigenvalues are:
sqrt(2)*eps*cos(pi/4 - 2/eps) + 1
sqrt(2)*eps*cos(pi/4 + 2/eps) + 1
The corresponding eigenvectors are:
Matrix([[-1], [1]])
Matrix([[1], [1]])
```

Inbesondere sind die Eigenwerte φ_1 und φ_2 unabhängig von ϵ . Wir bestimmen nun die Grenzwerte von $A(\epsilon)$ und $\lambda_i(\epsilon)$ für $\epsilon \rightarrow 0$:

```
1 print("For eps -> 0 the matrix A becomes:")
2 B = A.applyfunc( (lambda x: limit(x, eps, 0)) ) # apply limit entrywise
3 print( B )
4 print("For eps -> 0 the eigenvalues become:")
5 print( limit(lambda1, eps, 0) )
6 print( limit(lambda2, eps, 0) )
```

Wir erhalten die folgenden Grenzwerte:

```
For eps -> 0 the matrix A becomes:
Matrix([[1, 0], [0, 1]])
For eps -> 0 the eigenvalues become:
1
1
```

Exercise 5.26

Die Abbildung $f: \mathbb{R} \rightarrow \mathbb{R}$, $x \mapsto x^3 + 3x$ ist stetig mit $\lim_{x \rightarrow -\infty} f(x) = -\infty$ und $\lim_{x \rightarrow \infty} f(x) = \infty$; nach dem Zwischenwertsatz ist f deshalb surjektiv. Außerdem ist f differenzierbar mit $f'(x) = 3x^2 + 3 > 0$ für alle $x \in \mathbb{R}$, weshalb f streng monoton steigend, und somit injektiv ist. Die Abbildung f ist also bijektiv, weshalb die Gleichung $f(x) = a$ für jedes $a \in \mathbb{R}$ eine eindeutige reelle Lösung besitzt.

(Leider haben wir es auch nach viel herumprobieren nicht geschafft, `sympy` richtig entscheiden zu lassen, welche der drei Nullstellen jeweils reell sind. Daher argumentieren wir diesen Teil rein mathematisch ohne zugehörigen Code.)

Mithilfe von `scipy` lassen sich die drei verschiedenen Lösungen bestimmen:

```
1 from sympy import *
2
3 a, x = symbols('a x')
4 sol = solveset(x**3 + 3*x - a, x)
5 print(sol)
```

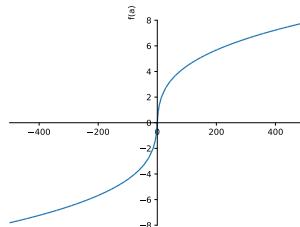
Wir erhalten die folgenden Lösungen:

```
{-(-1/2 - sqrt(3)*I/2)*(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)/3 +
 3/((-1/2 - sqrt(3)*I/2)*(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)),
 -(-1/2 + sqrt(3)*I/2)*(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)/3 +
 3/((-1/2 + sqrt(3)*I/2)*(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)),
 -(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)/3 + 3/(-27*a/2 + sqrt(729*a**2 + 2916)/2)**(1/3)}
```

Bei der ersten der drei Lösungen handelt es sich um die reelle Lösung. Wir können diese im geforderten Intervall $[-500, 500]$ mit dem folgenden Code plotten:

```
1 f = list(sol)[0]
2 plot(f, (a, -500, 500))
```

Damit erhalten wir die folgenden Graphen:



Exercise 5.27

Wir passen die bisherige newton-Methode dahingehend an, dass sie anstelle einer Funktion f einen „Funktionsausdruck“ der Form $\exp(x) + 2*x$ annimmt, und diesen intern in eine entsprechende Funktion umwandelt.

```
1 def newton(f, var, x0): # var = variable name
2     fprime = f.diff(var)
3     g = lambdify(var, f)
4     gprime = lambdify(var, fprime)
5     eps = 1.E-7 # when to stop
6     xold = x0    # current value
7     n = 1        # current iteration
8     while n <= 100:
9         d = gprime(xold)
10        if d == 0:
11            raise ZeroDivisionError("derivative vanishes at {}".format(xold))
12        xnew = xold - g(xold)/d
13        if abs(xnew - xold) < eps:
14            return xnew
15        xold = xnew
16        n += 1
17    raise TimeOutError("the calculation takes too long")
```

Hiermit bestimmen wir die Lösungen der Gleichungen $e^x + 2x = 0$ und $\cosh(x) = 2x$, wobei wir die gleichen Startwerte wie in den entsprechenden vorherigen Aufgaben nutzen:

```
1 x = symbols('x')
2 f = exp(x) + 2*x
3 g = cosh(x) - 2*x
4
5 x0 = 1
6 print("The root of e^x + 2x is {}".format(newton(f, x, 1)))
7
8 x1 = 0.5
9 x2 = 2
10 print("The functions cosh(x) and 2x intersect at {} and {}".format(newton(g,
   x1), newton(g, x, x2)))
```

Wir erhalten in der Konsole die folgenden Ergebnisse:

```
The root of e^x + 2x is -0.3517337112491958.
The functions cosh(x) and 2x intersect at 0.5893877634693505 and
2.1267998926782568.
```

Dies sind die gleichen Ergebnisse wie zuvor.

Exercise 5.28

Wir nutzen den folgenden Code:

```
1 from sympy import *
2
```

```

3| class TimeOutError(Exception):
4|     pass
5|
6| # f   : a matrix of expressions
7| # var : list of appearing variables
8| def newton(f, var, x0):
9|     n = len(var)
10|    if n != len(f):
11|        raise ValueError("wrong function type")
12|    Df = Matrix(n, n, (lambda i,j: f[i].diff(var[j])))
13|    def g(x):  # make f into a function
14|        sublist = list(zip(var, x))
15|        substitutor = (lambda e: e.subs( sublist ))
16|        return f.applyfunc( substitutor )
17|    def Dg(x): # make Df into a function
18|        sublist = list(zip(var, x))
19|        substitutor = (lambda e: e.subs( sublist ))
20|        return Df.applyfunc( substitutor )
21|    eps = 1.E-7 # when to stop
22|    xold = x0  # current value
23|    n = 1      # current iteration
24|    while n <= 100:
25|        D = Dg(xold)
26|        xnew = xold - D**(-1) @ g(xold)
27|        if (xnew - xold).norm() < eps:
28|            return xnew
29|        xold = xnew
30|        n += 1
31|    raise TimeOutError("the calculation takes too long")

```

Wir testen unser Programm anhand der gegebenen Funktion:

```

1 x, y, z = symbols('x y z')
2
3 f = Matrix([9*x**2 + 36*y**2 + 4*z**2 - 36, x**2 - 2*y**2 - 20*z, x**2 - y**2
4           + z**2])
5 x0 = [ Matrix([1,1,0]), Matrix([1,-1,0]), Matrix([-1,1,0]), Matrix([-1,-1,0]) ]
6 print("initial value \t\t\troot")
7 for i in range(4):
8     print( "{} \t\t\t".format(x0[i], newton(f, [x,y,z], x0[i]).applyfunc(float
9         )) )

```

Wir erhalten in der Konsole den folgenden Output:

```

initial value          root
Matrix([[1], [1], [0]])      Matrix([[0.893628234476483],
[0.894527010390578], [-0.0400892861591528]])
Matrix([[1], [-1], [0]])      Matrix([[0.893628234476483],
[-0.894527010390578], [-0.0400892861591528]])
Matrix([[-1], [1], [0]])      Matrix([[-0.893628234476483],
[0.894527010390578], [-0.0400892861591528]])
Matrix([[-1], [-1], [0]])      Matrix([[-0.893628234476483],
[-0.894527010390578], [-0.0400892861591528]])

```

Exercise 5.29

Wir bestimmen zunächst die allgemeine Lösung ohne Betrachtung des Anfangswertes:

```
1 from sympy import *
2
3 u = Function('u')
4 x = symbols('x')
5
6 ode = Eq(u(x).diff(x) + u(x), x)
7 gen_sol = dsolve(ode, u(x))
8 expr = simplify(gen_sol.rhs)
9 print("The general solution is: {}".format(expr))
```

Anschließend nutzen wir noch die Anfangsbedingung:

```
1 C1 = symbols('C1') # constant appearing the general solution
2 b = solve( Eq(expr.subs(x,0),1) )
3 expr2 = expr.subs(C1, b[0])
4 print("The initial value results in the solution: {}".format(expr2))
```

Wir erhalten in der Konsole den folgenden Output:

```
The general solution is: C1*exp(-x) + x - 1
The initial value results in the solution: x - 1 + 2*exp(-x)
```

Exercise 5.30

Wir lösen zunächst die Differentialgleichung ohne Betrachtung der Randwerte:

```
1 from sympy import *
2
3 u = Function('u')
4 x = symbols('x')
5
6 ode = Eq(u(x).diff(x,x), u(x))
7 gen_sol = dsolve(ode, u(x))
8 expr = simplify(gen_sol.rhs)
9 print("The general solution is {}".format(expr))
```

Anschließend nutzen wir noch die Anfangsbedingungen:

```
1 C1, C2 = symbols('C1 C2') # constants appearing in the general solution
2 b = solve( [Eq(expr.subs(x,0),0), Eq(expr.diff(x).subs(x,1),-1)], [C1, C2] )
3 expr2 = simplify(simplify(expr.subs(b))) # simplify not idempotent
4 print("The boundary values lead to the solution {}".format(expr2))
```

(Das doppelte Anwenden von `simplify` führt hier zu einem lesbareren Ergebnis als nur einmaliges Anwenden.) Wir erhalten in der Konsole den folgenden Output:

```
The general solution is C1*exp(-x) + C2*exp(x)
The boundary values lead to the solution -2*E*sinh(x)/(1 + exp(2))
```

Exercise 5.31

Wir bestimmen die Skalarprodukte $\langle p_i, p_j \rangle$ und tragen diese in eine Matrix ein:

```
1 from sympy import *
2 x = symbols('x')
3
4 def inner(f, g):
5     return integrate( f*g, (x, 0, 1) )
6
7 p = [ 1, x-sympify(1)/2, x**2 - x + sympify(1)/6 ]
8 B = Matrix( len(p), len(p), (lambda i,j: inner(p[i], p[j])) )
9
10 print(B)
```

Wir erhalten die folgende Matrix:

```
Matrix([[1, 0, 0], [0, 1/12, 0], [0, 0, 1/180]])
```

Dies ist eine Diagonalmatrix, was die paarweise Orthogonalität der p_i zeigt.

Exercise 5.32

Wir berechnen die Legendre-Polynome, indem wir auf die Polynome $1, x, x^2, \dots, x^n$ das Gram-Schmidt-Orthogonalisierungsverfahren anwenden:

```
1 from sympy import symbols, Matrix, integrate
2
3 x = symbols('x')
4
5 def inner(f,g):
6     return integrate( f*g, (x, -1, 1) )
7
8 def legendre(n):
9     p = [x**i for i in range(n+1)]
10    normsq = [] # list of squared norms
11    for i in range(n+1):
12        s = 0
13        for j in range(i):
14            s += inner(p[i], p[j])/normsq[j] * p[j]
15        p[i] -= s
16        normsq.append( inner(p[i], p[i]) )
17
18 return p
```

Wir testen die Orthogonalität der ersten 6 Legendre-Polynome:

```
1 p = legendre(6)
2 print( Matrix(6, 6, (lambda i,j: inner(p[i], p[j]))) )
```

Wir erhalten den folgenden Output:

```
Matrix([[2, 0, 0, 0, 0, 0], [0, 2/3, 0, 0, 0, 0], [0, 0, 8/45, 0, 0, 0], [0,
0, 0, 8/175, 0, 0], [0, 0, 0, 0, 128/11025, 0], [0, 0, 0, 0, 0, 128/43659]])
```

Es handelt sich um eine Diagonalmatrix, was die Orthogonalität zeigt.

7 The language C++

Exercise 7.33

Wir nutzen den folgenden Code:

```
1 #include <cmath>
2 #include <iostream>
3
4 #define PI 3.14159265358979323846
5
6 double simpson(double f(double), double a, double b, int n){
7     double h = (b-a)/n;          // length of subintervals
8     double result = f(a);       // current sum
9     double x = a;              // left border of current subinterval
10    for(int i = 1; i < n; i++) {
11        result += 4*f(x+h/2);
12        result += 2*f(x+h);
13        x += h;
14    }
15    result -= f(b);
16    result *= h/6;
17    return result;
18 }
19
20 int main(){
21     std::cout.precision(10); // set output precision
22     std::cout << simpson(sin,0,PI,2000) << std::endl;
23     return 0;
24 }
```

Hierdurch erhalten wir den folgenden Output:

```
$ g++ exercise_07_33.cpp -o exercise_07_33
$ ./exercise_07_33
1.999999178
```

Exercise 7.34

Wir nutzen eine selbstgeschriebene **Matrix**-Klasse mit der folgenden Header-Datei:

```
1 #include <vector>
2
3 #define EPS 1.E-14
4
5
```

```

6 class Matrix{
7     std::vector<std::vector<double>> mat;
8     int rows, cols;
9
10    public:
11        Matrix();
12        Matrix(int m, int n);
13        Matrix(int m, int n, double v);
14
15        // informations about the matrix
16        double& operator() (int i, int j);
17        int width();
18        int height();
19
20
21        // matrix operations
22        Matrix operator-();
23        Matrix operator+(Matrix B);
24        Matrix operator-(Matrix B);
25        Matrix operator*(Matrix B);
26
27        // elementary row operations
28        void permuteRows(int i, int j);
29        void multiplyRow(int i, double c);
30        void addRowToFrom(int i, int j, double c);
31
32    // comparing and printing
33    Matrix clean();
34    bool operator==(Matrix B);
35    void print();
36}
37
38
39 // solving Ax = y
40 std::vector<double> gaussSolve(Matrix A, std::vector<double> y);

```

Der konkrete Code der Matrix-Klasse ist wie folgt gegeben:

```

1 #include "matrix.hpp"
2 #include <cmath>
3 #include <vector>
4 #include <iostream>
5
6 #define EPS 1.E-14
7
8
9
10 Matrix::Matrix(){
11     mat = {};
12     rows = 0;
13     cols = 0;
14 }
15
16 Matrix::Matrix(int m, int n){
17     rows = m;
18     cols = n;
19     mat = std::vector<std::vector<double>>(rows, std::vector<double>(cols,0));

```

```

20 }
21 Matrix::Matrix(int m, int n, double v){
22     rows = m;
23     cols = n;
24     mat = std::vector<std::vector<double>>(rows, std::vector<double>(cols, v));
25 }
26
27 // informations about the matrix
28
29 int Matrix::height(){
30     return cols;
31 }
32
33 int Matrix::width(){
34     return rows;
35 }
36
37
38 double& Matrix::operator()(int i, int j){ // indices start at 0
39     return mat[i][j];
40 }
41
42
43 // matrix operations
44
45
46 Matrix Matrix::operator-(){
47     Matrix C((*this).height(), (*this).width());
48     for (int i = 0; i < C.height(); i++)
49         for (int j = 0; j < C.width(); j++)
50             C(i,j) = -(*this)(i,j);
51     return C;
52 }
53
54 Matrix Matrix::operator+(Matrix B){
55     if( rows != B.rows || cols != B.cols ){
56         std::cerr << "Error: Trying to add matrices with different dimensions."
57             << std::endl;
58         exit(1);
59     }
60     Matrix C(rows,cols);
61     for (int i = 0; i < rows; i++)
62         for (int j = 0; j < cols; j++)
63             C(i,j) = (*this)(i,j) + B(i,j);
64     return C;
65 }
66
67 Matrix Matrix::operator-(Matrix B){
68     if( rows != B.rows || cols != B.cols ) {
69         std::cerr << "Error: Trying to add matrices with different dimensions."
70             << std::endl;
71         exit(1);
72     }
73     Matrix C(rows,cols);
74     for (int i = 0; i < rows; i++)
75         for (int j = 0; j < cols; j++)

```

```

74     C(i,j) = (*this)(i,j) - B(i,j);
75     return C;
76 }
77
78 Matrix Matrix::operator*(Matrix B){
79     if(cols != B.rows){
80         std::cerr << "Error: Trying to multipliy matrices with wrong dimensions."
81             << std::endl;
82         exit(1);
83     }
84     Matrix C(rows, B.cols);
85     for (int i = 0; i < rows; i++)
86         for (int k = 0; k < cols; k++)
87             for (int j = 0; j < B.cols; j++)
88                 C(i,j) += (*this)(i,k) * B(k,j);
89     return C;
90 }
91 // elementary row operations
92
93 void Matrix::permuteRows(int i, int j){ // swap rows i <-> j
94     std::swap(mat[i],mat[j]);
95 }
96
97 void Matrix::multiplyRow(int i, double c){ // multipliy row i -> c*i
98     for(int j = 0; j < cols; j++)
99         mat[i][j] *= c;
100 }
101
102 void Matrix::addRowToFrom(int i, int j, double c) { // add row i -> i + c*j
103     for (int k = 0; k < cols; k++)
104         mat[i][k] += c * mat[j][k];
105 }
106
107 // comparing and printing
108
109 Matrix Matrix::clean(){
110     Matrix B = Matrix(rows, cols);
111     for(int i = 0; i < rows; i++)
112         for(int j = 0; j < cols; j++)
113             if(std::abs((*this)(i,j)) > EPS)
114                 B(i,j) = (*this)(i,j);
115     return B;
116 }
117
118 bool Matrix::operator==(Matrix B){
119     Matrix D = (*this) - B;
120     D = D.clean();
121     for (int i = 0; i < rows; i++)
122         for (int j = 0; j < cols; j++)
123             if (D(i,j) != 0)
124                 return false;
125     return true;
126 }
127
128 void Matrix::print(){

```

```

129 Matrix B = (*this).clean();
130     for (int i = 0; i < rows; i++) {
131         for (int j = 0; j < cols; j++)
132             std::cout << B(i,j) << "\t";
133             std::cout << std::endl;
134     }
135 }
136
137 // solving Ax = y by Gauss-Jordan
138
139 std::vector<double> gaussSolve(Matrix A, std::vector<double> y){
140     int rows = A.height();
141     int cols = A.width();
142     if(y.size() != rows){
143         std::cerr << "Trying to solve LGS with wrong dimensions." << std::endl;
144         exit(1);
145     }
146     std::vector<double> x = y;
147
148     for (int j = 0; j < cols; j++){
149         int k = j;
150         for (int i = j; i < rows; i++) { // find the biggest value in
151             j-th row
152             if (std::abs(A(i,j)) > std::abs(A(k,j)))
153                 k = i;
154         }
155         A.permuteRows(j,k); // biggest value w.l.o.g. in
156             the j-th row
157         std::swap(x[j],x[k]);
158         x[j] *= 1/A(j,j);
159         A.multiplyRow(j, 1/A(j,j));
160         for(int i = j+1; i < rows; i++){
161             x[i] -= x[j]*A(i,j);
162             A.addRowToFrom(i,j,-A(i,j));
163         }
164     }
165     for(int j = 0; j < cols; j++){
166         for(int i = 0; i < j; i++){
167             x[i] -= A(i,j) * x[j];
168             A.addRowToFrom(i, j, -A(i,j));
169         }
170     }
171     return x;
172 }
```

Wir nutzen außerdem die `Polynom`-Klasse aus Aufgabe 37. Für die Interpolation selbst nutzen wir nun das folgende Programm:

```

1 #include "matrix.hpp"
2 #include "polynomial.hpp"
3 #include <cmath>
4 #include <iostream>
5
6 #define PI 3.14159265358979323846
7
8 Matrix vandermonde(std::vector<double> v){
```

