Advanced Mathematics for Engineers - Laboratory Problems

Eric Saier (eric.saier@hs-weingarten.de)
Mat-Nr. 28224

January 26, 2017

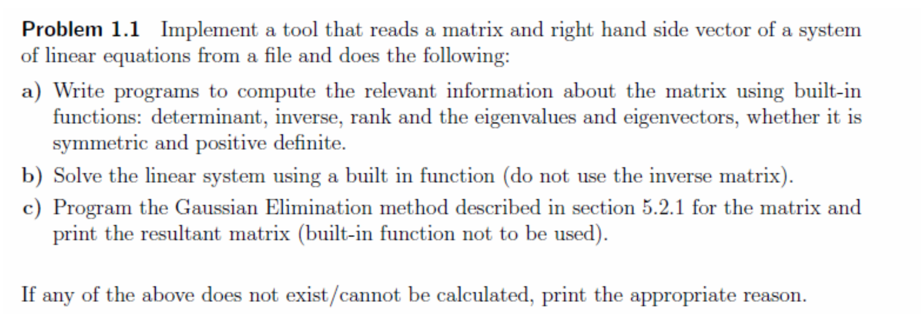
1 Linear Algebra

First, mylib to print mat

```python
def getNumberString(pNumber, strLength):
        nStr = str(round(pNumber, 2));

        if pNumber >= 0: nStr = " " + nStr;

        spacesBefore = strLength - len(nStr);

        if spacesBefore > 0:
                for i in range(spacesBefore): nStr = " " + nStr;

        return nStr;
#

def printVector(pVector):
    for element in pVector:
                print(getNumberString(element, 7));
#

def printMatrix(pMatrix):
        for row in pMatrix:
                rowStr = "";
                for element in row:
                        rowStr += getNumberString(element, 7) + " ";
                print(rowStr);
#
```

Listing 1: mylib script to properly print vectors and matrices

1.1 Problem 1.1



Problem 1.1 Implement a tool that reads a matrix and right hand side vector of a system of linear equations from a file and does the following:

a) Write programs to compute the relevant information about the matrix using built-in functions: determinant, inverse, rank and the eigenvalues and eigenvectors, whether it is symmetric and positive definite.

b) Solve the linear system using a built in function (do not use the inverse matrix).

c) Program the Gaussian Elimination method described in section 5.2.1 for the matrix and print the resultant matrix (built-in function not to be used).

If any of the above does not exist/cannot be calculated, print the appropriate reason.

prog to read from txt

txt must have following form:

```
1 -2 3 -4;
4.1 3 1.1 1;
2 4 1 4;
3 4 3 8;
####
6.1 5.2 4.2 3.1;
```

first matrix
elements seperated by single space row ends with semicolon
hashtag sign to indicate that matrix is over, vector comes next
vector as row, read same as matrix row.
following script to read txt

```python
def getVectorFromLine(line):
        numbers = [];
        numberString = "";
        for c in line:
                if c == ' ' or c == ';':
                        numbers.append(float(numberString));
                        numberString = "";
                else:
                        numberString += c;

        return numbers;
#


myMatrix = [];
myVector = [];

textFile = open('matrixTextFile.txt', 'r');

readMatrix = True;

for line in textFile:
        if len(line) > 0:
                if line[0] == '#':
                        readMatrix = False;
                else:
                        vectorRead = getVectorFromLine(line);

                        if readMatrix: myMatrix.append(vectorRead);
                        else: myVector = vectorRead;


print("Matrix from file:");
mylib.printMatrix(myMatrix);

print("Vector from file:");
```

```
mylib.printVector(myVector);
```

Listing 2: todo

afterwards, you have mat and vec from txt as vars *myMatrix* and *myVector* as python arrays, where the vector is a one-dim array and the matrix a 2 dim array, with first dim rows and second dim the elements.

1.1.1 a)

for in-built functions numpy (np).
np = numpy
easy / self-explanatory

```
matrixRows = len(myMatrix);
matrixCols = len(myMatrix[0]);
vectorRows = len(myVector);

isMatrixSquare = matrixRows == matrixCols;
notSquareStr = "matrix is not square";

matrixArr = np.array(myMatrix);
vectorArr = np.array(myVector);

print("determinant:");

if isMatrixSquare:
        detA = np.linalg.det(matrixArr);
        print(detA);
else:
        print(notSquareStr);

print("inverse:");

if isMatrixSquare:
        invA = np.linalg.inv(matrixArr);
        mylib.printMatrix(invA);
else:
        print(notSquareStr);

print("rank:");

rankA = np.linalg.matrix_rank(matrixArr);
print(rankA);

eigenValues = [];
eigenVectors = [];

if isMatrixSquare:
        eigA = np.linalg.eig(matrixArr);
        for eigi in eigA:
                for el in eigi:
                        if isinstance(el, np.ndarray):
                                eigenVectors.append(el);
                        else:
                                eigenValues.append(el);


        print("eigenvalues:");

        for eigenValue in eigenValues:
                print(eigenValue.real);

        print("eigenvectors:");

        for eigenVector in eigenVectors:
                print(eigenVector.real);

else:
        print("eigenvalues␣and␣eigenvectors");
        print(notSquareStr);

print("is␣symmetric:");

if isMatrixSquare:
        isSym = (matrixArr.transpose() == matrixArr).all();
        print(isSym);
else:
        print(notSquareStr);

print("is␣positive␣definite:");

if isMatrixSquare:
        isPosDef = np.all(np.linalg.eigvals(matrixArr));
        print(isPosDef);
else:
        print(notSquareStr);
```

Listing 3: todo

results:

```
determinant:
169.2

inverse:
  -0.13     0.43    -0.53     0.15
   0.11    -0.17     0.71    -0.28
   0.29    -0.24     0.3      0.03
  -0.11     0.01    -0.26     0.2

```

```
rank:
4

eigenvalues:
8.25668694865
5.18909337619
-0.222890162424
-0.222890162424

eigenvectors:
[-0.31568286   0.44289131   0.61685718   0.61685718]
[  -7.86684467e-05    3.70103040e-01   -4.34768294e-01   -4.34768294e
    -01]
[  0.38980754  -0.19285648  -0.40093074  -0.40093074]
[  0.86509792  -0.79352215   0.10527324   0.10527324]

is symmetric:
False

is positive definite:
True
```

Listing 4: Result of 1.1 a)

matrix is square (evtl noch andere conditions) therefore all things could be done.

1.1.2 b)

to solve the lin system, check whether matrix cols and vector rows are equal. afterwards, use built-in (numpy again).

```

if not isMatrixSquare:
        print(notSquareStr);
        sys.exit();

if matrixCols != vectorRows:
        print("number of matrix columns and vector rows aren't 
            equal");
        sys.exit();

linSolutions = np.linalg.solve(matrixArr, vectorArr);

xNum = 1;

for linSolution in linSolutions:
        print("x" + str(xNum) + " = " + str(linSolution));
        xNum += 1;
```

Listing 5: todo

XS

```
x1 = -0.33829787234
x2 = 1.88156028369
x3 = 1.88794326241
x4 = -1.13439716312
```

Listing 6: Result of 1.1 b)

1.1.3 c)

Y

```

def gaussElemMethod(matrix, vector):
        n = len(matrix[0]);

        for k in range(n - 1):
                maxL = -1;
                for l in range(k, n):
                        maxL = max(maxL, abs(matrix[l][k]));

                mFound = -1;

                for m in range(n):
                        if abs(matrix[m][k]) == maxL:
                                mFound = m;

                if mFound > 0 and matrix[mFound][k] == 0:
                        print("singular");
                        continue;

                for i in range(k + 1, n):
                        qik = matrix[i][k] / matrix[k][k];

                        for j in range(0, n):
                                matrix[i][j] = matrix[i][j] - qik *
                                     matrix[k][j];

                        vector[i] = vector[i] - qik * vector[k];



gaussElemMethod(myMatrix, myVector);

print("matrix after elimination:");
mylib.printMatrix(myMatrix);

print("vector after elimination:");
mylib.printVector(myVector);
```

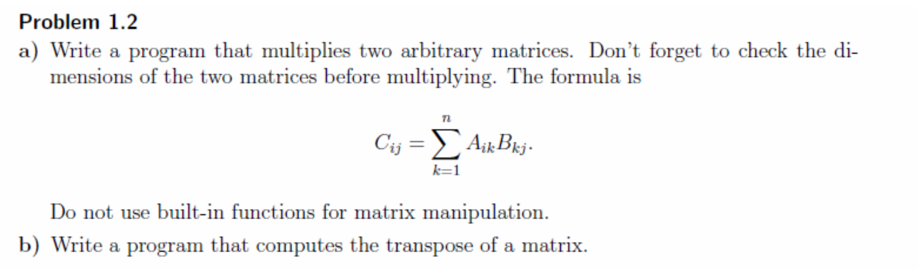
Listing 7: todo

Z

```
matrix after elimination:
     1.0     -2.0      3.0     -4.0
     0.0     11.2    -11.2     17.4
     0.0      0.0      3.0    -0.43
     0.0      0.0      0.0     5.04

vector after elimination:
     6.1
  -19.81
    6.15
   -5.71
```

Listing 8: Result of 1.1 c)

1.2 Problem 1.2



Problem 1.2

a) Write a program that multiplies two arbitrary matrices. Don't forget to check the dimensions of the two matrices before multiplying. The formula is

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Do not use built-in functions for matrix manipulation.

b) Write a program that computes the transpose of a matrix.

prog to read from txt

1.2.1 a)

Y

```

matrix1 = [[1.8, -2], [3, -4.1], [3, 2]];
matrix2 = [[1, -2, -3, 4], [-5, 4, 1, 1]];

print("matrix␣A:");
mylib.printMatrix(matrix1);

print("matrix␣B:");
mylib.printMatrix(matrix2);

print("A␣x␣B:");
```

```
matrix1Rows = len(matrix1);
matrix2Rows = len(matrix2);

matrix1Cols = len(matrix1[0]);
matrix2Cols = len(matrix2[0]);

resultMatrix = [];

if matrix1Cols != matrix2Rows:
        print("number of matrix A columns and matrix B rows aren't 
            equal");

for matrix1Row in range(matrix1Rows):

        resultVector = [];
        for matrix2Col in range(matrix2Cols):

                mySum = 0;

                for k in range(matrix1Cols):
                        mySum = mySum + matrix1[matrix1Row][k] *
                            matrix2[k][matrix2Col];

                resultVector.append(mySum);

        resultMatrix.append(resultVector);

mylib.printMatrix(resultMatrix);
```

Listing 9: todo

X

```
matrix A:
     1.8     -2.0
     3.0     -4.1
     3.0      2.0

matrix B:
     1.0     -2.0     -3.0      4.0
    -5.0      4.0      1.0      1.0

A x B:
    11.8    -11.6     -7.4      5.2
    23.5    -22.4    -13.1      7.9
    -7.0      2.0     -7.0     14.0
```

Listing 10: Result of 1.2 a)

1.2.2 b)

Y

```

matrix3 = [[1, 2.4], [3.3, -4], [-5, -6.1]];

print("matrix C:");
mylib.printMatrix(matrix3);

matrix3Rows = len(matrix3);
matrix3Cols = len(matrix3[0]);

transposedMatrix = [];

for j in range(matrix3Cols):
        transposedVector = [];

        for i in range(matrix3Rows):
                transposedVector.append(matrix3[i][j]);

        transposedMatrix.append(transposedVector);

print("C transposed:");
mylib.printMatrix(transposedMatrix);
```

Listing 11: todo

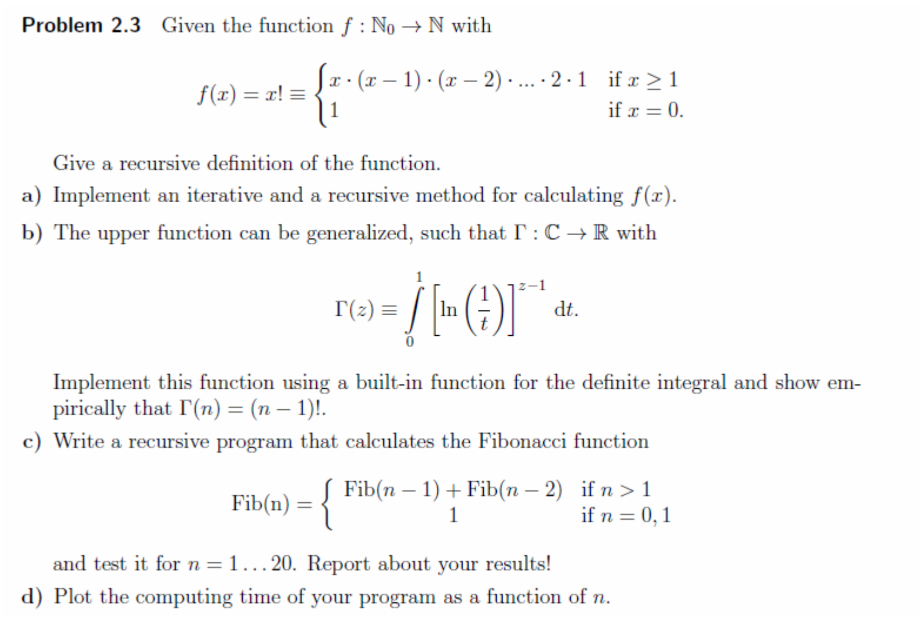
X

```
matrix C:
    1.0     2.4
    3.3    -4.0
   -5.0    -6.1

C transposed:
    1.0     3.3    -5.0
    2.4    -4.0    -6.1
```

Listing 12: Result of 1.2 b)

2 Calculus - Selected Topics

2.1 Problem 2.3



Problem 2.3 Given the function $f : \mathbb{N}_0 \to \mathbb{N}$ with

$$f(x) = x! \equiv \begin{cases} x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 2 \cdot 1 & \text{if } x \geq 1 \\ 1 & \text{if } x = 0. \end{cases}$$

Give a recursive definition of the function.

a) Implement an iterative and a recursive method for calculating $f(x)$.

b) The upper function can be generalized, such that $\Gamma : \mathbb{C} \to \mathbb{R}$ with

$$\Gamma(z) \equiv \int_0^1 \left[\ln\left(\frac{1}{t}\right)\right]^{z-1} dt.$$

Implement this function using a built-in function for the definite integral and show empirically that $\Gamma(n) = (n-1)!$.

c) Write a recursive program that calculates the Fibonacci function

$$\text{Fib(n)} = \begin{cases} \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 0, 1 \end{cases}$$

and test it for $n = 1 \ldots 20$. Report about your results!

d) Plot the computing time of your program as a function of n.