```

9   int n = v.size();
10  Matrix V = Matrix(n,n);
11  for(int i=0; i < n; i++)
12    for(int j = 0; j < n; j++)
13      V(i,j) = pow(v[i],j);
14  return V;
15 }
16
17 Polynomial interpol(std::vector<double> points, std::vector<double> values){
18  Matrix V = vandermonde(points);
19  std::vector<double> coeff = gaussSolve(V,values);
20  Polynomial p = Polynomial(coeff);
21  return p;
22 }
23
24 int main(){
25  int n = 5;           // number of points
26  double left = -1.5; // left interval boundary
27  double right = 1.5; // right interval boundary
28  double h = (right - left)/(n-1); // length of subintervals
29
30
31  std::vector<double> v(n), w(n);
32  double x = left;           // current x value
33  for(int i = 0; i < n; i++){ // creates coordinates to be interpolated
34    v[i] = x;
35    w[i] = tan(x);
36    x += h;
37  }
38  Polynomial p = interpol(v,w);
39  p.print();
40
41  return 0;
42 }
```

Wir erhalten damit in der Konsole den folgende Output:

```
$ g++ exercise_07_34.cpp polynomial.cpp matrix.cpp -o exercise_07_34
$ ./exercise_07_34
4.83486x^3 - 1.47748x
```

Exercise 7.35

Wir nutzen erneut die bereits `Matrix`-Klasse.

Es sei $n = 5$ die Anzahl der Punkte und $h = 1/(n-1) = 1/4$ der Abstand der Punkte. Aus den gegebenen Randwerten erhalten wir für die Werte $u_i = u(ih)$, dass $u_0 = u(0) = 0$ und $u_n = u(1) = u(0)$. Aus der Approximation

$$u''(x) \approx \frac{1}{h^2}(u(x+h) + u(x-h) - 2u(x))$$

und der Bedingung $u'' = 4u$ die Gleichungen

$$4u_i = \frac{1}{h^2}(u_{i+1} + u_{i-1} - 2u) \quad \text{für alle } i = 1, \dots, n-1,$$

also das lineare Gleichungssystem

$$\begin{pmatrix} C & -1 & & & \\ -1 & C & \ddots & & \\ & -1 & \ddots & -1 & \\ & & \ddots & C & -1 \\ & & & -1 & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} u_0 \\ 0 \\ \vdots \\ 0 \\ su_n \end{pmatrix},$$

wobei $C = 2 + 4h^2$.

```

1 #include "matrix.hpp"
2 #include <cmath>
3 #include <vector>
4 #include <iostream>
5 #include <iomanip>
6
7
8
9 double f_exact(double x){ return 2*sinh(2*x); } // the exact solution
10
11 int main(){
12     int n = 5;                      // number of points
13     double left = 0;                // left border
14     double right = 1;               // right border
15     double lb = 0;                 // left boundary value
16     double rb = exp(2) - exp(-2); // right boundary value
17
18     std::vector<double> u(n,0);    // function values
19     u[0] = lb;
20     u[n-1] = rb;
21
22     int m = n-2;                  // number of points in the middle
23     double h = (right - left)/(n-1); // length of intervals
24     double C = 2 + 4*h*h;          // constant for linear system
25
26     Matrix A = Matrix(m, m, 0);    // constructing the band matrix
27     for(int i = 0; i < m; i++){
28         A(i,i) = C;
29         for(int i = 0; i < m-1; i++){
30             A(i,i+1) = -1;
31             A(i+1,i) = -1;
32         }
33         std::vector<double> b(m,0);    // constructing the solution vector
34         b[0] = u[0];
35         b[m-1] = u[n-1];
36
37         std::vector<double> u_inner = gaussSolve(A,b);
38         for(int i = 0; i < m; i++){

```

```

39     u[i+1] = u_inner[i];
40 }
41
42 // output
43 std::cout << std::fixed;
44 std::cout << std::setprecision(8);
45 std::cout << "x\t\t approximation\t exact" << std::endl;
46 double x = 0; // current position
47 for(int i = 0; i < n; i++){
48     std::cout << x
49             << "\t| "
50             << u[i]
51             << "\t| "
52             << f_exact(x)
53             << std::endl;
54     x += h;
55 }
56 return 0;
57 }
```

Wir erhalten in der Konsole den folgende Output:

x	approximation	exact
0.00000000	0.00000000	0.00000000
0.25000000	1.05269418	1.04219061
0.50000000	2.36856190	2.35040239
0.75000000	4.27657010	4.25855891
1.00000000	7.25372082	7.25372082

Exercise 7.36

Es sei $n = 5$ die Anzahl der Punkte und $h = 1/(n - 1) = 1/4$ der Abstand der n im Intervall $[0, 1]$ gleichmäßig verteilten Punkte $x_i = ih$. Aus der Randbedingung $u(0) = 0$ erhalten wir für die Werte $u_i = u(x_i)$, dass $u_0 = u(x_0) = u(0) = 0$. Mit den Annäherungen

$$u''(x) \approx \frac{1}{h^2}(u(x+h) + u(x-h) - 2u(x))$$

und

$$u'(x) \approx \frac{1}{h}(u(x+h) - u(x))$$

erhalten wir aus der Differenzialgleichung $u''(x) - 5u'(x) + 4u(x) = x^2$ für alle $i \geq 1$ die linearen Gleichungen

$$\frac{1}{h^2}(u_{i+1} + u_{i-1} - 2u_i) - \frac{5}{h}(u_{i+1} - u_i) + 4u_i = x_i^2. \quad (7.1)$$

Um die Bedingung $u'(1) = 0$ zu simulieren, fügen wir noch einen zusätzlichen Punkt $x_n = 1 + h$ hinzu, und für den entsprechenden Funktionswert $u_n = u(1 + h)$ die

Gleichung

$$\frac{1}{h}(u_n - u_{n-1}) = u'(1) = 0. \quad (7.2)$$

Wir erhalten somit ein lineares Gleichungssystem in den n Unbekannten u_1, \dots, u_n , sowie $n - 1$ Gleichungen der Form (7.1) (für $i = 1, \dots, n - 1$) und der zusätzlichen Gleichung (7.2). Um dieses zu Lösen, nutzen wir erneut unsere Matrix-Klasse.

Wir erhalten somit das folgende Programm:

```

1 #include "matrix.hpp"
2 #include <cmath>
3 #include <vector>
4 #include <iostream>
5 #include <iomanip>
6
7 double f_exact(double x){
8     return -0.659305*exp(x) + 0.0030549*exp(4*x) + x*x/4 + 5*x/8 + 21/32.0;
9 }
10
11 int main(){
12     int n = 5;
13
14     double left = 0;      // left interval value
15     double right = 1;    // right interval value
16     double lb = 0;       // left boundary condition for u
17     double rb = 0;       // right boundary condition for u'
18
19     double h = (right - left)/(n-1); // distance between points
20     std::vector<double> u(n+1,0);   // additional point u[n] to the right of
21         // the interval
22     u[0] = lb;
23
24     Matrix A = Matrix(n,n);           // LGS for the values u[1] to u[n]
25     std::vector<double> b(n,0);
26
27     // set matrix entries
28     A(0,0) = -2/(h*h) + 5/h + 4;
29     A(0,1) = 1/(h*h) - 5/h;
30     for(int i = 1; i < n; i++){
31         A(i,i-1) = 1/(h*h);
32         A(i,i)   = -2/(h*h) + 5/h + 4;
33         A(i,i+1) = 1/(h*h) - 5/h;
34     }
35     A(n-1, n-2) = -1;
36     A(n-1, n-1) = 1;
37
38     // set result vector
39     double x = 0;
40     for(int i = 0; i < n-1; i++){
41         x += h;
42         b[i] = x*x;
43     }
44     b[0] += -u[0]/(h*h);
45     b[n-1] = h*rb;
46
47     std::vector<double> u_right = gaussSolve(A,b);

```

```

47|     for(int i = 0; i < n; i++)
48|         u[i+1] = u_right[i];
49|
50|     // printing results
51|     std::cout << std::fixed;
52|     std::cout << std::setprecision(6);
53|     x = 0; // current point, variable already initialized before
54|     std::cout << "x\t\t approx\t\t exact" << std::endl;
55|     for(int i = 0; i < n; i++){
56|         std::cout << x
57|             << "\t| "
58|             << u[i]
59|             << "\t| "
60|             << f_exact(x)
61|             << std::endl;
62|         x += h;
63|     }

```

Wir erhalten den folgenden Output:

x	approx	exact
0.000000	0.000000	-0.000000
0.250000	-0.000451	-0.010135
0.500000	-0.014724	-0.033187
0.750000	-0.034856	-0.068764
1.000000	-0.129808	-0.094135

Unsere Annäherung ist nicht sonderlich exakt. Dies lässt sich etwa damit erklären, dass wir in unserem Vorgehen die Bedingung $u'(1) = 0$ dadurch implementieren, dass $u_n = u_{n-1}$, also $u(1) = u(1+h) = u(5/4)$. Die exakte Lösung verändert sich im Bereich $[1, 5/4]$ allerdings verhältnismäßig stark, weshalb diese Umsetzung der Randbedingung $u'(1) = 0$ nicht sonderlich gut funktioniert.

Dieses Problem lässt nach, wenn mehr Punkte verwendet werden. So enthält man etwa für $n = 24$ die folgenden Ergebnisse:

x	approx	exact
0.000000	0.000000	-0.000000
0.043478	-0.000891	-0.001071
0.086957	-0.002016	-0.002388
0.130435	-0.003416	-0.003989
0.173913	-0.005131	-0.005909
0.217391	-0.007200	-0.008181
0.260870	-0.009659	-0.010837
0.304348	-0.012544	-0.013902
0.347826	-0.015886	-0.017399
0.391304	-0.019710	-0.021343
0.434783	-0.024036	-0.025743
0.478261	-0.028874	-0.030595
0.521739	-0.034226	-0.035886
0.565217	-0.040075	-0.041586
0.608696	-0.046391	-0.047645
0.652174	-0.053117	-0.053993
0.695652	-0.060172	-0.060531
0.739130	-0.067436	-0.067126
0.782609	-0.074747	-0.073605

0.826087	-0.081886	-0.079745
0.869565	-0.088570	-0.085265
0.913043	-0.094428	-0.089813
0.956522	-0.098987	-0.092949
1.000000	-0.101646	-0.094135

Exercise 7.37

Die Klasse `Polynomial` hat die folgende Header-Datei:

```

1 #include <vector>
2
3 class Polynomial{
4     std::vector<double> coeff;
5
6 public:
7     Polynomial();
8     Polynomial(std::vector<double> coefficients);
9     Polynomial(double c);
10
11    Polynomial operator-();
12    Polynomial operator+(Polynomial p);
13    Polynomial operator-(Polynomial p);
14    Polynomial operator*(Polynomial p);
15
16    double operator()(double x);
17
18    Polynomial clean();
19    bool operator==(Polynomial p);
20    void print();
21 };

```

Der hinterliegende Code ist wie folgt:

```

1 #include "polynomial.hpp"
2 #include <cmath>
3 #include <vector>
4 #include <iostream>
5 #include <string>
6
7 #define EPS 1.E-14
8
9
10
11 Polynomial::Polynomial(std::vector<double> coefficients){
12     coeff = coefficients;
13 }
14
15 Polynomial::Polynomial(double c){
16     coeff = {c};
17 }
18
19 Polynomial Polynomial::operator*(Polynomial p){
20     int d1 = coeff.size();

```

```

21 |     int d2 = p.coeff.size();
22 |     std::vector<double> result(d1+d2-1, 0);
23 |     for(int i = 0 ; i < d1; i++){
24 |         for(int j = 0; j < d2; j++){
25 |             result[i+j] += coeff[i]*p.coeff[j];
26 |         }
27 |     }
28 |     return Polynomial(result);
29 | }
30 |
31 Polynomial Polynomial::operator+(Polynomial p){
32     int d1 = coeff.size();
33     int d2 = p.coeff.size();
34     std::vector<double> result(std::max(d1,d2), 0);
35     for(int i = 0; i < d1; i++)
36         result[i] += coeff[i];
37     for(int j = 0; j < d2; j++)
38         result[j] += p.coeff[j];
39     return Polynomial(result);
40 }
41 |
42 Polynomial Polynomial::operator-(Polynomial p){
43     int d1 = coeff.size();
44     int d2 = p.coeff.size();
45     std::vector<double> result(std::max(d1,d2), 0);
46     for(int i = 0; i < d1; i++)
47         result[i] += coeff[i];
48     for(int j = 0; j < d2; j++)
49         result[j] -= p.coeff[j];
50     return Polynomial(result);
51 }
52 |
53 Polynomial Polynomial::operator-(){
54     int d = coeff.size();
55     std::vector<double> result;
56     for(int i = 0; i < d; i++)
57         result.push_back(-coeff[i]);
58     return Polynomial(result);
59 }
60 |
61 double Polynomial::operator()(double x){
62     double result = 0;
63     int d = coeff.size();
64     for(int i = 0; i < d; i++)
65         result += coeff[i]*pow(x,i);
66     return result;
67 }
68 |
69 Polynomial Polynomial::clean(){
70     int d = coeff.size();
71     std::vector<double> result(d,0);
72     for(int i = 0; i < d; i++)
73         if(std::abs(coeff[i]) > EPS)
74             result[i] = coeff[i];
75     return Polynomial(result);
76 }

```

```

77 | bool Polynomial::operator==(Polynomial p){
78 |     Polynomial diff = (*this) - p;
79 |     diff = diff.clean();
80 |     int d = diff.coeff.size();
81 |     for(int i = 0; i < d; i++)
82 |         if(diff.coeff[i] != 0)
83 |             return false;
84 |     return true;
85 | }
86 |
87 |
88 | void Polynomial::print(){
89 |     Polynomial p = (*this).clean();
90 |
91 |     if(p == Polynomial(0))
92 |         std::cout << "0";
93 |     else{
94 |         int d = p.coeff.size();
95 |         bool first = true;           // if other coefficients have already been
96 |                                         // printed
97 |         double c;                 // current coefficient
98 |         std::string prefix, power; // prefix and power for the summands
99 |         for(int i = d-1; i >= 0; i--){
100 |             c = p.coeff[i];
101 |             if(c == 0)
102 |                 continue;
103 |             prefix, power = "";
104 |
105 |             if (c > 0){
106 |                 if (!first)
107 |                     prefix = " + ";
108 |             else if (c < 0){
109 |                 if (!first)
110 |                     prefix = " - ";
111 |                 else
112 |                     prefix = "-";
113 |             }
114 |
115 |             if (i >= 2)
116 |                 power = "x^" + std::to_string(i);
117 |             else if(i == 1)
118 |                 power = "x";
119 |
120 |             std::cout << prefix << (c > 0 ? c : -c) << power;
121 |             first = false;
122 |         }
123 |         std::cout << std::endl;
124 |     }
125 | }
```

Wir testen unsere `Polynomial`-Klasse mit dem folgenden Code:

```

1 #include "polynomial.hpp"
2 #include <iostream>
3
4
```

```

5
6 int main(){
7     Polynomial p({1, 2, 3}), q({0, 2, 1});
8     std::cout << "P: ";
9     p.print();
10    std::cout << "The value P(2): " << p(2) << std::endl;
11    std::cout << "-P: ";
12    (-p).print();
13    std::cout << "Q: ";
14    q.print();
15    std::cout << "P*Q: ";
16    (p*q).print();
17    std::cout << "P+Q: ";
18    (p+q).print();
19    std::cout << "Test P = Q: " << (p == q) << std::endl;
20    std::cout << "Test P = P: " << (p == p) << std::endl;
21 }

```

Wir erhalten in der Konsole den zu erwartenden Output:

```

$ g++ exercise_07_37.cpp polynomial.cpp -o exercise_07_37
$ ./exercise_07_37
P: 3x^2 + 2x + 1
The value P(2): 17
-P: -3x^2 - 2x - 1
Q: 1x^2 + 2x
P*Q: 3x^4 + 8x^3 + 5x^2 + 2x
P+Q: 4x^2 + 4x + 1
Test P = Q: 0
Test P = P: 1

```

Exercise 7.38

Unsere Quaternion-Klasse hat die folgende Header-Datei:

```

1 #include<vector>
2
3 class Quaternion {
4     std::vector<double> coord;
5
6 public:
7
8     Quaternion();
9     Quaternion(double x, double i, double j, double k);
10    Quaternion(double q[4]);
11    Quaternion(double x);
12
13    double abs();
14    Quaternion operator-();
15    Quaternion conjugate();
16    Quaternion inverse();
17    Quaternion clean();
18
19    Quaternion operator+(Quaternion q);

```

```

20| Quaternion operator-(Quaternion q);
21| Quaternion operator*(Quaternion q);
22| Quaternion operator/(double r);
23| Quaternion operator/(Quaternion q);
24|
25| bool operator==(Quaternion q);
26|
27| void print();
28}

```

Der konkrete Code ist wie folgt:

```

1 #include "quaternion.hpp"
2 #include <vector>
3 #include <cmath>
4 #include <iostream>
5
6
7 #define EPS 1.E-14
8
9
10 Quaternion::Quaternion(){
11     coord = {0,0,0,0};
12 }
13
14 Quaternion::Quaternion(double x, double i, double j, double k){
15     coord = {x,i,j,k};
16 }
17
18 Quaternion::Quaternion(double q[4]){
19     coord = {q[0], q[1], q[2], q[3]};
20 }
21
22 Quaternion::Quaternion(double x){
23     coord = {x, 0, 0, 0};
24 }
25
26 Quaternion Quaternion::operator+(Quaternion q){
27     double result[4];
28     for(int i = 0 ; i < 4; i++)
29         result[i] = coord[i] + q.coord[i];
30     return Quaternion(result);
31 }
32
33 Quaternion Quaternion::operator-(){
34     double result[4];
35     for(int i = 0; i < 4; i++)
36         result[i] = -coord[i];
37     return Quaternion(result);
38 }
39
40 Quaternion Quaternion::operator-(Quaternion q){
41     double result[4];
42     for(int i = 0; i < 4; i++)
43         result[i] = coord[i] - q.coord[i];
44     return Quaternion(result);
45 }

```

```

46|
47| Quaternion Quaternion::operator*(Quaternion q){
48|     double r, x, y, z;
49|     r = coord[0] * q.coord[0]
50|     - coord[1] * q.coord[1]
51|     - coord[2] * q.coord[2]
52|     - coord[3] * q.coord[3];
53|     x = coord[0] * q.coord[1]
54|     + coord[2] * q.coord[3]
55|     - coord[3] * q.coord[2]
56|     + coord[1] * q.coord[0];
57|     y = coord[0] * q.coord[2]
58|     - coord[1] * q.coord[3]
59|     + coord[3] * q.coord[1]
60|     + coord[2] * q.coord[0];
61|     z = coord[0] * q.coord[3]
62|     + coord[1] * q.coord[2]
63|     - coord[2] * q.coord[1]
64|     + coord[3] * q.coord[0];
65|     return Quaternion({r, x, y, z});
66}
67|
68| double Quaternion::abs(){
69|     double squared = coord[0] * coord[0]
70|     + coord[1] * coord[1]
71|     + coord[2] * coord[2]
72|     + coord[3] * coord[3];
73|     return sqrt(squared);
74}
75|
76| Quaternion Quaternion::conjugate(){
77|     double result[] = {coord[0], -coord[1], -coord[2], -coord[3]};
78|     return Quaternion(result);
79}
80|
81| Quaternion Quaternion::operator/(double r){
82|     double result[4];
83|     for(int i = 0; i < 4; i++)
84|         result[i] = coord[i]/r;
85|     return Quaternion(result);
86}
87|
88| Quaternion Quaternion::inverse(){
89|     double norm = (*this).abs() * (*this).abs();
90|     return (*this).conjugate()/norm;
91}
92|
93| Quaternion Quaternion::operator/(Quaternion q){
94|     return (*this) * q.inverse();
95}
96|
97| Quaternion Quaternion::clean(){
98|     double result[] = {0,0,0,0};
99|     for(int i = 0; i < 4; i++)
100|         if(std::abs(coord[i]) > EPS)
101|             result[i] = coord[i];

```