2.1.1 a)

X

```python
n = int(input("n = "));

factorialIterative = 1;

for i in range(n):
    factorialIterative *= (i + 1);


print("iterative result: " + str(factorialIterative));

def factorialFunc(x):
    if x <= 0: return 1;

    return x * factorialFunc(x - 1);


factorialRecursive = factorialFunc(n);

print("recursive result: " + str(factorialRecursive));
```

Listing 13: todo

results:

```
R
```

Listing 14: Result of 1.1 a)

X

2.1.2 b)

X

```
import scipy.integrate as itg;

def bFunc(t, z):
        return math.pow(math.log(1 / t), z - 1);


for i in range(1, 10):
        factorialResult = factorialFunc(i - 1);
        integrateResult = itg.quad(lambda x: bFunc(x, i), 0, 1);

        print("n = " + str(i));
        print("(n - 1)! = " + str(factorialResult));
        print("Gamma(n) = " + str(integrateResult));
```

Listing 15: todo

results:

```
R
```

Listing 16: Result of 1.1 a)

X

2.1.3 c)

X

```

def fibonacciFunc(x):
        if x <= 1: return 1;

        return fibonacciFunc(x - 1) + fibonacciFunc(x - 2);


results = [];

```

```
for i in range(20):
        fib = fibonacciFunc(i);
        print("fib(" + str(i + 1) + ") = " + str(fib));
        results.append(fib);
```

Listing 17: todo

results:

```
R
```

Listing 18: Result of 1.1 a)

X

2.1.4 d)

X

```

runtimes = [];

for i in range(30):
        timeStart = time.time();
        fibonacciFunc(i);
        ms = 1000 * (time.time() - timeStart);
        runtimes.append(ms);


plt.plot(runtimes);
plt.xlabel('x');
plt.ylabel('runtime fib(x) [ms]');
plt.show();
```

Listing 19: todo

results:

2.2 Problem 2.4

X

```

C
```

Listing 20: todo

Y

d)

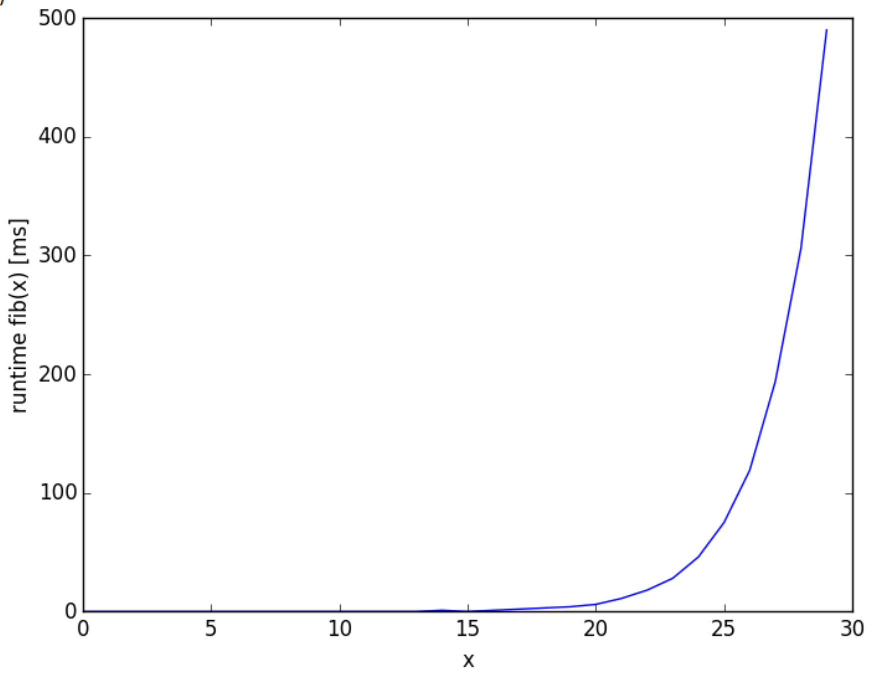
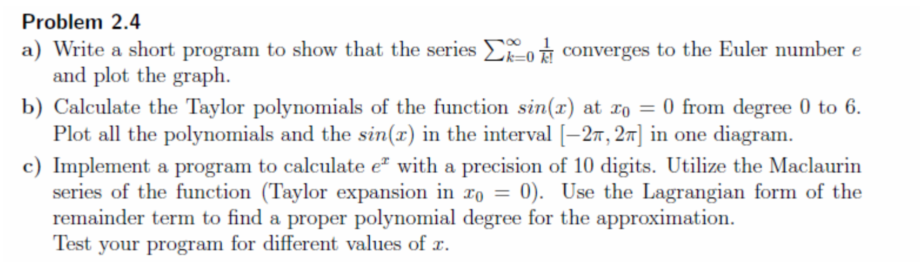


Figure 1: todo



Problem 2.4

a) Write a short program to show that the series $\sum_{k=0}^{\infty} \frac{1}{k!}$ converges to the Euler number e and plot the graph.

b) Calculate the Taylor polynomials of the function $sin(x)$ at $x_0 = 0$ from degree 0 to 6. Plot all the polynomials and the $sin(x)$ in the interval $[-2\pi, 2\pi]$ in one diagram.

c) Implement a program to calculate e^x with a precision of 10 digits. Utilize the Maclaurin series of the function (Taylor expansion in $x_0 = 0$). Use the Lagrangian form of the remainder term to find a proper polynomial degree for the approximation.
Test your program for different values of x.

2.2.1 a)

X

```

C
```

Listing 21: todo

results:

```
R
```

Listing 22: Result of 1.1 a)

X

2.2.2 b)

X

```

C
```

Listing 23: todo

results:

```
R
```

Listing 24: Result of 1.1 a)

X

2.2.3 c)

X

```

C
```

Listing 25: todo

results:

```
R
```

Listing 26: Result of 1.1 a)

X

2.2.4 d)

X

```

C
```

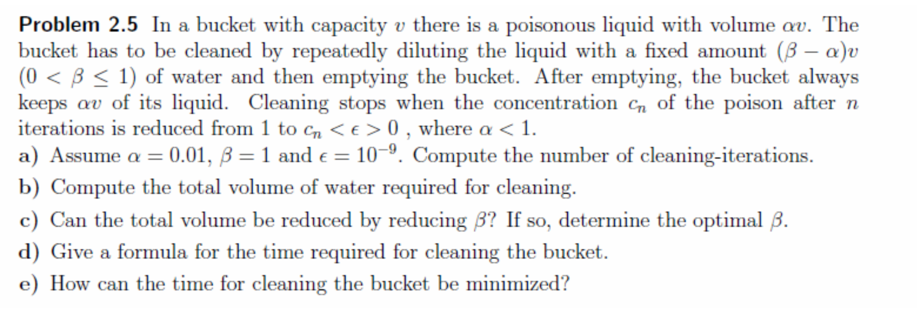
Listing 27: todo

results:

```
R
```

Listing 28: Result of 1.1 a)

X



Problem 2.5 In a bucket with capacity v there is a poisonous liquid with volume αv. The bucket has to be cleaned by repeatedly diluting the liquid with a fixed amount $(\beta - \alpha)v$ $(0 < \beta \leq 1)$ of water and then emptying the bucket. After emptying, the bucket always keeps αv of its liquid. Cleaning stops when the concentration c_n of the poison after n iterations is reduced from 1 to $c_n < \epsilon > 0$, where $\alpha < 1$.

a) Assume $\alpha = 0.01$, $\beta = 1$ and $\epsilon = 10^{-9}$. Compute the number of cleaning-iterations.

b) Compute the total volume of water required for cleaning.

c) Can the total volume be reduced by reducing β? If so, determine the optimal β.

d) Give a formula for the time required for cleaning the bucket.

e) How can the time for cleaning the bucket be minimized?

2.3 Problem 2.5

X

```

C
```

Listing 29: todo

Y

2.3.1 a)

X

```

C
```

Listing 30: todo

results:

```
R
```

Listing 31: Result of 1.1 a)

X

2.3.2 b)

X

```

C
```

Listing 32: todo

results:

```
R
```

Listing 33: Result of 1.1 a)

X

2.3.3 c)

X

```

C
```

Listing 34: todo

results:

```
R
```

Listing 35: Result of 1.1 a)

X

2.3.4 d)

X

```

C
```

Listing 36: todo

results:

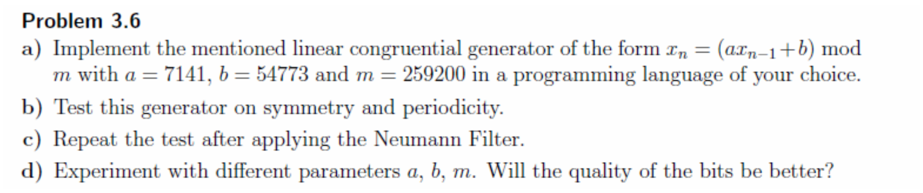
```
R
```

Listing 37: Result of 1.1 a)

X

3 Statistics and Probability

3.1 Problem 3.6



Problem 3.6

a) Implement the mentioned linear congruential generator of the form $x_n = (ax_{n-1} + b) \bmod m$ with $a = 7141$, $b = 54773$ and $m = 259200$ in a programming language of your choice.

b) Test this generator on symmetry and periodicity.

c) Repeat the test after applying the Neumann Filter.

d) Experiment with different parameters a, b, m. Will the quality of the bits be better?

X

```

C
```

Listing 38: todo

Y

3.1.1 a)

X

```

C
```

Listing 39: todo

results:

```
R
```

Listing 40: Result of 1.1 a)

X

3.1.2 b)

X

```

C
```

Listing 41: todo

results:

```
R
```

Listing 42: Result of 1.1 a)

X

3.1.3 c)

X

```

C
```

Listing 43: todo

results:

```
R
```

Listing 44: Result of 1.1 a)

X

3.1.4 d)

X

```

C
```

Listing 45: todo

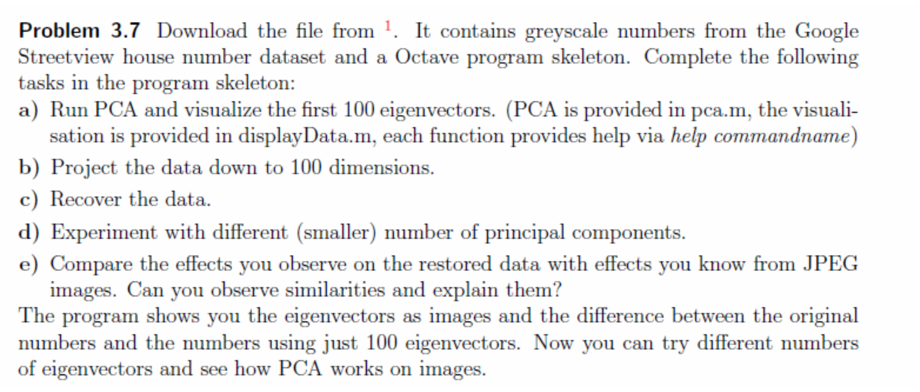
results:

```
R
```

Listing 46: Result of 1.1 a)

X

3.2 Problem 3.7



Problem 3.7 Download the file from [1]. It contains greyscale numbers from the Google Streetview house number dataset and a Octave program skeleton. Complete the following tasks in the program skeleton:

a) Run PCA and visualize the first 100 eigenvectors. (PCA is provided in pca.m, the visualisation is provided in displayData.m, each function provides help via *help commandname*)

b) Project the data down to 100 dimensions.

c) Recover the data.

d) Experiment with different (smaller) number of principal components.

e) Compare the effects you observe on the restored data with effects you know from JPEG images. Can you observe similarities and explain them?

The program shows you the eigenvectors as images and the difference between the original numbers and the numbers using just 100 eigenvectors. Now you can try different numbers of eigenvectors and see how PCA works on images.

X

```
C
```

Listing 47: todo

Y

3.2.1 a)

X

```
C
```

Listing 48: todo

results:

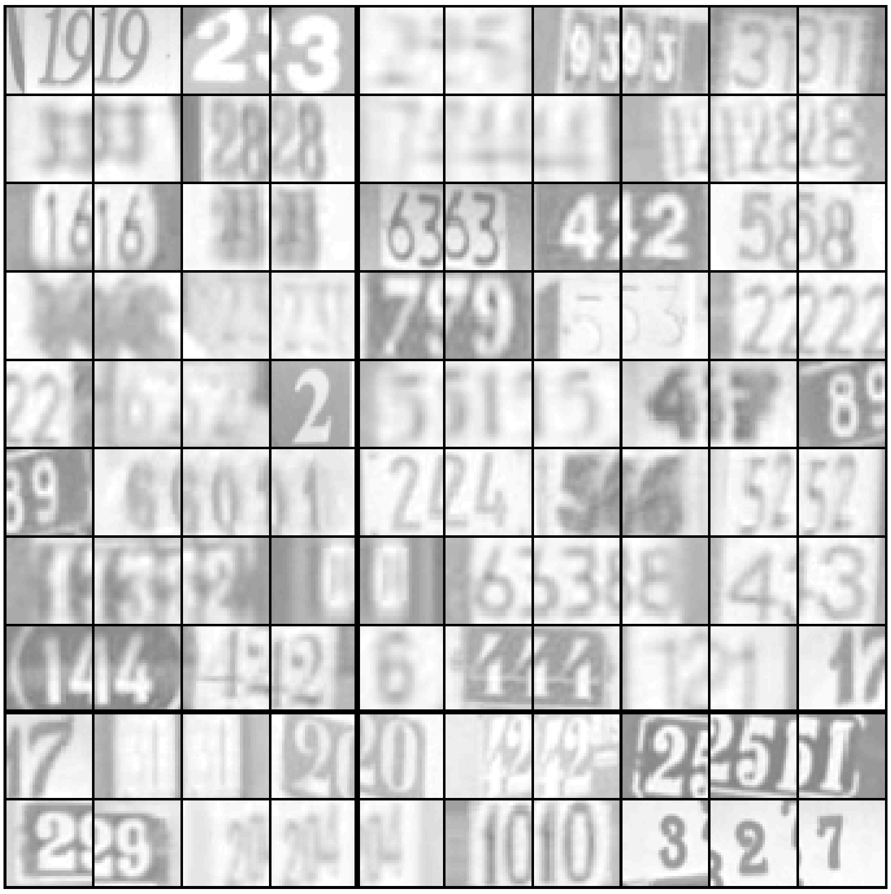


Figure 2: todo

X

3.2.2 b)

X

```
C
```

Listing 49: todo

results:

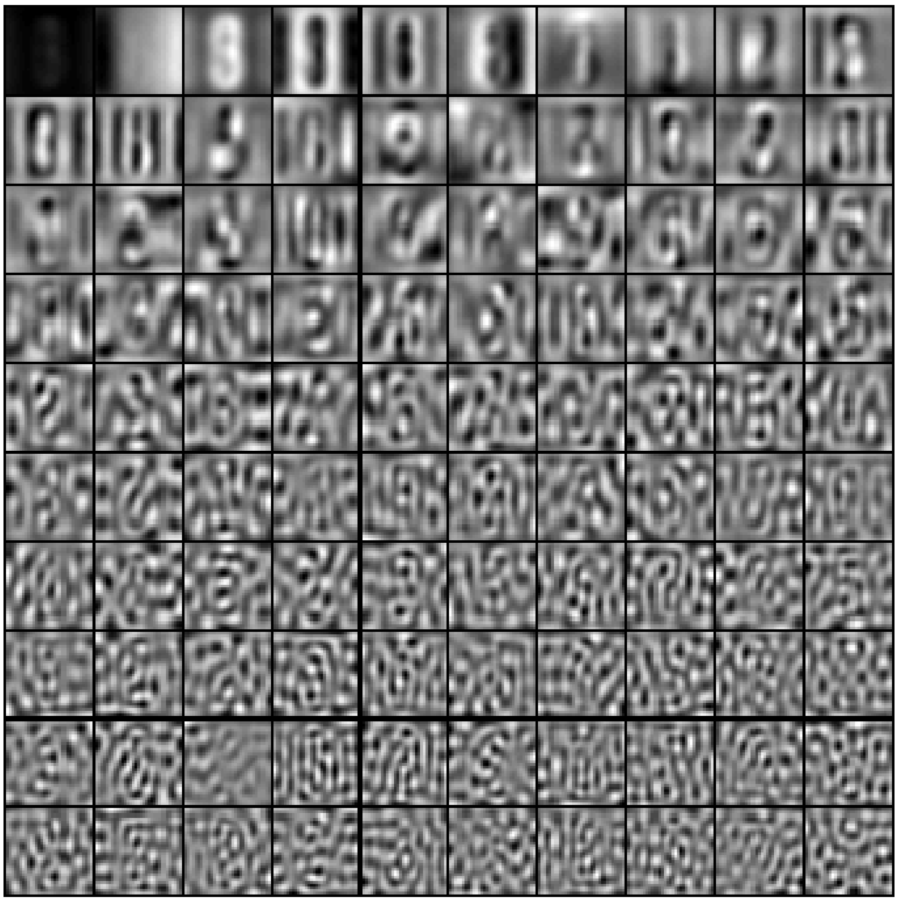


Figure 3: todo

X

3.2.3 c)

X

```
C
```

Listing 50: todo

results:

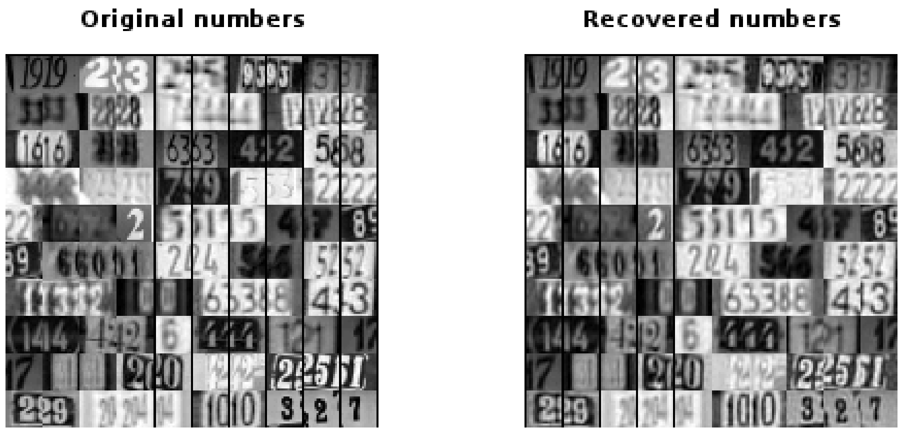


Figure 4: todo

X

3.2.4 d)

X

```
C
```

Listing 51: todo

results:
X

3.2.5 e)

Like the JPEG compression detail gets lost, especially between two areas where the color contrast is high.

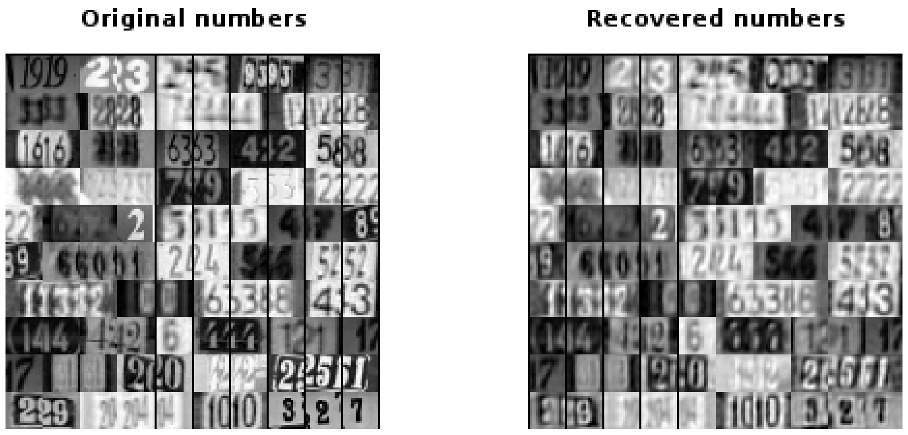


Figure 5: todo - 50 principal components

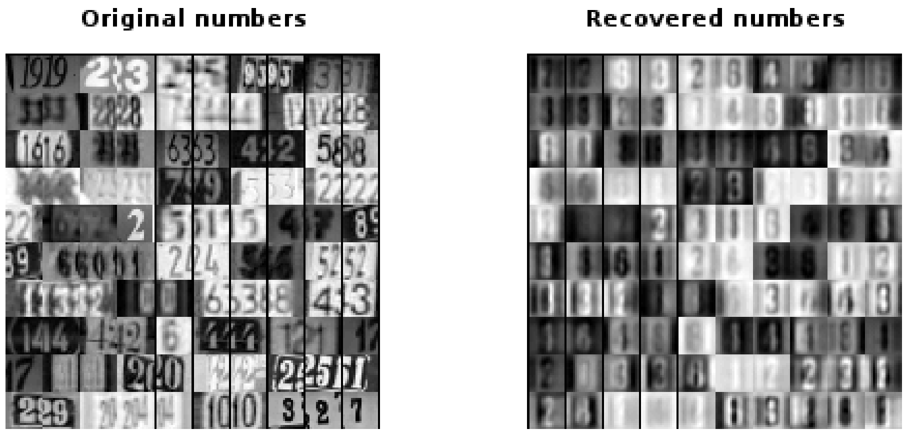
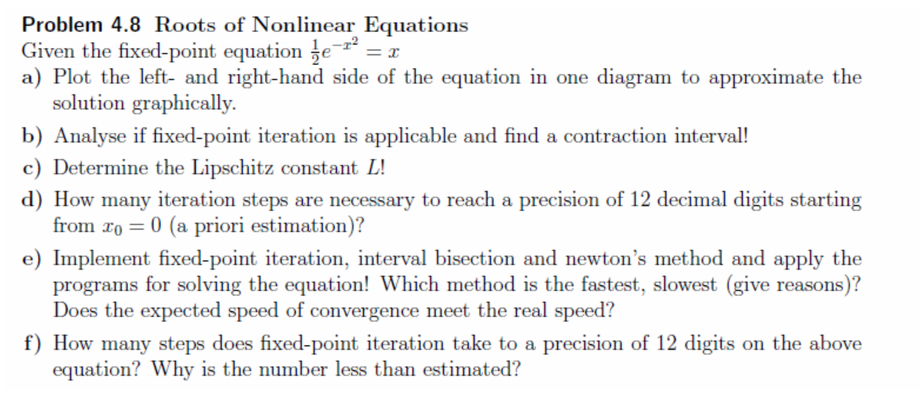


Figure 6: todo - 10 principal components

4 Numerical Mathematics Fundamendals

4.1 Problem 4.8



Problem 4.8 Roots of Nonlinear Equations
Given the fixed-point equation $\frac{1}{2}e^{-x^2} = x$

a) Plot the left- and right-hand side of the equation in one diagram to approximate the solution graphically.

b) Analyse if fixed-point iteration is applicable and find a contraction interval!

c) Determine the Lipschitz constant L!

d) How many iteration steps are necessary to reach a precision of 12 decimal digits starting from $x_0 = 0$ (a priori estimation)?

e) Implement fixed-point iteration, interval bisection and newton's method and apply the programs for solving the equation! Which method is the fastest, slowest (give reasons)? Does the expected speed of convergence meet the real speed?

f) How many steps does fixed-point iteration take to a precision of 12 digits on the above equation? Why is the number less than estimated?