```

102    return Quaternion(result);
103 }
104
105 bool Quaternion::operator==(Quaternion q){
106     Quaternion diff = (*this) - q;
107     diff = diff.clean();
108     for(int i = 0; i < 4; i++){
109         if(diff.coord[i] != 0)
110             return false;
111     }
112     return true;
113 }
114
115 void Quaternion::print(){
116     if(*this == Quaternion())
117         std::cout << "0" << std::endl;
118     else{
119         bool first = true;
120         char symbol[] = {'\0', 'i', 'j', 'k'};
121         std::string prefix; // current prefix
122         double c;           // current coordinate
123         for(int i = 0; i < 4; i++){
124             c = coord[i];
125             if(c == 0)
126                 continue;
127             prefix = "";
128
129             if(c > 0){
130                 if(!first)
131                     prefix = " + ";
132             }
133             else{
134                 if(first)
135                     prefix = "-";
136                 else
137                     prefix = " - ";
138             }
139             std::cout << prefix << (c > 0 ? c : -c) << symbol[i];
140             first = false;
141         }
142         std::cout << std::endl;
143     }
}

```

Wir testen unsere Quaternion-Klasse mit dem folgende Programm:

```

1 #include "quaternion.hpp"
2 #include<iostream>
3
4
5 int main(){
6     Quaternion q1(1,2,3,4), q2(2,3,4,5);
7     std::cout << "Q1: ";
8     q1.print();
9     std::cout << "Negative of Q1: ";
10    (-q1).print();
11    std::cout << "Conjugate of Q1: ";
12    q1.conjugate().print();
}

```

```

13 std::cout << "Absolute value of Q1: " << q1.abs() << std::endl;
14 std::cout << "Q2: ";
15 q2.print();
16 std::cout << "Test for Q1 = Q2: " << (q1 == q2) << std::endl;
17 std::cout << "Test for Q1 = Q1: " << (q1 == q1) << std::endl;
18 std::cout << "Q1 * Q2: ";
19 (q1 * q2).print();
20 std::cout << "Q2 * Q1: ";
21 (q2 * q1).print();
22 std::cout << "Q1 / Q2: ";
23 (q1 / q2).print();
24 std::cout << "(Q1 / Q2) * Q2: ";
25 ((q1/q2)*q2).print();
26 std::cout << "Hamilton equations:" << std::endl;
27 Quaternion i = Quaternion({0,1,0,0});
28 Quaternion j = Quaternion({0,0,1,0});
29 Quaternion k = Quaternion({0,0,0,1});
30 std::cout << "i^2 = ";
31 (i*i).print();
32 std::cout << "j^2 = ";
33 (j*j).print();
34 std::cout << "k^2 = ";
35 (k*k).print();
36 std::cout << "ijk = ";
37 (i*j*k).print();
38
39 return 0;
40 }

```

In der Konsole erhalten wir den zu erwartenden Output:

```

$ g++ exercise_07_38.cpp quaternion.cpp -o exercise_07_38
$ ./exercise_07_38
Q1: 1 + 2i + 3j + 4k
Negative of Q1: -1 - 2i - 3j - 4k
Conjugate of Q1: 1 - 2i - 3j - 4k
Absolute value of Q1: 5.47723
Q2: 2 + 3i + 4j + 5k
Test for Q1 = Q2: 0
Test for Q1 = Q1: 1
Q1 * Q2: -36 + 6i + 12j + 12k
Q2 * Q1: -36 + 8i + 8j + 14k
Q1 / Q2: 0.740741 + 0.037037i + 0.0740741k
(Q1 / Q2) * Q2: 1 + 2i + 3j + 4k
Hamilton equations:
i^2 = -1
j^2 = -1
k^2 = -1
ijk = -1

```

8 Matlab

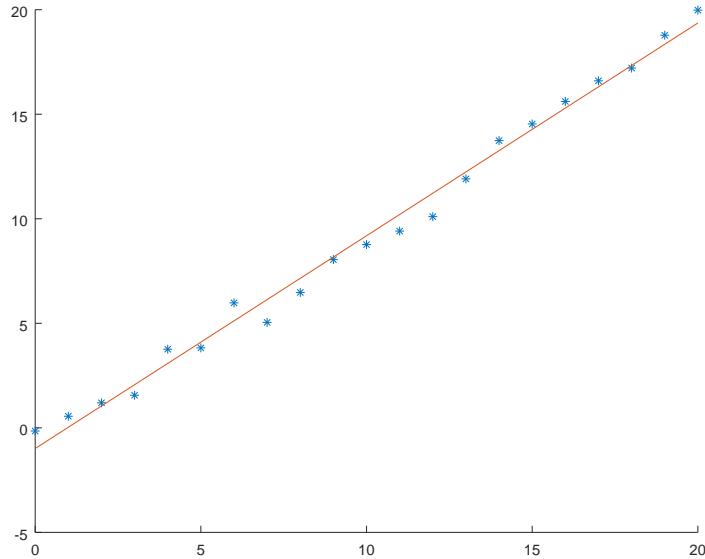
Mit Ausnahme von Aufgabe 41 sind alle Aufgaben in Octave geschrieben.

Exercise 8.39

Wir nutzen den folgenden Code:

```
1 n = 20;      % number of points
2 x = 0 : n;    % x values
3 y = x + 2*(rand(size(x)) - 1); % y values
4
5 A = zeros(length(x), 2);    % initialize zero matrix
6 A(:,1) = x';               % insert x values in first column
7 A(:,2) = ones(length(y),1); % insert ones into second column
8
9 a = (A\y)';   % find coefficients of regression line
10
11 hold on;          % to plot both points and the line
12 plot(x, y, '*'); % plotting the points
13 plot(x, a(1)*x + a(2)); % plotting the line
```

Wir erhalten damit die beispielsweise die folgende Regressionsgerade:

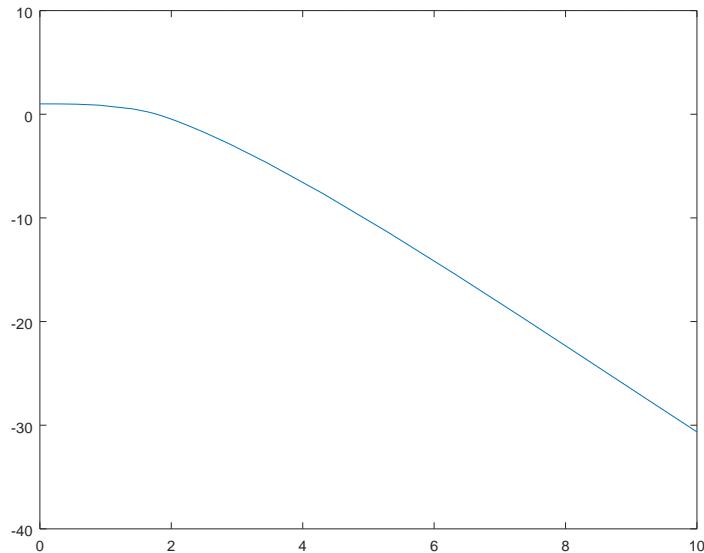


Exercise 8.40

Wir nutzen den folgenden Code:

```
1 x_interval = [0 10]; % interval on which we work
2 dydx = @(x,y) [y(2) y(3) -1/y(1)]';
3 y0 = [1 0 0];           % initial values
4
5 [x,y] = ode45(dydx, x_interval, y0);
6 u = y(:,1);
7
8 plot(x,u);
```

Wir erhalten damit den folgenden Graphen:

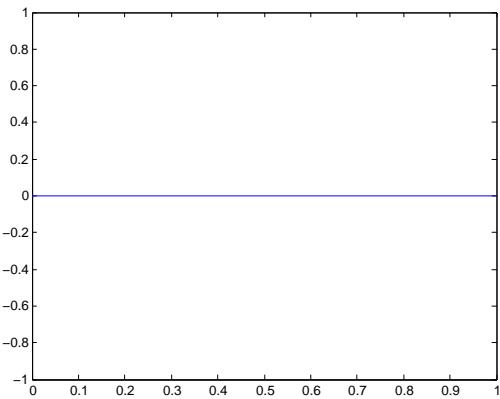


Exercise 8.41

Wir nutzen den folgenden Code:

```
1 dydx = @(x,y) [y(2), 2 * y(2) - y(1)]; % the differential equation
2 bvs = @(lb, rb)[lb(1), rb(1)]';          % boundary conditions
3 sol_init = bvpinit([0 1], [0 0]);           % assumptions to work with
4 sol = bvp4c(dydx, bvs, sol_init);          % solve the bvp
5
6 % plotting the solution
7 x = linspace(0,1);
8 w = deval(sol, x, 1);
9 u = w(1,:);
10 plot(x,u); % the zero function
```

Wir erhalten damit als Lösung die konstante Nullfunktion.



Exercise 8.42

Wir schreiben die Funktion `poisson_solver` aus Abschnitt 4.8 nach Matlab (bzw. Octave) um:

```

1 function poisson_solver(f,g,m)
2
3 n = m-1;
4 h = 1/(n+1);
5
6 A = gallery('poisson',n);
7
8 % construction of grid
9 lin = linspace(0,1,n+2);
10 [x,y] = meshgrid(lin);
11
12 % initialize solution matrix with 0s
13 u = zeros(n+2, n+2);
14
15 % insert boundary values
16 u(:,1) = arrayfun(g, x(:,1), y(:,1));
17 u(:,n+2) = arrayfun(g, x(:,n+2), y(:,n+2));
18 u(1,:) = arrayfun(g, x(1,:), y(1,:));
19 u(n+2,:) = arrayfun(g, x(n+2,:), y(n+2,:));
20
21 % initialize solution matrix with 0s
22 F = zeros(n, n);
23
24 % insert function values f_ij
25 F = arrayfun(f, x(2:n+1,2:n+1), y(2:n+1,2:n+1));
26
27 % modify boundary-adjacent values
28 F(:,1) += ( u(2 : n+1, 1) / h^2 ); % left column
29 F(:,n) += ( u(2 : n+1, n+2) / h^2 ); % right column
30 F(1,:) += ( u(1, 2 : n+1) / h^2 ); % top row
31 F(n,:) += ( u(n+2, 2 : n+1) / h^2 ); % bottom row
32
33 % as column-vector

```