X

```

def fFixed(x):
        return 0.5 * pow(math.e, -pow(x, 2));


def fFixedDerived(x):
        return -pow(math.e, -pow(x, 2)) * x;


def f(x):
        return fFixed(x) - x;


def fDerived(x):
        return fFixedDerived(x) - 1;
```

Listing 52: todo

Y

4.1.1 a)

X

```

xses = [];
fLeft = [];
fRight = [];

for i in range(200):
        x = (i / 100.0) - 1;

        xses.append(x);
        fLeft.append(fFixed(x));
        fRight.append(x);


plt.plot(xses, fLeft);
plt.plot(xses, fRight);
plt.xlabel('x');
plt.ylabel('y');
plt.show();
```

Listing 53: todo

results:

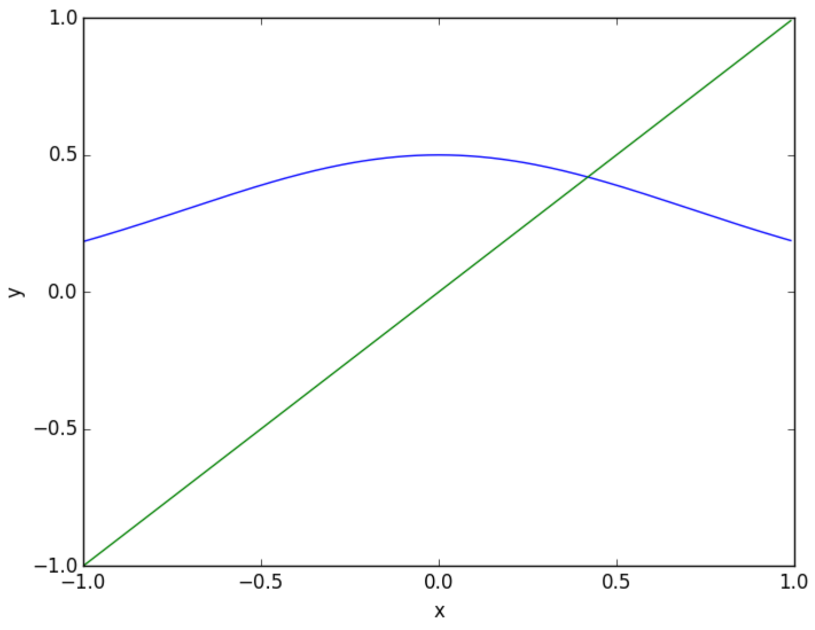


Figure 7: todo

graphical solution: little less than 0.5

4.1.2 b)

fixed-point iteration is applicable since 0 ¡ L ¡ 1 (see c)) contraction interval: [0, 1] -¿ [0, 1]

4.1.3 c)

X

```

# -0.70711 and 0.70711 are roots of the 2nd derivative of f
lipschitz = abs(fFixedDerived(0.70711));

print("L = " + str(lipschitz));
```

Listing 54: todo

results:

```
L = 0.428881942471
```

Listing 55: Result of 1.1 a)

X

4.1.4 d)

X

```

steps = 0;
err = 1;
xdiff = f(0);

while err > 1e-12:
        steps = steps + 1;
        err = (pow(lipschitz, steps) / (1 - lipschitz)) * xdiff;

print(str(steps) + " iteration steps required");
```

Listing 56: todo

results:

```
33 iteration steps required
```

Listing 57: Result of 1.1 a)

4.1.5 e)

```

def isPreciseEnough(val, goal, precision):
        f = pow(10, precision);

        return math.floor(val * f) == math.floor(goal * f);

```

```

def getFixedPoint(x0, goal, precision):
        x = x0;
        steps = 1;

        while steps < 100:
                x = fFixed(x);

                if (isPreciseEnough(x, goal, precision)):
                        break;

                steps = steps + 1;

        return steps;


def mySign(x):
        return -1 if x < 0 else 1;


def getRootBisection(pXA, pXB, goal, precision):
        xa = pXA;
        xb = pXB;
        xm = -1;
        steps = 1;

        while steps < 100:
                xm = xa + (xb - xa) / 2.0;

                if (isPreciseEnough(xm, goal, precision)):
                        break;

                if mySign(f(xa)) != mySign(f(xm)):
                        xb = xm;
                else:
                        xa = xm;

                steps = steps + 1;

        return steps;


def getRootNewton(x0, goal, precision):
        x = x0;
        steps = 1;

        while steps < 100:
                fD = fDerived(x);

                if abs(fD) < 0.00001: fD = 0.00001;
                x = x - float(f(x) / float(fD));

                if (isPreciseEnough(x, goal, precision)):
                        break;

                steps = steps + 1;

        return steps;


def getTime():
        return time.time();


def printTimeDiff(startTime, endTime):
        averageMS = round((endTime - startTime) / 10.0, 5);
        print("average time taken: " + str(averageMS) + " milliseconds");


def findRoot(type, goal, precision, p1 = 0, p2 = 0):
        startTime = getTime();

        steps = 0;

        for i in range(10000):
                param1 = p1 + 0.000001 * math.sin(i * 7);

                if (type == 0): steps = getFixedPoint(param1, goal, precision);
                elif (type == 1): steps = getRootBisection(param1, p2, goal, precision);
                else: steps = getRootNewton(param1, goal, precision);

        print("steps taken: " + str(steps));
        printTimeDiff(startTime, getTime());



def getFixedPointRec2(x, maxSteps, stepsTaken):
        newX = fFixed(x);

        if stepsTaken < maxSteps:
                return getFixedPointRec2(newX, maxSteps, stepsTaken + 1);
        else:
                return newX;

```

```
def getFixedPointRec1(x0, maxSteps):
        return getFixedPointRec2(x0, maxSteps, 1);


fP33 = getFixedPointRec1(0, 100);

precision = 6;
print("correct digits required: " + str(precision));

print("fixed point iteration:");
findRoot(0, fP33, precision, 0, -1);

print("interval bisection:");
findRoot(1, fP33, precision, 0, 1);

print("newton's method:");
findRoot(2, fP33, precision, 0, -1);

precision = 12;
print("correct digits required: " + str(precision));

print("fixed point iteration:");
findRoot(0, fP33, precision, 0, -1);

print("interval bisection:");
findRoot(1, fP33, precision, 0, 1);

print("newton's method:");
findRoot(2, fP33, precision, 0, -1);

precision = 18;
print("correct digits required: " + str(precision));

print("fixed point iteration:");
findRoot(0, fP33, precision, 0, -1);

print("interval bisection:");
findRoot(1, fP33, precision, 0, 1);

print("newton's method:");
findRoot(2, fP33, precision, 0, -1);
```

Listing 58: todo

results:

```
correct digits required: 6
fixed point iteration:
steps taken: 14
average time taken: 0.0382 milliseconds
```

```
interval bisection:
steps taken: 17
average time taken: 0.0914 milliseconds

newton's method:
steps taken: 4
average time taken: 0.0237 milliseconds

correct digits required: 12
fixed point iteration:
steps taken: 27
average time taken: 0.1185 milliseconds

interval bisection:
steps taken: 40
average time taken: 0.2703 milliseconds

newton's method:
steps taken: 4
average time taken: 0.0321 milliseconds

correct digits required: 18
fixed point iteration:
steps taken: 35
average time taken: 0.1501 milliseconds

interval bisection:
steps taken: 51
average time taken: 0.3547 milliseconds

newton's method:
steps taken: 5
average time taken: 0.0377 milliseconds
```

Listing 59: Result of 1.1 a)

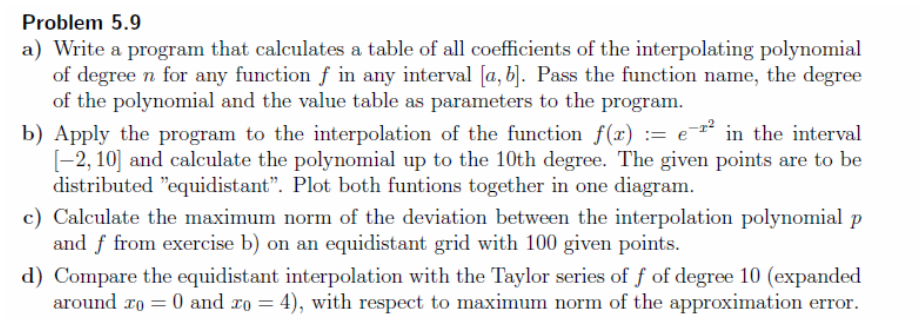
Newton's method is the fastest because it also utilizes the first derivative of the function.
Interval bisection is the slowest because by always using the middle of the interval as a bound, it reaches the root the slowest.

4.1.6 f)

27 steps were needed instead of the estimated 33 steps. It took less steps because the a priori estimation always asumes the worst case scenario.

5 Function Approximation

5.1 Problem 5.9



Problem 5.9

a) Write a program that calculates a table of all coefficients of the interpolating polynomial of degree n for any function f in any interval $[a, b]$. Pass the function name, the degree of the polynomial and the value table as parameters to the program.

b) Apply the program to the interpolation of the function $f(x) := e^{-x^2}$ in the interval $[-2, 10]$ and calculate the polynomial up to the 10th degree. The given points are to be distributed "equidistant". Plot both funtions together in one diagram.

c) Calculate the maximum norm of the deviation between the interpolation polynomial p and f from exercise b) on an equidistant grid with 100 given points.

d) Compare the equidistant interpolation with the Taylor series of f of degree 10 (expanded around $x_0 = 0$ and $x_0 = 4$), with respect to maximum norm of the approximation error.

5.1.1 a)

X

```
def getCoefficients(fx, degree, a, b):
        xses = [];
        yses = [];

        step = (b - a) / float(degree - 1);

        for i in range(degree):
                x = a + i * step;
                xses.append(x);
                yses.append(fx(x));

        matA = [];

        for xVal in xses:
                row = [];
                for i in range(degree):
                        row.append(pow(xVal, i));

                matA.append(row);

        matrixArr = np.array(matA);
        vectorArr = np.array(yses);

        linSolutions = np.linalg.solve(matrixArr, vectorArr);

        return linSolutions;
#
```

Listing 60: todo

results:

```
R
```

Listing 61: Result of 1.1 a)

X

5.1.2 b)

X

```

def polynomialF(coefficients, x):
        sum = 0.0;

        for i in range(len(coefficients)):
                sum += coefficients[i] * pow(x, i);

        return sum;
#

def myF(x):
        return pow(math.e, -pow(x, 2));
#


coefficients = getCoefficients(myF, 10, -2, 10);

xses = [];
realYses = [];
approxYses = [];

for i in range(100):
        x = -2 + i * (12 / 100.0);

        xses.append(x);
        realYses.append(myF(x));
        approxYses.append(polynomialF(coefficients, x));


plt.plot(xses, realYses);
plt.plot(xses, approxYses);
plt.xlabel("x");
plt.ylabel("y");
plt.show();
```

Listing 62: todo

results:

X

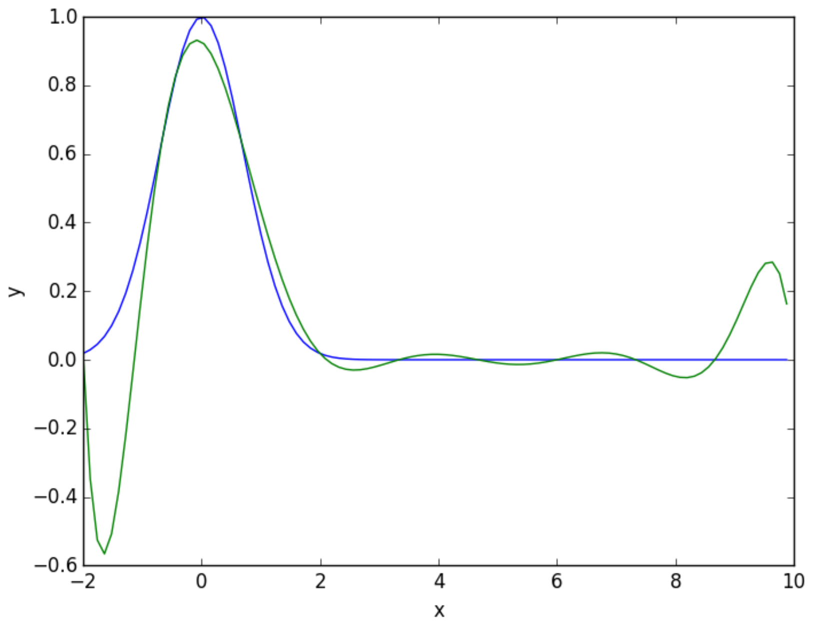


Figure 8: todo

5.1.3 c)

X

```

maxApproxDev = −1;

for i in range(100):
        realY = realYses[i];
        approxY = approxYses[i];

        maxApproxDev = max(maxApproxDev, abs(realY − approxY));


print("maximum deviation: " + str(maxApproxDev));
```

Listing 63: todo

results:

```
maximum deviation: 0.634025332357
```

Listing 64: Result of 1.1 a)

X

5.1.4 d)

X

```


def taylorFuncAt0(x):
        sum = 1;
        sum -= pow(x, 2);
        sum += pow(x, 4) / 2.0;
        sum -= pow(x, 6) / 6.0;
        sum += pow(x, 8) / 24.0;
        sum -= pow(x, 10) / 120.0;
        sum += pow(x, 12) / 720.0;
        sum -= pow(x, 14) / 5040.0;
        sum += pow(x, 16) / 40320.0;

        return sum;
#

def taylorFuncAt4(x):
        ep16 = float(pow(math.e, 16));
        xm4 = x - 4;

        sum = 1 / ep16;
        sum -= 8 * xm4 / ep16;
        sum += 31 * pow(xm4, 2) / ep16;
        sum -= 232 * pow(xm4, 3) / (3 * ep16);
        sum += 835 * pow(xm4, 4) / (6 * ep16);
        sum -= 2876 * pow(xm4, 5) / (15 * ep16);
        sum += 18833 * pow(xm4, 6) / (90 * ep16);
        sum -= 58076 * pow(xm4, 7) / (315 * ep16);
        sum += 332777 * pow(xm4, 8) / (2520 * ep16);
        sum -= 43325 * pow(xm4, 9) / (567 * ep16);
        sum += 3937007 * pow(xm4, 10) / (113400 * ep16);

        return sum;
#

taylor0Yses = [];
taylor4Yses = [];

for i in range(100):
        x = -2 + i * (12 / 100.0);

        taylor0Yses.append(taylorFuncAt0(x));
        taylor4Yses.append(taylorFuncAt4(x));


maxTaylor0Dev = -1;
maxTaylor4Dev = -1;
```

```
for i in range(100):
        realY = realYses[i];
        taylor0Y = taylor0Yses[i];
        taylor4Y = taylor4Yses[i];

        maxTaylor0Dev = max(maxTaylor0Dev, abs(realY - taylor0Y));
        maxTaylor4Dev = max(maxTaylor4Dev, abs(realY - taylor4Y));


print("maximum deviation of taylor series with c = 0: " + str(
    maxTaylor0Dev));
print("maximum deviation of taylor series with c = 4: " + str(
    maxTaylor4Dev));
```

Listing 65: todo

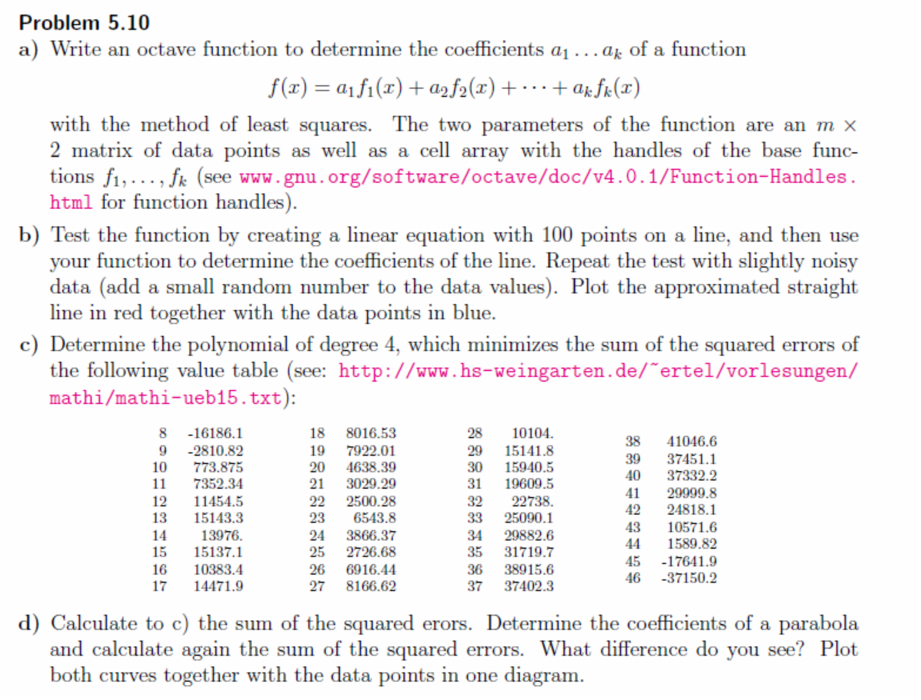
results:

```
maximum deviation of taylor series with c = 0: 1.88827128486e+11
maximum deviation of taylor series with c = 4: 354.936464804
```

Listing 66: Result of 1.1 a)

X

5.2 Problem 5.10



Problem 5.10

a) Write an octave function to determine the coefficients $a_1 \dots a_k$ of a function

$$f(x) = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_k f_k(x)$$

with the method of least squares. The two parameters of the function are an $m \times 2$ matrix of data points as well as a cell array with the handles of the base functions $f_1, \dots, f_k$ (see `www.gnu.org/software/octave/doc/v4.0.1/Function-Handles.html` for function handles).

b) Test the function by creating a linear equation with 100 points on a line, and then use your function to determine the coefficients of the line. Repeat the test with slightly noisy data (add a small random number to the data values). Plot the approximated straight line in red together with the data points in blue.

c) Determine the polynomial of degree 4, which minimizes the sum of the squared errors of the following value table (see: `http://www.hs-weingarten.de/~ertel/vorlesungen/mathi/mathi-ueb15.txt`):

8	-16186.1	18	8016.53	28	10104.	38	41046.6
9	-2810.82	19	7922.01	29	15141.8	39	37451.1
10	773.875	20	4638.39	30	15940.5	40	37332.2
11	7352.34	21	3029.29	31	19609.5	41	29999.8
12	11454.5	22	2500.28	32	22738.	42	24818.1
13	15143.3	23	6543.8	33	25090.1	43	10571.6
14	13976.	24	3866.37	34	29882.6	44	1589.82
15	15137.1	25	2726.68	35	31719.7	45	-17641.9
16	10383.4	26	6916.44	36	38915.6	46	-37150.2
17	14471.9	27	8166.62	37	37402.3		

d) Calculate to c) the sum of the squared erors. Determine the coefficients of a parabola and calculate again the sum of the squared errors. What difference do you see? Plot both curves together with the data points in one diagram.

5.2.1 a)

X

```

def getCoefficients(baseFunctions, dataPoints):
        l = len(baseFunctions);

        matA = [];

        for i in range(l):
                matrixRow = [];
                for j in range(l):
                        elemSum = 0;

                        for dp in dataPoints:
                                x = dp[0];
                                elemSum += baseFunctions[i](x) *
                                    baseFunctions[j](x);

                        matrixRow.append(elemSum);

                matA.append(matrixRow);

        vecB = [];

        for i in range(l):
                elemSum = 0;

                for dp in dataPoints:
                        elemSum += dp[1] * baseFunctions[i](dp[0]);

                vecB.append(elemSum);


        matrixArr = np.array(matA);
        vectorArr = np.array(vecB);

        linSolutions = np.linalg.solve(matrixArr, vectorArr);

        return linSolutions;
#
```

Listing 67: todo

X

5.2.2 b)

X

```
def myF2(x): return pow(x, 2);

def myF1(x): return x;

def myF0(x): return 1;

baseFns = [];

baseFns.append(myF1);
baseFns.append(myF0);

linPoints = [];
rndPoints = [];
xses = [];
yses = [];
rndYses = [];

for i in range(100):
        x = i / 10.0;
        y = x * 0.64 + 2.3;

        xses.append(x);
        yses.append(y);

        rnd = random.random() - 0.5;
        rndY = y + rnd * 0.6;
        rndYses.append(rndY);

        linPoints.append([x, y]);
        rndPoints.append([x, rndY]);


print("coefficients:");
print(getCoefficients(baseFns, linPoints));

print("approximated coefficients:");
print(getCoefficients(baseFns, rndPoints));


plt.plot(xses, yses);
plt.plot(xses, rndYses, "ro");
plt.xlabel("x");
plt.ylabel("y");
plt.show();
```

Listing 68: todo

results:

```
coefficients:
```

```
[ 0.64     0.23 ]

approximated coefficients:
[ 0.63139257    2.33442881 ]
```

Listing 69: Result of 1.1 a)

X

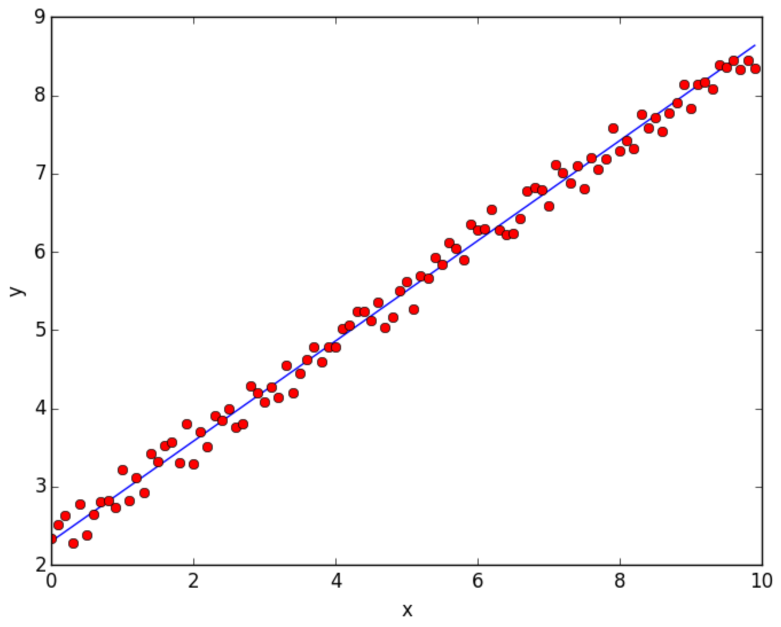


Figure 9: todo

5.2.3 c)

X

```


def myF4(x): return pow(x, 4);
def myF3(x): return pow(x, 3);
def myF2(x): return pow(x, 2);
def myF1(x): return x;
def myF0(x): return 1;


def polynomialF(coefficients, x):
        sum = 0.0;
        nc = len(coefficients);

        for i in range(nc):
                sum += coefficients[i] * pow(x, nc - i - 1);

        return sum;
#

txtBaseFns4 = [];
txtBaseFns2 = [];

txtBaseFns4.append(myF4);
txtBaseFns4.append(myF3);
txtBaseFns4.append(myF2);
txtBaseFns4.append(myF1);
txtBaseFns4.append(myF0);

txtBaseFns2.append(myF2);
txtBaseFns2.append(myF1);
txtBaseFns2.append(myF0);

txtPoints = [];

txtPoints.append([8, -16186.1]);
txtPoints.append([9, -2810.82]);
txtPoints.append([10, 773.875]);
txtPoints.append([11, 7352.34]);
txtPoints.append([12, 11454.5]);
txtPoints.append([13, 15143.3]);
txtPoints.append([14, 13976.]);
txtPoints.append([15, 15137.1]);
txtPoints.append([16, 10383.4]);
txtPoints.append([17, 14471.9]);
txtPoints.append([18, 8016.53]);
txtPoints.append([19, 7922.01]);
txtPoints.append([20, 4638.39]);
txtPoints.append([21, 3029.29]);
txtPoints.append([22, 2500.28]);
txtPoints.append([23, 6543.8]);
txtPoints.append([24, 3866.37]);
txtPoints.append([25, 2726.68]);
txtPoints.append([26, 6916.44]);
txtPoints.append([27, 8166.62]);
txtPoints.append([28, 10104.]);
txtPoints.append([29, 15141.8]);
txtPoints.append([30, 15940.5]);
txtPoints.append([31, 19609.5]);
txtPoints.append([32, 22738.]);
txtPoints.append([33, 25090.1]);
txtPoints.append([34, 29882.6]);
txtPoints.append([35, 31719.7]);
txtPoints.append([36, 38915.6]);
txtPoints.append([37, 37402.3]);
txtPoints.append([38, 41046.6]);
txtPoints.append([39, 37451.1]);
txtPoints.append([40, 37332.2]);
txtPoints.append([41, 29999.8]);
txtPoints.append([42, 24818.1]);
txtPoints.append([43, 10571.6]);
txtPoints.append([44, 1589.82]);
txtPoints.append([45, -17641.9]);
txtPoints.append([46, -37150.2]);

txtCoefficients4 = getCoefficients(txtBaseFns4, txtPoints);
txtCoefficients2 = getCoefficients(txtBaseFns2, txtPoints);

txtPointsXses = [];
txtPointsYses = [];

error4 = 0;
error2 = 0;

for i in range(len(txtPoints)):
        txtPoint = txtPoints[i];

        txtPointsXses.append(txtPoint[0]);
        txtPointsYses.append(txtPoint[1]);

        y4 = polynomialF(txtCoefficients4, i + 8);
        y2 = polynomialF(txtCoefficients2, i + 8);

        error4 += pow(y4 - txtPoint[1], 2);
        error2 += pow(y2 - txtPoint[1], 2);


print("error of degree 4 polynomial: " + str(error4));
print("error of degree 2 polynomial: " + str(error2));

txtFXses = [];
txtF4Yses = [];
txtF2Yses = [];

for i in range(152):
        x = 8 + i * 0.25;

        txtFXses.append(x);

        y4 = polynomialF(txtCoefficients4, x);
        y2 = polynomialF(txtCoefficients2, x);

```

```
            txtF4Yses.append(y4);
            txtF2Yses.append(y2);


plt.plot(txtFXses, txtF4Yses);
plt.plot(txtFXses, txtF2Yses);
plt.plot(txtPointsXses, txtPointsYses, "ro");
plt.xlabel("x");
plt.ylabel("y");
plt.show();
```

Listing 70: todo

results:

```
R
```

Listing 71: Result of 1.1 a)

X

5.2.4 d)

siehe c) - aufdröseln

```

C
```

Listing 72: todo

results:

```
error of degree 4 polynomial: 101457690.277
error of degree 2 polynomial: 7711489909.2
```

Listing 73: Result of 1.1 a)

X

Y

5.3 Problem 5.11

X

```

C
```

Listing 74: todo

Y

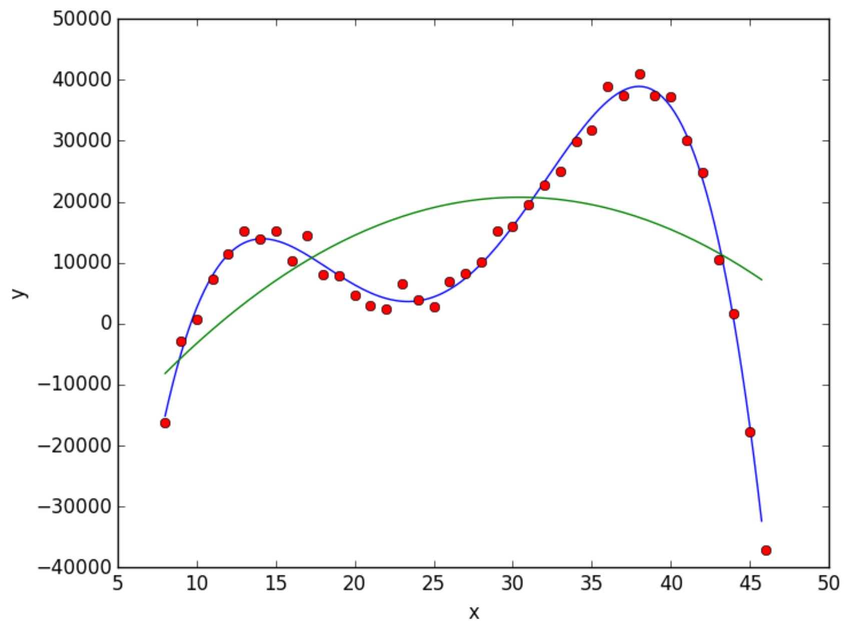


Figure 10: todo

Problem 5.11

a) From the data in problem 5.10 c), extract all data points with an even x within one line of code.

b) Divide these data points randomly in two matrices X and T of size 10×2 (each consisting of 10 data points).

c) X is called the training data, T the test data. Determine the polynomials of degree 2-8, each minimizing the sum of the squared errors on the training data X. Plot the seven polynomial functions together with the data points in one diagram. The points in X have to be plotted in blue and those in T with red. Plot each function in a different color using the following cell array of color characters: `{'b','c','r','g','m','y','k'}`

d) For each polynomial, calculate the sum of the squared errors on the training data X and the test data T. Plot both squared error sums in dependence of degree n in one diagram (using again blue for X and red for T).

e) Since we divide the data randomly to X and T, the outcome might change if we run the program again. Run the program from b) to c) several times. What is the decisive difference in the plots of the squared error sums for X and T and can you explain this difference? At what degree do we have the smallest error on the test data in most of the cases?

f) The method you applied is called cross validation. In practice, we would choose the degree which minimizes the error on the test data. Why does it make sense, to use different data subsets for the approximation and for the test?

5.3.1 a)

(getCoefficients und txtPoints von vorheriger Aufgabe) (ref zu Listing)

```

```

```
def myF8(x): return pow(x, 8);

def myF7(x): return pow(x, 7);

def myF6(x): return pow(x, 6);

def myF5(x): return pow(x, 5);

def myF4(x): return pow(x, 4);

def myF3(x): return pow(x, 3);

def myF2(x): return pow(x, 2);

def myF1(x): return x;

def myF0(x): return 1;


def getBaseFunctionList(degree):
        baseFunctions = [];

        if degree >= 8: baseFunctions.append(myF8);
        if degree >= 7: baseFunctions.append(myF7);
        if degree >= 6: baseFunctions.append(myF6);
        if degree >= 5: baseFunctions.append(myF5);
        if degree >= 4: baseFunctions.append(myF4);
        if degree >= 3: baseFunctions.append(myF3);
        if degree >= 2: baseFunctions.append(myF2);

        baseFunctions.append(myF1);
        baseFunctions.append(myF0);

        return baseFunctions;
#

def polynomialF(coefficients, x):
        sum = 0.0;

        nc = len(coefficients);

        for i in range(nc):
                sum += coefficients[i] * pow(x, nc - i - 1);

        return sum;
#


evenPoints = [];
```

```
for txtPoint in txtPoints:
        if txtPoint[0] % 2 == 0:
                evenPoints.append(txtPoint);


matrixX = [];
matrixT = [];

for evenPoint in evenPoints:
        rnd = random.random();

        if len(matrixT) >= 10 or (rnd < 0.5 and len(matrixX) < 10):
                matrixX.append(evenPoint);
        else:
                matrixT.append(evenPoint);



print("c)");

allYses = [];

xses = [];

for i in range(152):
        xses.append(8 + i * 0.25);


for i in range(7):
        degree = i + 2;

        baseFunctionList = getBaseFunctionList(degree);

        coefficients = getCoefficients(baseFunctionList, matrixX);

        yses = [];

        for x in xses:
                y = polynomialF(coefficients, x);
                yses.append(y);

        allYses.append(yses);


xXses = [];
xYses = [];
tXses = [];
tYses = [];

```

```
for i in range(10):
        xXses.append(matrixX[i][0]);
        xYses.append(matrixX[i][1]);
        tXses.append(matrixT[i][0]);
        tYses.append(matrixT[i][1]);


#colors = ["r", "g", "b", "c", "m", "y", "k"];
colors = ["b", "c", "r", "g", "m", "y", "k"];


plt.plot(xXses, xYses, "bo");
plt.plot(tXses, tYses, "ro");

for i in range(len(allYses)):
        plt.plot(xses, allYses[i], colors[i]);

plt.xlabel("x");
plt.ylabel("y");
plt.show();


print("␣");
print("d)");

def getError(coefficients, mat):
        errorSum = 0;

        for elem in mat:
                fY = polynomialF(coefficients, elem[0]);

                errorSum += pow(fY - elem[1], 2);

        return errorSum;
#

def getErrors(degree, matX, matT):
        baseFunctionList = getBaseFunctionList(degree);

        coefficients = getCoefficients(baseFunctionList, matX);

        errorSumX = getError(coefficients, matX);
        errorSumT = getError(coefficients, matT);

        return [errorSumX, errorSumT];
#

degrees = [];
errorsX = [];
errorsT = [];
```

```
for i in range(7):
        degree = i + 2;

        degrees.append(degree);

        errors = getErrors(degree, matrixX, matrixT);

        errorsX.append(errors[0]);
        errorsT.append(errors[1]);
#

plt.plot(degrees, errorsX, "b");
plt.plot(degrees, errorsT, "r");

plt.xlabel("degree");
plt.ylabel("error");
plt.show();
```

Listing 75: todo

results:

```
R
```

Listing 76: Result of 1.1 a)

X

5.3.2 b)

X

```

C
```

Listing 77: todo

results:

```
R
```

Listing 78: Result of 1.1 a)

X

5.3.3 c)

X

```

C
```

Listing 79: todo

results:

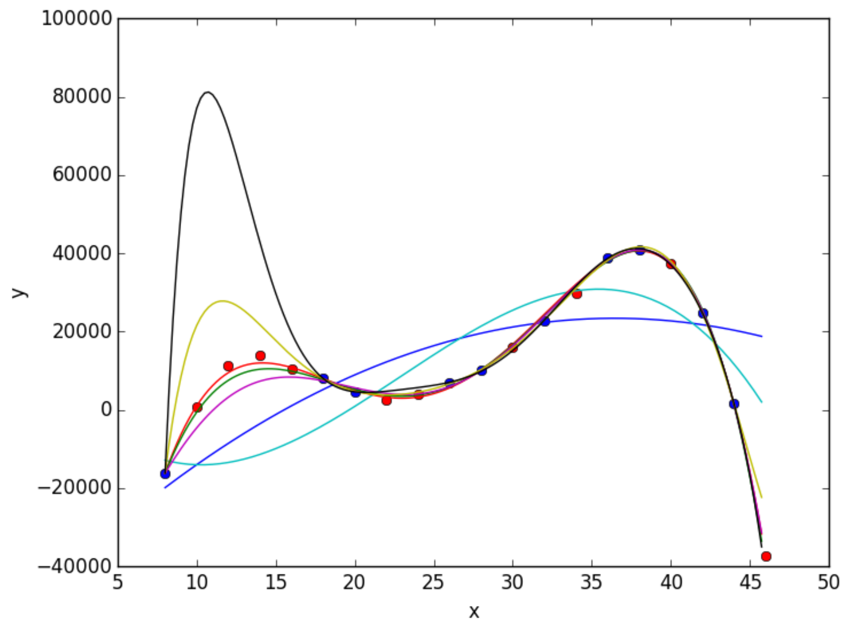


Figure 11: todo

X

5.3.4 d)

X

```
C
```

Listing 80: todo

results:
X

5.3.5 e)

The deviation to the original approximated function (where all data points were used) minimizes the more evenly the points of X and T are distributed. The reason for that is that when there's a large interval of test points only, the

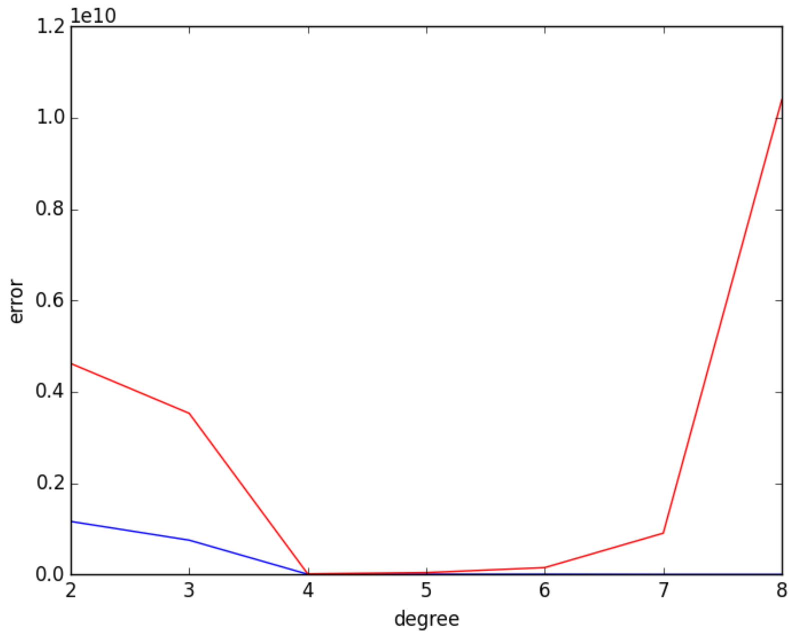


Figure 12: todo

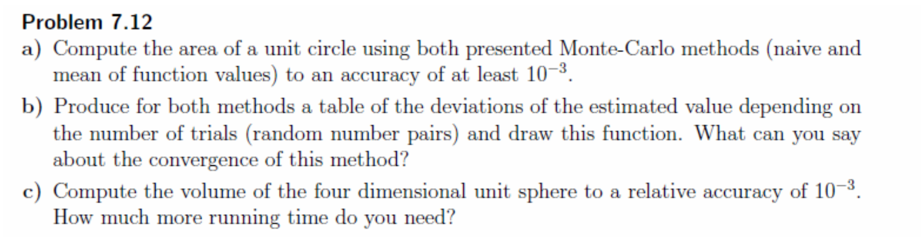
approximated function has no bounds at all in that interval and can deviate as much as it needs to, without increasing the error. The results seems to be the the best with polynomials of degree 4 and 5.

5.3.6 f)

It makes sense because the approximated function is supposed to be close to all data values, and even the points ”in between” which are not part of both the training and test data.

6 Numerical Integration and Solution of Ordinatry Diffential Equations

6.1 Problem 7.12


Problem 7.12

a) Compute the area of a unit circle using both presented Monte-Carlo methods (naive and mean of function values) to an accuracy of at least 10^{-3}.

b) Produce for both methods a table of the deviations of the estimated value depending on the number of trials (random number pairs) and draw this function. What can you say about the convergence of this method?

c) Compute the volume of the four dimensional unit sphere to a relative accuracy of 10^{-3}. How much more running time do you need?

X

```

C
```

Listing 81: todo

Y

6.1.1 a)

X

```
def getRandCoord(a, b):
        return random.random() * (b - a) + a;
#

def circleF(x):
        newX = (x % 2) - 1;
        return math.sqrt(1 - pow(newX, 2));
#

def doNaiveMonteCarloTest(xMin, xMax, yMin, yMax):
        randX = getRandCoord(xMin, xMax);
        randY = getRandCoord(yMin, yMax);

        return circleF(randX) > randY;
#

goal = math.pi;

def doNaiveMonteCarlo(precision):
        areaEstimated = 0;
        tries = 0.0;
        successes = 0.0;

        while abs(areaEstimated - goal) > precision and tries <
            100000:
                if doNaiveMonteCarloTest(0, 4, 0, 1):
                        successes += 1;

                tries += 1;

                areaEstimated = 4 * (successes / tries);

        return [areaEstimated, tries];
#

def doMeanMonteCarlo(precision):
        areaEstimated = 0;
        tries = 0.0;
        sum = 0.0;

        while abs(areaEstimated - goal) > precision and tries <
            100000:
                randX = getRandCoord(0, 4);

                sum += circleF(randX);
                tries += 1;

                areaEstimated = 4 * (sum / tries);

        return [areaEstimated, tries];
#


minError = 0.001;


naiveMonteCarlo = doNaiveMonteCarlo(minError);

print("naive method:");
print("estimated area: " + str(naiveMonteCarlo[0]));
print("tries: " + str(naiveMonteCarlo[1]));

meanMonteCarlo = doMeanMonteCarlo(minError);

print(" ");
print("mean value method:");
print("estimated area: " + str(meanMonteCarlo[0]));
print("tries: " + str(meanMonteCarlo[1]));
```

Listing 82: todo

results:

```
naive method:
```

```
estimated area: 3.14241702843
tries: 7141.0

mean value method:
estimated area: 3.14239623321
tries: 658.0
```

Listing 83: Result of 1.1 a)

X

6.1.2 b)

X

```

def getNaiveMonteCarloDeviation(tries):
        successes = 0.0;

        for i in range(tries):
                if doNaiveMonteCarloTest(0, 4, 0, 1):
                        successes += 1;

        areaEstimated = 4 * (successes / tries);

        return abs(areaEstimated - goal);
#

def getMeanMonteCarloDeviation(tries):
        sum = 0.0;

        for i in range(tries):
                randX = getRandCoord(0, 4);

                sum += circleF(randX);

        areaEstimated = 4 * (sum / tries);

        return abs(areaEstimated - goal);
#


xses = [];
naiveYses = [];
meanYses = [];

for i in range(1, 250):
        naiveDeviation = getNaiveMonteCarloDeviation(i * 10);
        meanDeviation = getMeanMonteCarloDeviation(i * 10);

        xses.append(i);
        naiveYses.append(naiveDeviation);
        meanYses.append(meanDeviation);


plt.plot(xses, naiveYses);
plt.plot(xses, meanYses);
plt.xlabel("tries");
plt.ylabel("deviation");
plt.show();
```

Listing 84: todo

results:

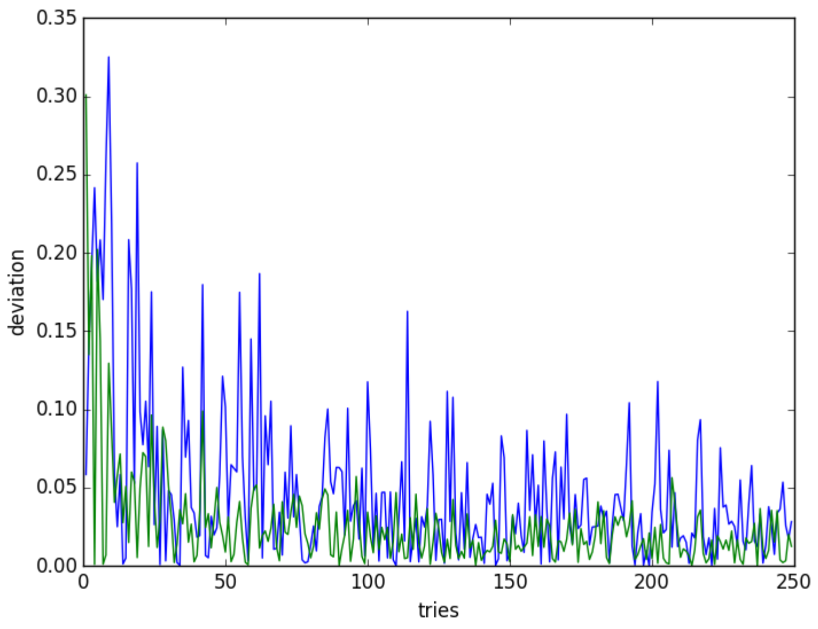


Figure 13: todo

X

6.1.3 c)

X

```
def isIn4DSphere4D(x, y, z, a):
        return math.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2) + pow(a,
            2)) < 1;
#

def doNaiveMonteCarloTest4D():
        randX = getRandCoord(0, 1);
        randY = getRandCoord(0, 1);
        randZ = getRandCoord(0, 1);
        randA = getRandCoord(0, 1);

        return isIn4DSphere4D(randX, randY, randZ, randA);
#

goal2 = pow(math.pi, 2) / 2.0;

def doNaiveMonteCarlo4D(precision):
        areaEstimated = 0;
        tries = 0.0;
        successes = 0.0;

        while abs(areaEstimated - goal2) > precision and tries <
            1000000:
                if doNaiveMonteCarloTest4D():
                        successes += 1;

                tries += 1;

                areaEstimated = 16 * (successes / tries);

        return [areaEstimated, tries];
#



fourDSphere = doNaiveMonteCarlo4D(minError);
print("estimated area of 4D unit sphere: " + str(fourDSphere[0]));

total2DTries = 0.0;
total4DTries = 0.0;

runs = 250;

for i in range(runs):
        total2DTries += doNaiveMonteCarlo(minError)[1];
        total4DTries += doNaiveMonteCarlo4D(minError)[1];

avg2DTries = total2DTries / float(runs);
avg4DTries = total4DTries / float(runs);

print(" ");
print("average tries for 2d unit circle: " + str(avg2DTries));
print("average tries for 4d unit sphere: " + str(avg4DTries));
```

Listing 85: todo

The deviation tends to get smaller, yet it doesn't converge.

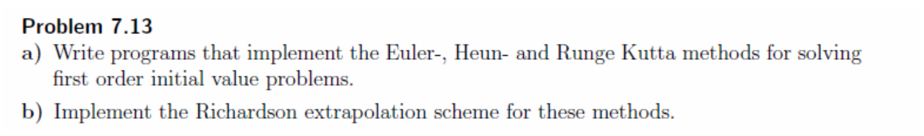
results:

```
estimated area of 4D unit sphere: 4.93493975904

average tries for 2d unit circle: 4928.92
average tries for 4d unit sphere: 14141.112
```

Listing 86: Result of 1.1 a)

X

6.2 Problem 7.13



Problem 7.13

a) Write programs that implement the Euler-, Heun- and Runge Kutta methods for solving first order initial value problems.

b) Implement the Richardson extrapolation scheme for these methods.

6.2.1 a)

X

```

def odeSolver(f, a, b, y0, h, yp1Method):
        yses = [];

        x = a;
        y = y0;

        yses.append(y);

        while x <= b:
                y = yp1Method(f, x, y, h);

                yses.append(y);

                x += h;
        #

        return yses;
#

def eulerYP1(f, x, y, h):
        return y + h * f(x, y);
#

def heunYP1(f, x, y, h):
        k1 = h * f(x, y);
        k2 = h * f(x + h, y + k1);
        return y + 0.5 * (k1 + k2);
#

def rungeKuttaYP1(f, x, y, h):
        k1 = h * f(x, y);
        k2 = h * f(x + 0.5 * h, y + 0.5 * k1);
        k3 = h * f(x + 0.5 * h, y + 0.5 * k2);
        k4 = h * f(x + h, y + k3);

        return y + (1 / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4);
#


def euler(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, eulerYP1);
#

def heun(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, heunYP1);
#

def rungeKutta(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, rungeKuttaYP1);
#


def odeSystemSolver(fVector, a, b, y0Vector, h, yp1Methods):
        yVectorses = [];

        x = a;
        yVector = y0Vector;

        yVectorses.append(yVector);

        while x <= b:
                yp1Vector = yp1Methods(fVector, x, yVector, h);

                yVectorses.append(yp1Vector);

                yVector = yp1Vector;

                x += h;
        #

        return yVectorses;
#

def rungeKuttaYP1System(fVector, x, yVector, h):
        nF = len(fVector);

        k1Vector = [];
        yVectorPlusHalfK1Vector = [];

        for i in range(nF):
                k1El = h * fVector[i](x, yVector);
                k1Vector.append(k1El);
                yVectorPlusHalfK1Vector.append(yVector[i] + 0.5 *
                    k1El);

        k2Vector = [];
        yVectorPlusHalfK2Vector = [];

        for i in range(nF):
                k2El = h * fVector[i](x + 0.5 * h,
                    yVectorPlusHalfK1Vector);
                k2Vector.append(k2El);
                yVectorPlusHalfK2Vector.append(yVector[i] + 0.5 *
                    k2El);

        k3Vector = [];
        yVectorPlusK3Vector = [];

        for i in range(nF):
                k3El = h * fVector[i](x + 0.5 * h,
                    yVectorPlusHalfK2Vector);
                k3Vector.append(k3El);
                yVectorPlusK3Vector.append(yVector[i] + k3El);

        yp1Vector = [];

        for i in range(nF):
                k4El = h * fVector[i](x + h, yVectorPlusK3Vector);

                yp1 = yVector[i] + (1 / 6.0) * (k1Vector[i] + 2 *
                    k2Vector[i] + 2 * k3Vector[i] + k4El);

                yp1Vector.append(yp1);

        return yp1Vector;
#

def rungeKuttaSystem(fVector, a, b, y0Vector, h):
        return odeSystemSolver(fVector, a, b, y0Vector, h,
            rungeKuttaYP1System);
#
```

Listing 87: todo

X

6.2.2 b)

X

```
def fkFunc(f, x, y, h, yp1Func, q, pk, step):
        if step == 1:
                fh = yp1Func(f, x, y, h);
                fqh = yp1Func(f, x, y, q * h);
        else:
                fh = fkFunc(f, x, y, h, yp1Func, q, pk, step - 1);
                fqh = fkFunc(f, x, y, q * h, yp1Func, q, pk, step -
                     1);

        return fh + ((fh - fqh) / (pow(q, pk) - 1.0));
#

def eulerYP1Richardson(f, x, y, h):
        return fkFunc(f, x, y, h, eulerYP1, 5, 23, 5);
#

def heunYP1Richardson(f, x, y, h):
        return fkFunc(f, x, y, h, heunYP1, 2, 13, 5);
#

def rungeKuttaYP1Richardson(f, x, y, h):
        return fkFunc(f, x, y, h, rungeKuttaYP1, 4, 18, 5);
#

def eulerRichardson(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, eulerYP1Richardson);
#

def heunRichardson(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, heunYP1Richardson);
#

def rungeKuttaRichardson(f, a, b, y0, h):
        return odeSolver(f, a, b, y0, h, rungeKuttaYP1Richardson);
#
```

Listing 88: todo

X

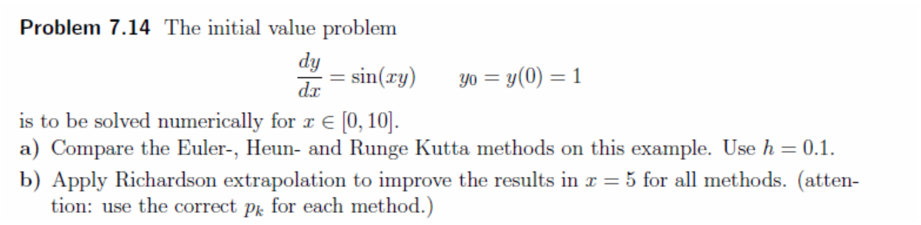
```

trueVals = [];

for i in range(8): trueVals.append(exT(i / 10.0, 0));

bestError = 9999;
bestQ = -1;
```

```
bestPK = -1;

for i in range(2, 6):
        for j in range(2, 25):
                qFU = i;
                pkFU = j;

                fns = heunRichardson(exT, 0, 0.6, 1, 0.1);

                error = 0;

                for k in range(8):
                        error += abs(trueVals[k] - fns[k]);

                if (error < bestError):
                        bestError = error;
                        bestQ = qFU;
                        bestPK = pkFU;
#

print(bestQ);
print(bestPK);
```

Listing 89: todo

X

6.3 Problem 7.14



Problem 7.14 The initial value problem

$$\frac{dy}{dx} = \sin(xy) \qquad y_0 = y(0) = 1$$

is to be solved numerically for $x \in [0, 10]$.

a) Compare the Euler-, Heun- and Runge Kutta methods on this example. Use $h = 0.1$.

b) Apply Richardson extrapolation to improve the results in $x = 5$ for all methods. (attention: use the correct p_k for each method.)

(iwo import functions from prob 7.13)

6.3.1 a)

X

```
def myF(x, y):
        return math.sin(x * y);

xses = [];

for i in range(102): xses.append(i / 10.0);
```

```
eulerYses = dif.euler(myF, 0, 10, 1, 0.1);
heunYses = dif.heun(myF, 0, 10, 1, 0.1);
rungeKuttaYses = dif.rungeKutta(myF, 0, 10, 1, 0.1);

plt.plot(xses, eulerYses);
plt.plot(xses, heunYses);
plt.plot(xses, rungeKuttaYses);
plt.xlabel("x");
plt.ylabel("y");
plt.show();
```

Listing 90: todo

results:

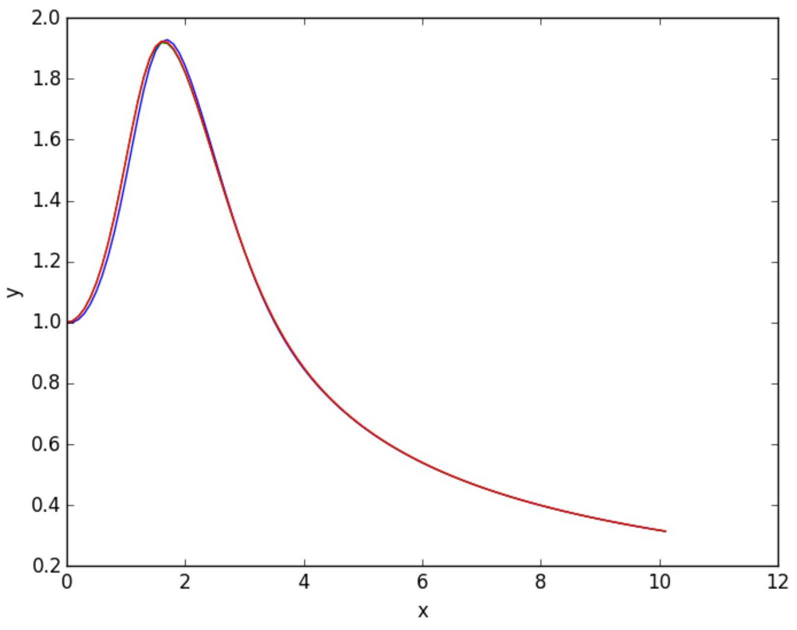


Figure 14: $\alpha = 2$, initially 3 sheep and 4 wolfs

6.3.2 b)

X

```
eulerRichardsonYses = dif.eulerRichardson(myF, 0, 10, 1, 0.1);
heunRichardsonYses = dif.heunRichardson(myF, 0, 10, 1, 0.1);
rungeKuttaRichardsonYses = dif.rungeKuttaRichardson(myF, 0, 10, 1,
    0.1);
```

```
print("euler: " + str(eulerRichardsonYses[50]));
print("heun: " + str(heunRichardsonYses[50]));
print("runge-kutta: " + str(rungeKuttaRichardsonYses[50]));
```

Listing 91: todo

results:

X

```
euler: 0.656662162824
heun: 0.657837825456
runge-kutta: 0.657541150858
```

Listing 92: Result of 1.1 a)

X

6.4 Problem 7.15

Problem 7.15 Apply the Runge Kutta method to the predator-prey example ?? and experiment with the parameter α and the initial values. Try to explain the population results biologically.

X

```
def sheepFunc10(t, yVector):
        return 10 * yVector[0] * (1 - yVector[1]);
#

def sheepFunc9(t, yVector):
        return 9 * yVector[0] * (1 - yVector[1]);
#

def sheepFunc12(t, yVector):
        return 12 * yVector[0] * (1 - yVector[1]);
#

def wolfsFunc(t, yVector):
        return yVector[1] * (yVector[0] - 1);
#

fses1 = [];
y0ses1 = [];

fses1.append(sheepFunc10);
fses1.append(wolfsFunc);
y0ses1.append(3);
y0ses1.append(1);

fses2 = [];
```

```
y0ses2 = [];

fses2.append(sheepFunc9);
fses2.append(wolfsFunc);
y0ses2.append(4);
y0ses2.append(2);

fses3 = [];
y0ses3 = [];

fses3.append(sheepFunc12);
fses3.append(wolfsFunc);
y0ses3.append(3);
y0ses3.append(2);

def testPredatorPrey(fses, y0ses):
        yVectorses = dif.rungeKuttaSystem(fses, 0, 5, y0ses, 0.05);

        xses = [];
        sheepYses = [];
        wolfsYses = [];

        x = 0;

        for yVector in yVectorses:
                sheepYses.append(yVector[0]);
                wolfsYses.append(yVector[1]);
                xses.append(x);
                x += 0.05;

        plt.plot(xses, sheepYses);
        plt.plot(xses, wolfsYses);
        plt.xlabel("x");
        plt.ylabel("y");
        plt.show();
#

testPredatorPrey(fses1, y0ses1);
testPredatorPrey(fses2, y0ses2);
testPredatorPrey(fses3, y0ses3);
```

Listing 93: todo

results:

6.5 Problem 7.16

X

```
def myF(x, y):
```

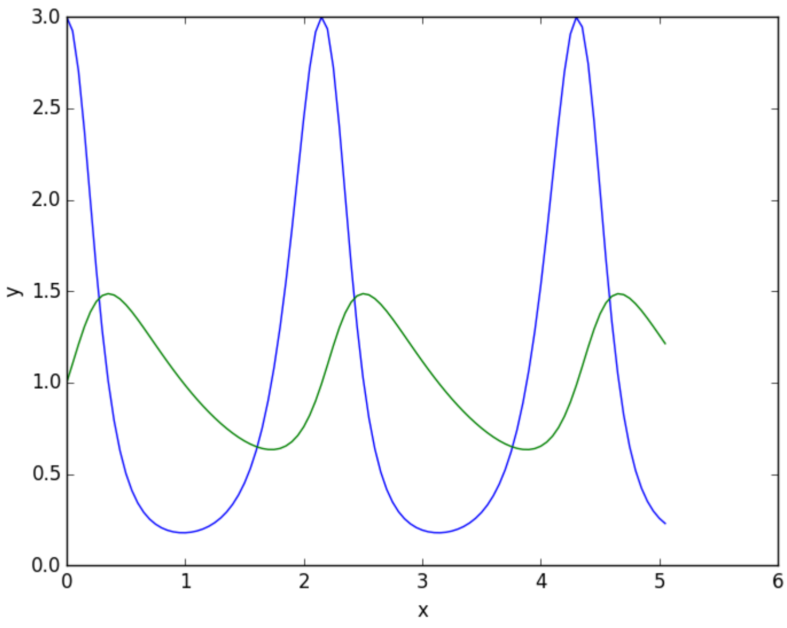


Figure 15: $\alpha = 2$, initially 3 sheep and 4 wolfs

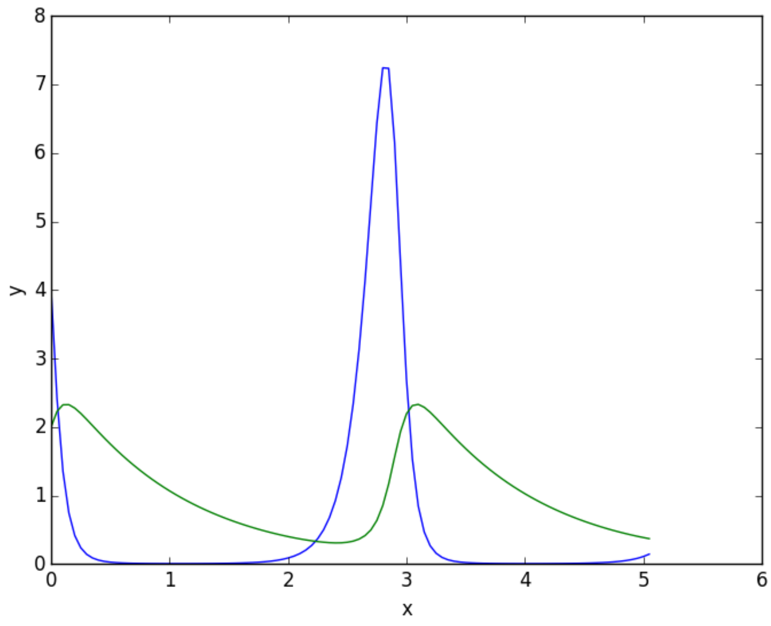


Figure 16: $\alpha = 2$, initially 3 sheep and 4 wolfs

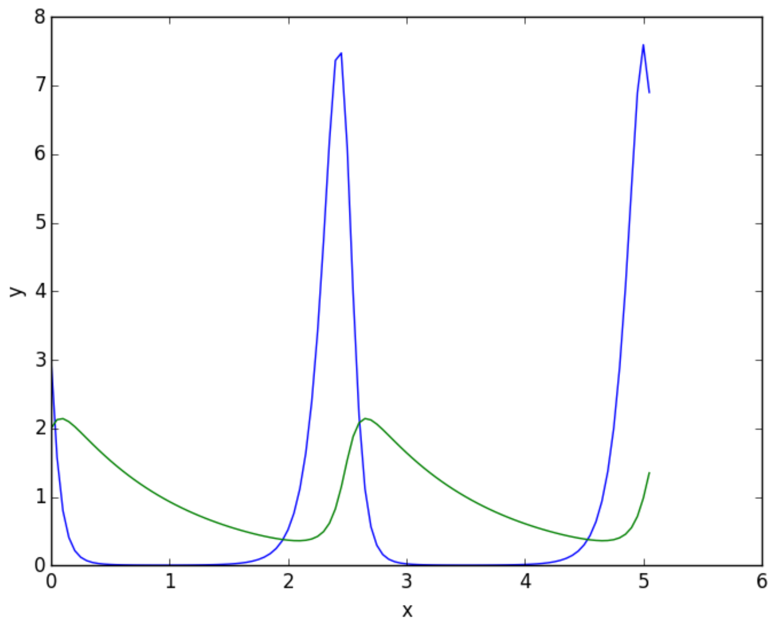
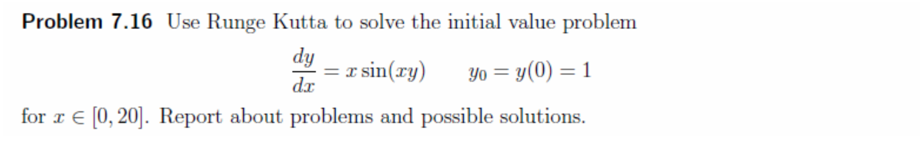


Figure 17: $\alpha = 2$, initially 3 sheep and 4 wolfs



Problem 7.16 Use Runge Kutta to solve the initial value problem

$$\frac{dy}{dx} = x\sin(xy) \qquad y_0 = y(0) = 1$$

for $x \in [0, 20]$. Report about problems and possible solutions.

```
        return x * math.sin(x * y);
#

xses = [];

for i in range(201): xses.append(i / 10.0);

yses = dif.rungeKutta(myF, 0, 20, 1, 0.1);

plt.plot(xses, yses);
plt.xlabel("x");
plt.ylabel("y");
plt.show();
```

Listing 94: todo

results:

X

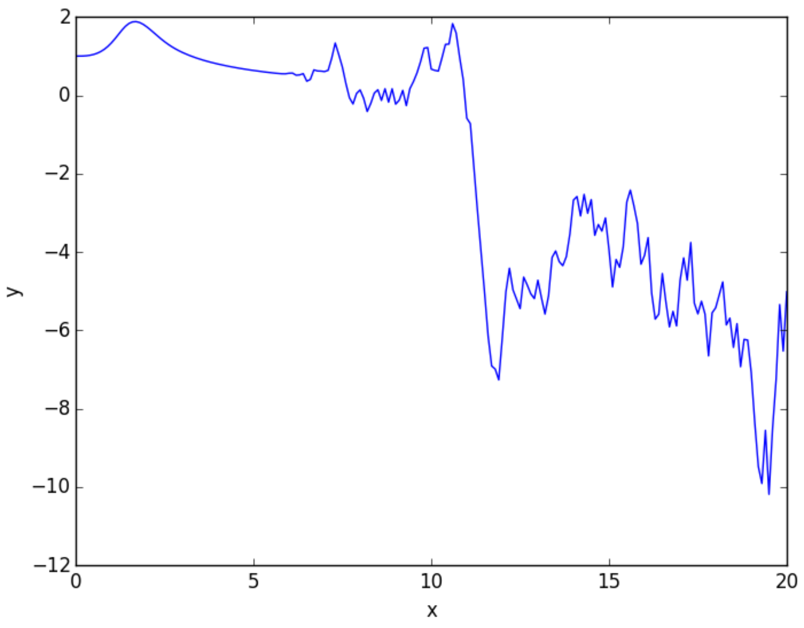


Figure 18: $\alpha = 2$, initially 3 sheep and 4 wolfs