```

34 F = reshape(F, n*n, 1);
35 % solve the inner points
36 u_inner = A \ (h*h*F);
37
38 % store values in solution-matrix
39 u(2:n+1, 2:n+1) = reshape(u_inner,n,n);
40
41 % plotting the solution
42 mesh(x,y,u, 'LineWidth', 1);
43 xlabel('x');
44 ylabel('y');
45
46 end

```

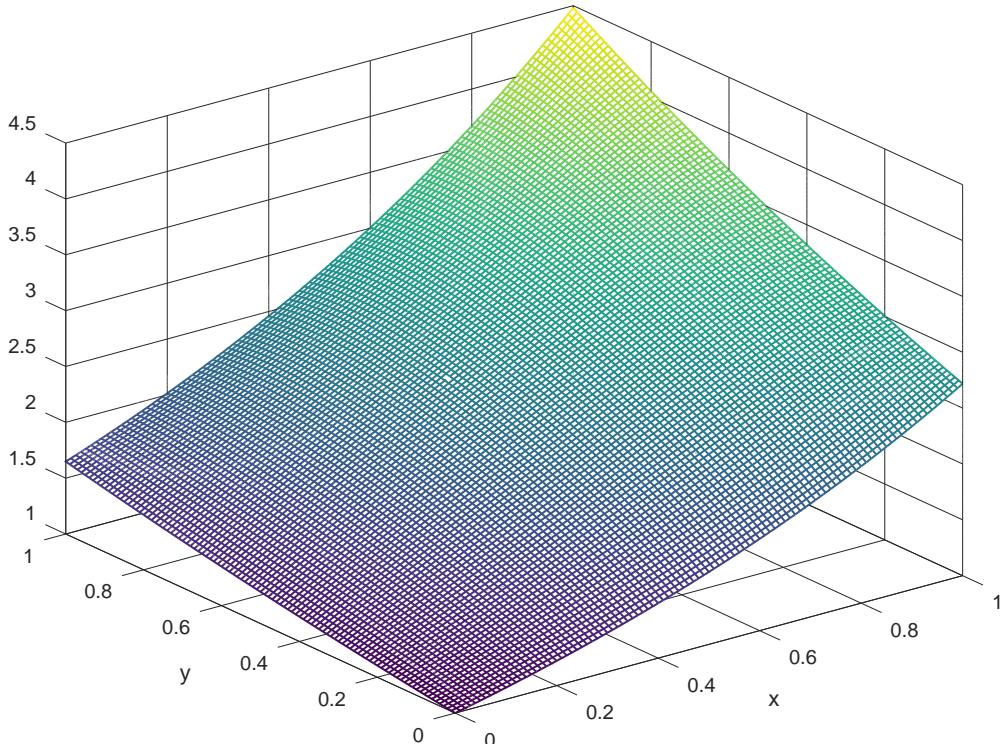
Wir testen die Funktion `poisson_solver` zunächst an den in (1) gegebenen Funktionen $f(x, y) = 1.25 \cdot e^{x+y/2}$, $g = e^{x+y/2}$ mithilfe des folgenden Codes:

```

1 f = @(x,y) 1.25 * exp(x + y/2);
2 g = @(x,y) exp(x + y/2);
3
4 poisson_solver(f, g, 100);

```

Wir erhalten hierdurch den folgenden Graphen:



(Die auffällige weiße Lücke in der Mitte des Graphen entsteht beim Exportieren der Grafik aus Octave, und ist im Graphen selbst nicht vorhanden.)

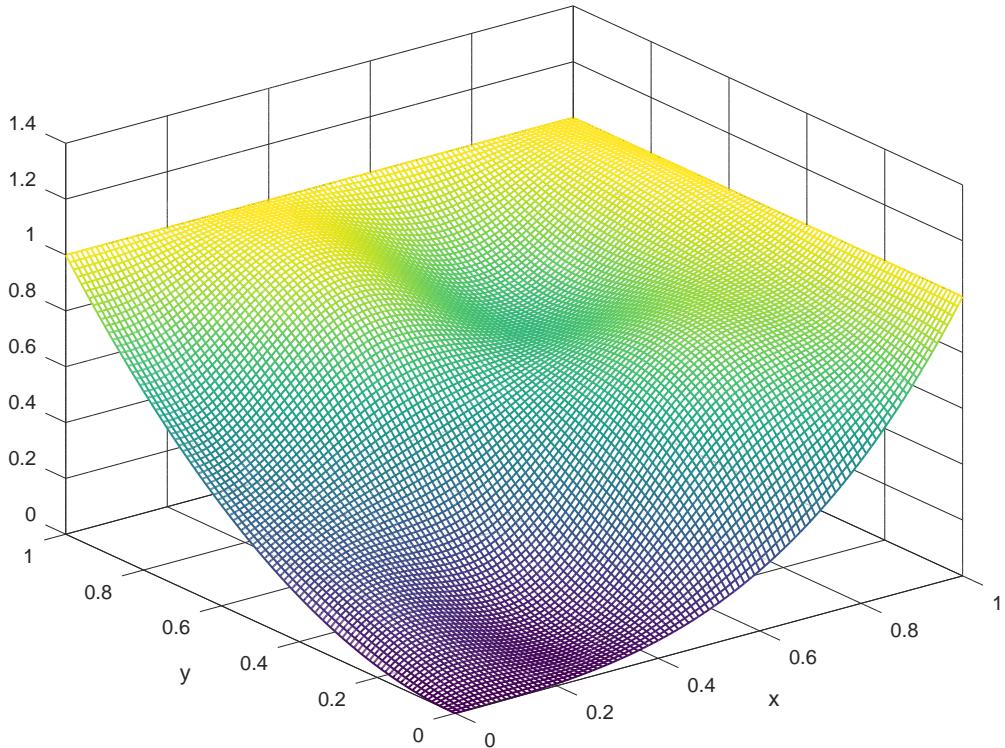
Wir testen die Funktion `poisson_solver` auch noch an in (2) gegebenen Funktionen. Hierfür nutzen wir den folgenden Code:

```

1 f = @(x,y) 20 * cos(3*pi*x) * sin(2*pi*y);
2 function z = g(x,y)
3   z = 0;
4   if x == 0
5     z = y.^2;
6   elseif y == 0
7     z = x.^3;
8   elseif (y == 1) || (x == 1)
9     z = 1;
10  end
11 end
12
13 poisson_solver(f, @g, 100);

```

Wir erhalten hierdurch den folgenden Graphen:



Exercise 8.43

Wir modifizieren die Funktion `poisson_solver` aus der vorherigen Aufgabe dahingehend, dass wir ein allgemeines Lösen über rechteckigen Gebieten $[x_1, x_2] \times [y_1, y_2]$ erlauben:

```

1 function poisson_solver(f,g,m)
2
3     n = m-1;
4     h = 1/(n+1);
5
6     A = gallery('poisson',n);
7
8 % construction of grid
9     lin = linspace(0,1,n+2);
10    [x,y] = meshgrid(lin);
11
12 % initialize solution matrix with 0s
13    u = zeros(n+2, n+2);
14
15 % insert boundary values
16    u(:,1) = arrayfun(g, x(:,1), y(:,1));
17    u(:,n+2) = arrayfun(g, x(:,n+2), y(:,n+2));
18    u(1,:) = arrayfun(g, x(1,:), y(1,:));
19    u(n+2,:) = arrayfun(g, x(n+2,:), y(n+2,:));
20
21 % initialize solution matrix with 0s
22    F = zeros(n, n);
23
24 % insert function values f_ij
25    F = arrayfun(f, x(2:n+1,2:n+1), y(2:n+1,2:n+1));
26
27 % modify boundary-adjacent values
28    F(:,1) += ( u(2 : n+1, 1) / h^2 ); % left column
29    F(:,n) += ( u(2 : n+1, n+2) / h^2 ); % right column
30    F(1,:) += ( u(1, 2 : n+1) / h^2 ); % top row
31    F(n,:) += ( u(n+2, 2 : n+1) / h^2 ); % bottom row
32
33 % as column-vector
34    F = reshape(F, n*n, 1);
35
36 % solve the inner points
37    u_inner = A \ (h*h*F);
38
39 % store values in solution-matrix
40    u(2:n+1, 2:n+1) = reshape(u_inner,n,n);
41
42 % plotting the solution
43    mesh(x,y,u, 'LineWidth', 1);
44    xlabel('x');
45    ylabel('y');
46 end

```

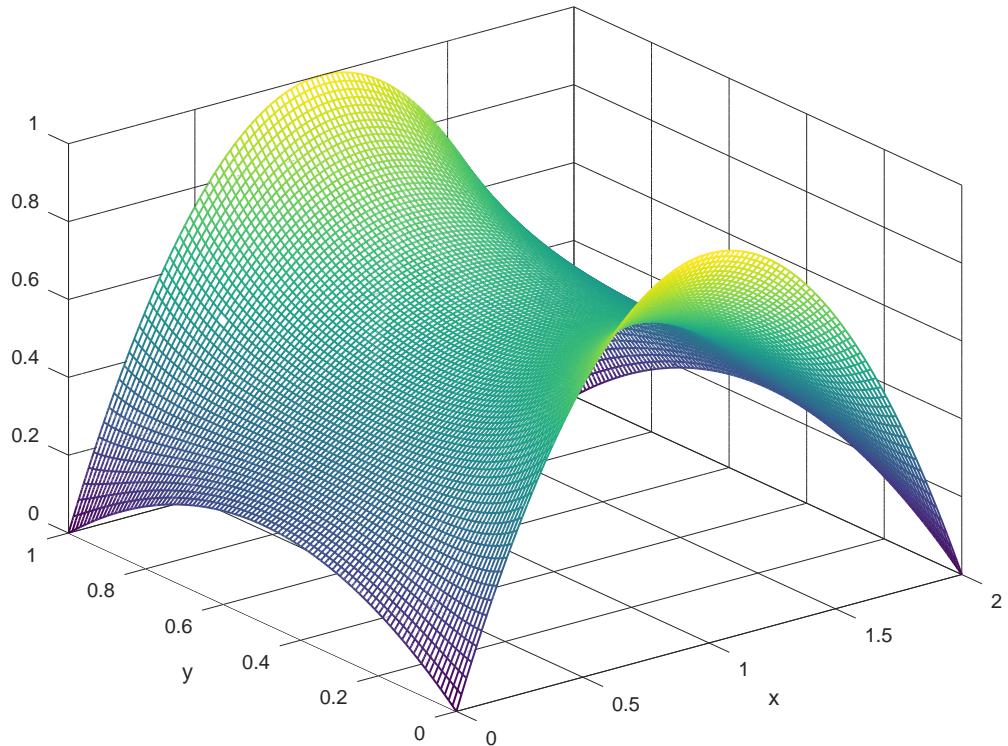
Wir testen diese erweiterte Version der Funktion `poisson_solver` anhand der gegebenen Funktionen $f(x,y) = x^2 + y^2$ und $g(x,y) = x(2-x) + y(1-y)$ mithilfe des folgenden Codes:

```

1 f = @(x,y) x^2 + y^2;
2 g = @(x,y) x*(2-x) + y*(1-y);
3
4 poisson_solver(f, g, [0,2], [0,1], 100);

```

Wir erhalten als Ergebnis den folgenden Graphen:



Exercise 8.44

Mithilfe des in der Aufgabenstellung geschilderten Vorgehens erhalten wir den folgenden Code:

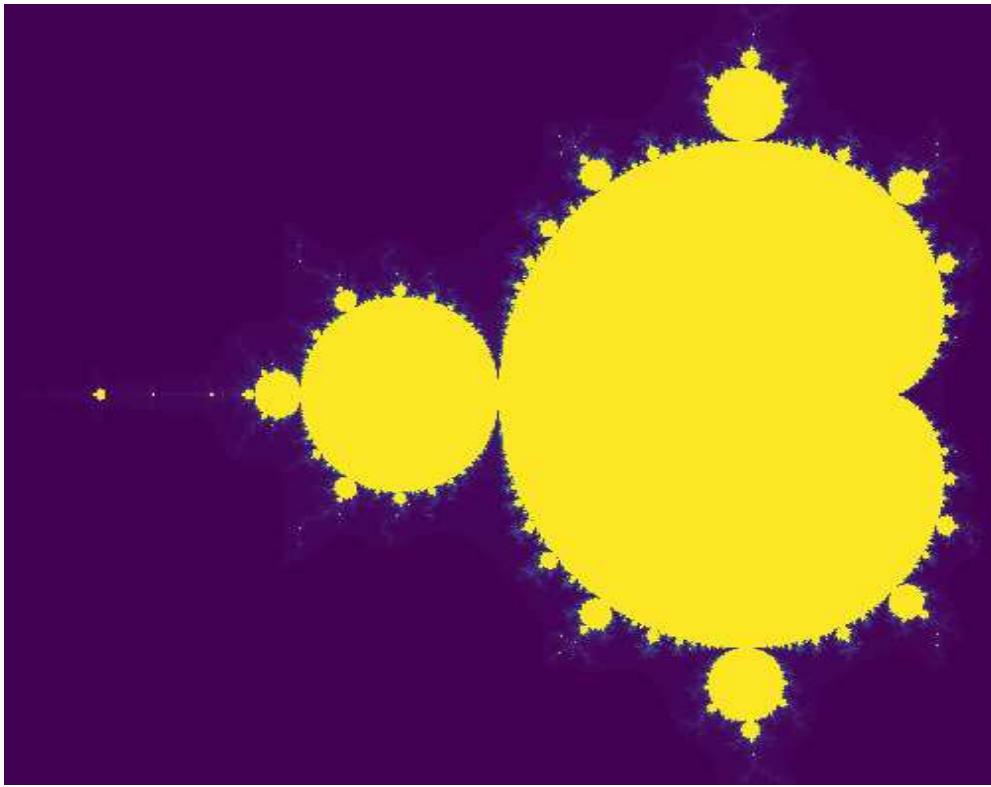
```
1 % size = 2000;
2 % iter = 2000;
3 size = 500;
4 iter = 400;
5 cutoff = 2;
6
7 xlimit = [-2, 0.5];
8 ylimit = [1, -1];
9 % xlimit = [-0.7, -0.8];
10 % ylimit = [0.1, 0.2];
11
12 x = linspace(xlimit(1), xlimit(2), size);
13 y = linspace(ylimit(1), ylimit(2), size);
14
15 [xx, yy] = meshgrid(x, y);
16 cc = xx + i*yy;
17
```

```

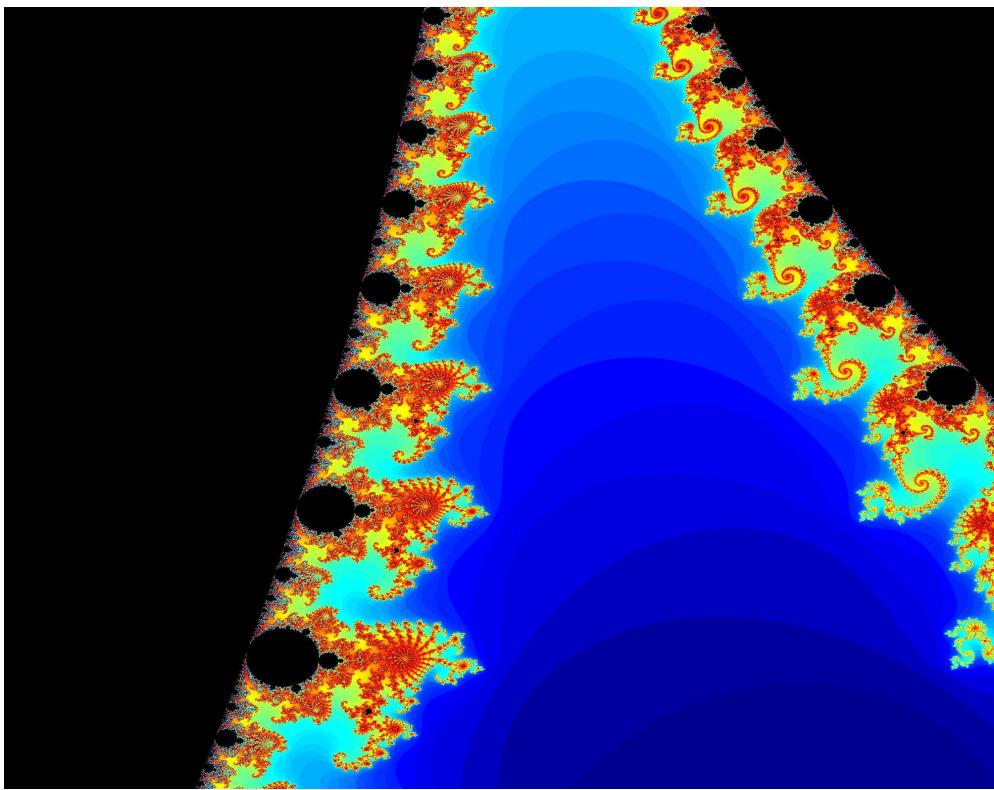
18 zz = zeros(length(y), length(x));
19 nn = zeros(length(y), length(x));
20
21 for j = 1 : iter
22     zz = zz.^2 + cc;
23     nn += (zz < 2);
24 end
25
26 % nn = log(nn);
27
28 % colormap( [jet();flipud( jet() );0 0 0] );
29 imagesc(abs(nn));
30 axis off;

```

Innerhalb weniger Sekunden erhalten wir hieraus das folgende Bild der Mandelbrotmenge:



Durch Ändern der Parameter (man betrachte die im Quellcode auskommentierten Zeilen) und mehr Rendering-Zeit lässt sich die Mandelbrotmenge nun in verschiedenen Bereichen mit unterschiedlichen Auflösungen und Farbverläufen betrachten:



9 Maple (done in SymPy)

Wir haben die Aufgaben im diesem Kapitel, soweit möglich, mit SymPy gearbeitet, da wir keinen Zugriff auf eine Maple-Installation hatten.

Exercise 9.45

Da wir diese Aufgabe in SymPy bereits in Aufgabe 30 von Kapitel 5 gelöst haben, geben wir den Code hier nicht noch einmal an.

Exercise 9.46

Da es bei dieser Aufgabe um den spezifischen Maple-Output geht, konnten wir diese Aufgabe nicht in SymPy bearbeiten.

Exercise 9.47

(1)

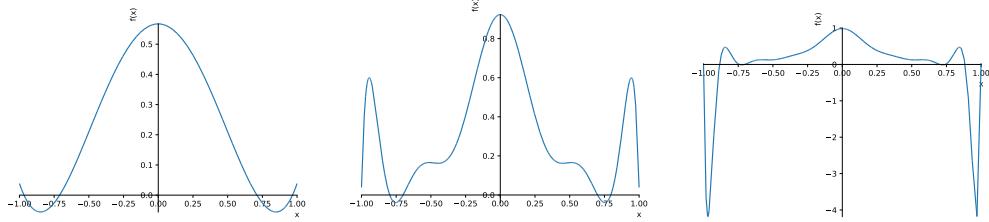
Mithilfe der `interpolate`-Methode, die in SymPy bereits zur Verfügung steht, erhalten wir den folgenden Code zum Interpolieren mit n auf $[-1, 1]$ gleichmäßig verteilten Punkten:

```
1 from sympy import *
2 x = symbols('x')
3 f_expr = 1/(1 + 25*x**2)
4
5 def rungepolate(n):
6     f = (lambda z: f_expr.subs('x',z))
7
8     xval = [ -1 + i*sympify(2)/n for i in range(n+1) ]
9     yval = [f(z) for z in xval]
10    points = list(zip(xval,yval))
11    poly = interpolate(points, x) # sympy interpolate
12
13    return poly
14
```

Wir testen unsere Funktion für die Werte $n = 5, 11, 17$ mit dem folgenden Code:

```
1 for n in [5, 11, 17]:
2     p = rungepolate(n)
3     plot(p, (x, -1, 1))
```

Wir erhalten hierdurch die folgenden Graphen:



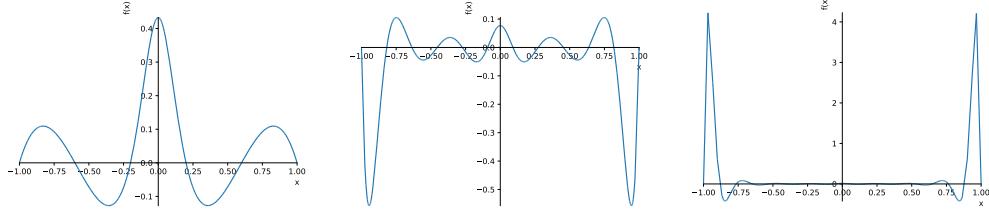
Es fällt auf, dass die Interpolationen zu den Intervallrändern hin zunehmend stark oszillieren.

(2)

Mithilfe des folgenden Codes betrachten wir den Unterschied $f(x) - P_n(x)$:

```
1 for n in [5, 11, 17]:
2     p = rungepolate(n)
3     plot(f_expr - p, (x, -1, 1))
```

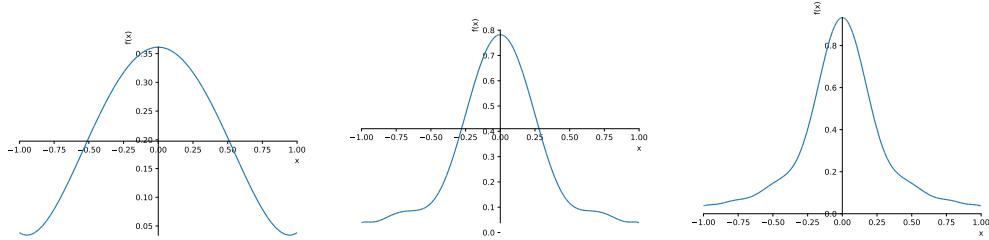
Unsere Vermutung scheint sich zu bestätigen:



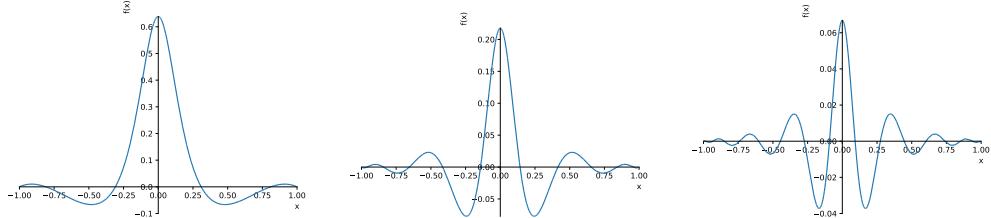
Exercise 9.48

Im vorherigen Programm muss nur der Ausdruck $-1 + i * \text{sympify}(2)/n$ durch den Ausdruck $-\cos(i * \pi/n).evalf()$ ersetzt werden. Dabei arbeiten wir hier aus Geschwindigkeitsgründen mit floats.

Hierdurch ergeben sich die folgenden Interpolationen:



Die Oszillationen scheinen für die Chebyshev-Knoten nicht aufzutreten. Dies ergibt sich auch bei Betrachtung des Fehlers:



Exercise 9.49

Da das Intervall $[-1, 1]$ symmetrisch um 0 ist, sind die Unterräume $U, G \subseteq C[-1, 1]$ der ungeraden und geraden Funktionen orthogonal bezüglich des L^2 -Skalarproduktes. Da die Polynome x^i alle gerade oder ungerade sind (abhängig von der Parität von i) sind somit auch alle Legendre-Polynome gerade oder ungerade. Da die Runge-Funktion gerade ist, genügt es für die Approximation dabei, die geraden Legendre-Polynome zu betrachten; dies sind gerade die Legendre-Polynome von geraden Grad (denn x^i ist genau dann gerade, wenn i gerade ist).

Wir nutzen nun den folgenden Code, um die Liste der Approximationen vom Grad $\leq n$ zu berechnen:

```

1 from sympy import *
2
3 x = symbols('x')
4 f = sympify("1/(1 + 25*x**2)")
5
6 def inner(f,g):
7     return integrate(f*g,(x,-1,1))
8
9 # copied from chapter 5, exercise 32
10 # calculates the even legendre polynomials of degree <= n
11 def legendreeven(n):
12     m = n//2 + 1 # number of polynomials to be computed
13     p = [x**(2*i) for i in range(m)] # to be orthogonalized
14     normsq = [] # list of squares of norms
15     for i in range(n//2 + 1):
16         s = 0 # projection onto previous polynomials
17         for j in range(i):
18             s += inner(p[i], p[j])/normsq[j] * p[j]
19         p[i] -= s
20         normsq.append( inner(p[i], p[i]) )
21     return p
22
23 # returns the first approximations of even degree <= n
24 def approx(n):
25     L = legendreeven(n)
26     approxlist = []
27     p = 0 #current approximation
28     for q in L:

```

```

29|         p += inner(f,q)/inner(q,q) * q
30|         approxlist.append(p)
31|     return approxlist

```

(1)

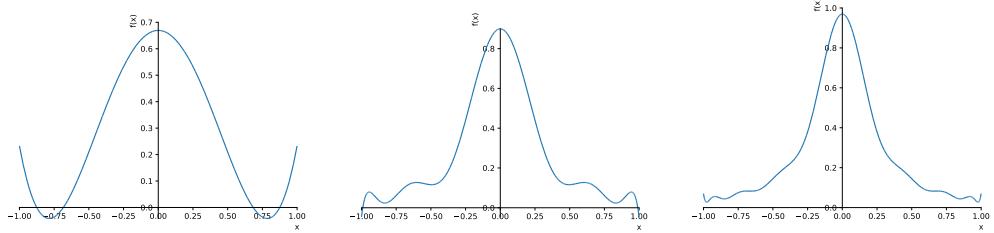
Wir testen die Approximationen für die bekannten Werte $n = 5, 11, 17$:

```

1 nlist = [5, 11, 17]
2 approxlist = approx(max(nlist))
3
4 # (a)
5 for n in nlist:
6     plot(approxlist[n//2], (x, -1, 1))
7
8 # (b)
9 for n in nlist:
10    plot(approxlist[n//2] - f, (x, -1, 1))

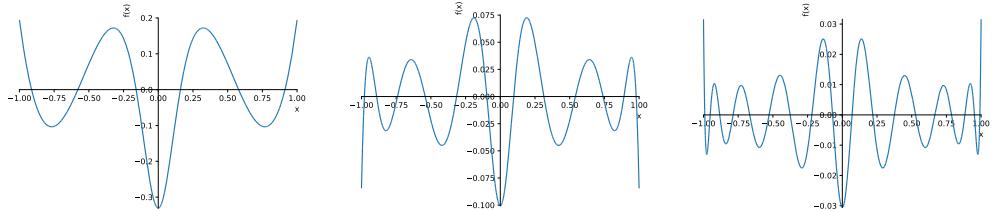
```

Hierdurch ergeben sich für die Approximationen die folgenden Graphen:



(2)

Die Fehler sind entsprechend klein:



Exercise 9.50

Wir können die gegebenen drei Punkte bereits durch ein quadratisches Polynom interpolieren:

```

1 from sympy import *
2 x = symbols('x');
4
5 # tries to interpolate the given points by a polynomial of degree deg
6 # only works because we have less pants then the degree
7
8 xval = [1,2,3]
9 yval = [1,4,3]
10 deg = 3
11
12 A = Matrix(3, deg+1, (lambda i,j : xval[i]**j)) # vandermonde matrix
13 result = Matrix(yval)
14 coeff_all = A.gauss_jordan_solve(result)
15 coeff = coeff_all[0] # parameters in the solution set
16 for var in list(coeff_all[1]): # set them all to 0
17     coeff = coeff.subs(var, 0) # to get a specific solution
18
19 p = 0
20 for i in range(deg+1):
21     p += coeff[i]*x**i
22
23 print(p)

```

Wir erhalten damit das folgende Polynom:

```
-2*x**2 + 9*x - 6
```

Da wir keinen Zugriff auf eine Maple-Installation haben, können wir nicht überprüfen, wie Maple die Splines wählt. Unsere Implementation minimiert den Grad der genutzten Polynome.

Exercise 9.51

Wir SciPy, da SymPy in der aktuellen Version 1.1.1 noch nicht über eine entsprechende Funktion verfügt.

```

1 from scipy import linspace
2 from scipy.interpolate import InterpolatedUnivariateSpline
3 import matplotlib.pyplot as plt
4
5 f = (lambda z: 1/(1 + 25*z**2))
6
7 # appoximation by splines with n equidistant points
8 def rungespline(n):
9     xvals = linspace(-1,1,n)
10    yvals = f(xvals)
11
12    spline = InterpolatedUnivariateSpline(xvals, yvals, k=3) # degree 3
13
14    lin = linspace(-1,1,100)
15    plt.clf()
16    plt.plot(lin, f(lin))

```

```
17|     plt.plot(lin, spline(lin))
```

Wir plotten die Approximation für $n + 1$ Punkte mit $n = 5, 7, 11$ mit dem folgenden Code:

```
1|  
2| for n in [5,7,11]:  
3|     rungespline(n)
```

Hierdurch ergeben sich die folgenden Graphen:

