Advanced Mathematics for Engineers - Laboratory Problems

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

January 28, 2017

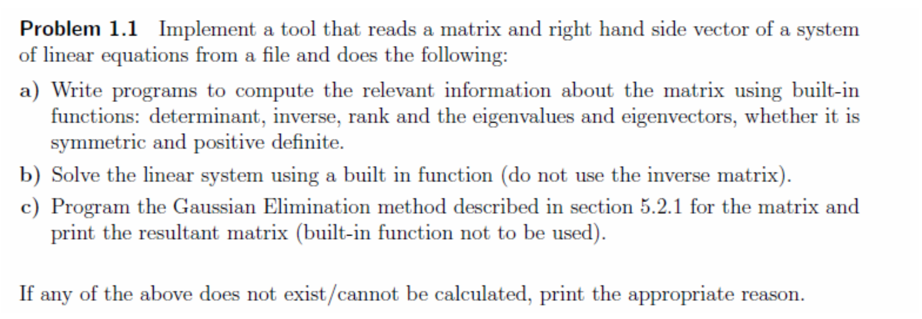
1 Linear Algebra

First of all, a function to properly print vectors and matrices could come in handy. This can be realized the following way:

```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: Functions 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.

To read a matrix and a vector from a text file, first a form has to be determined, for example like this:

```
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;
```

The first part is the matrix, where each line represents one row and each element within that row is seperated by a space. After a line of at least one hashtag, the following values are the elements of the vector, again seperated by a single space. If that form is adhered to, the matrix and vector can be read from the text file with the following script:

```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: Extracting the matrix and vector from the text file

Afterwards, the matrix and vector from the text file can be used as Python arrays, where the vector is a one-dimensional array and the matrix a two-dimensional array, with the single rows in the first dimension.

1.1.1 a)

For the built-in functions the Python library *numpy* can be used:

```
import numpy as np;

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: Problem 1.1 a)

The results are:

```
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)

Since the matrix in the text file is square all calculations could be done.

1.1.2 b)

In order to solve the linear system, it has to be check first whether number of columns in the matrix equals the number of rows in the vector. If so, a built-in function from *numpy* can be used to solve the system:

```
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: Problem 1.1 b)

The resulting x values are:

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

```
x4 = -1.13439716312
```

Listing 6: Result of 1.1 b)

1.1.3 c)

The Gaussian Elemination can be realized without an built-in function in the following way:

```
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);
```

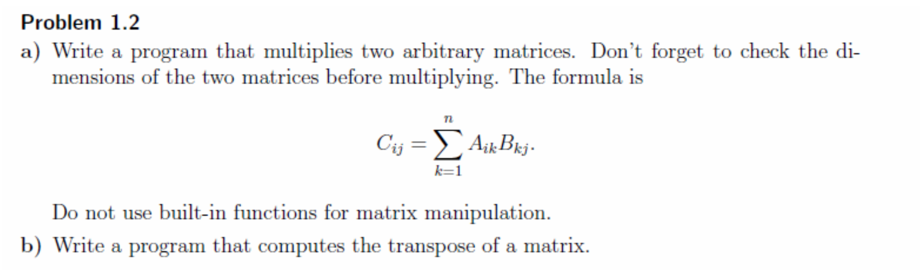
Listing 7: Problem 1.1 c)

Applying this function to the matrix and vector from the text file leads to the following results:

```
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.

1.2.1 a)

Multiplying two matrices can be done with the following script:

```
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: Problem 1.2 a)

The result of the multiplication of the two matrices defined in the script is:

```
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)

Transposing a matrix can be done easily with an own function as well:

```
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: Problem 1.2 b)

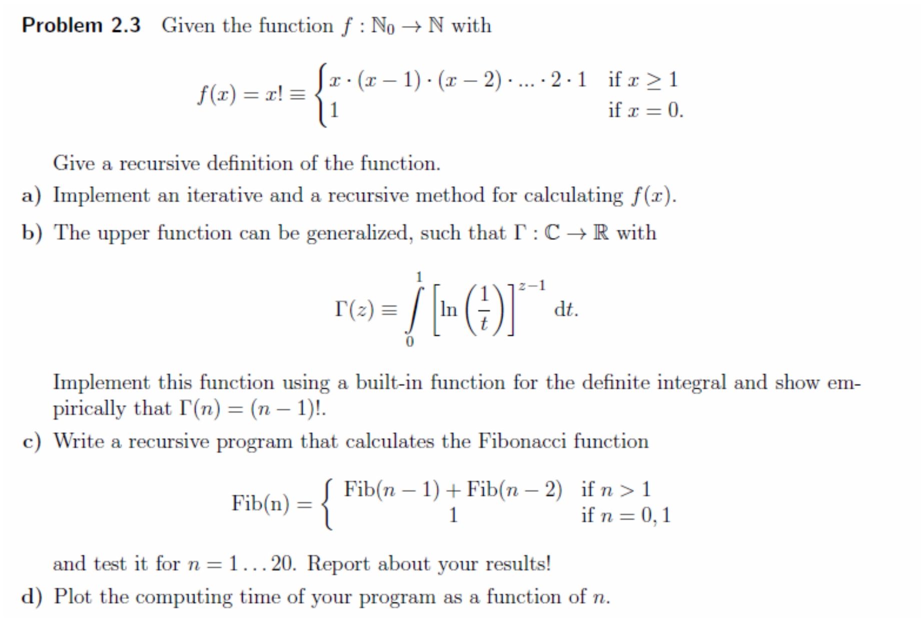
The result is:

```
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)

```
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: Problem 2.3 a)

The results of both factorial functions are:

```
n = 9
iterative result: 362880
recursive result: 362880
```

Listing 14: Result of 2.3 a)

2.1.2 b)

The integral can be calculated with a built-in function from the scipy library.

```
import scipy.integrate as itg;

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

for i in range(1, 7):
        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: Problem 2.3 b)

The results are:

```
n = 1
(n - 1)! = 1
Gamma(n) = (1.0, 1.1102230246251565e-14)

n = 2
(n - 1)! = 1
Gamma(n) = (1.0000000000000004, 1.6653345369377348e-15)

n = 3
(n - 1)! = 2
Gamma(n) = (2.0000000000000018, 1.687538997430238e-14)

n = 4
(n - 1)! = 6
Gamma(n) = (6.000000000000022, 2.9398705692074145e-13)

n = 5
(n - 1)! = 24
Gamma(n) = (24.00000000000007, 3.4425795547576854e-12)

n = 6
(n - 1)! = 120
Gamma(n) = (120.00000000002257, 2.0904167286062147e-10)
```

Listing 16: Result of 2.3 b)

The values of the gamma function are not exact because the quad function only calculates an approximation. The second value that's returned is an estimation of the error.

2.1.3 c)

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

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

for i in range(1, 21):
        fib = fibonacciFunc(i);
        print("fib(" + str(i) + ") = " + str(fib));
```

Listing 17: Problem 2.3 c)

The calculated elements of the Fibonacci sequence are:

```
fib(1) = 1
fib(2) = 2
fib(3) = 3
fib(4) = 5
fib(5) = 8
fib(6) = 13
fib(7) = 21
fib(8) = 34
fib(9) = 55
fib(10) = 89
fib(11) = 144
fib(12) = 233
fib(13) = 377
fib(14) = 610
fib(15) = 987
fib(16) = 1597
fib(17) = 2584
fib(18) = 4181
fib(19) = 6765
fib(20) = 10946
```

Listing 18: Result of 2.3 c)

As it can be seen, the Fibonacci function seems to grow exponentially.

2.1.4 d)

```
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: Problem 2.3 d)

The graph of the runtime of the Fibonacci sequence looks like this:

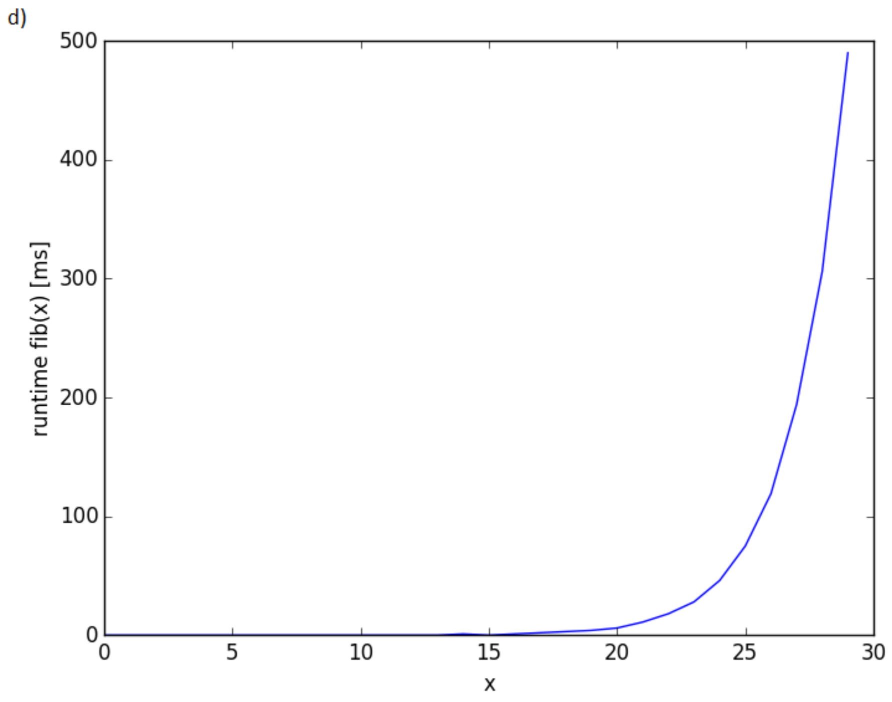
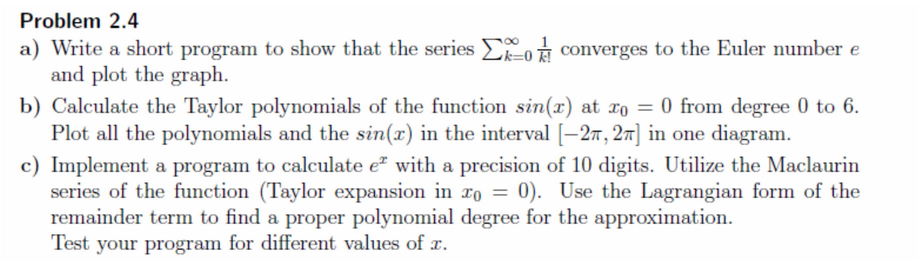


Figure 1: Runtime of the Fibonacci function

As well as the y values of the Fibonacci function, its runtime seems to grow exponentially too.

2.2 Problem 2.4



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)

To graphically show that the series converges to the Euler number e, the series could be defined as a function $f(x)$ where x defines the number of iterations of the sum, like $f(x) = \sum_{k=0}^{\lfloor x \rfloor} \frac{1}{k!}$.

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

    return x * factorialFunc(x - 1);

def aFunc(x):
        sum = 0
        for i in range(x):
                sum += (1.0 / factorialFunc(i));
        return sum;

results = [];
eulers = [];

for i in range(10):
        results.append(aFunc(i));
        eulers.append(math.e);

plt.plot(results);
plt.plot(eulers);
plt.xlabel('x');
plt.ylabel('y');
plt.show();
```

Listing 20: Problem 2.4 a)

The graph of the function with a line for e looks like this:

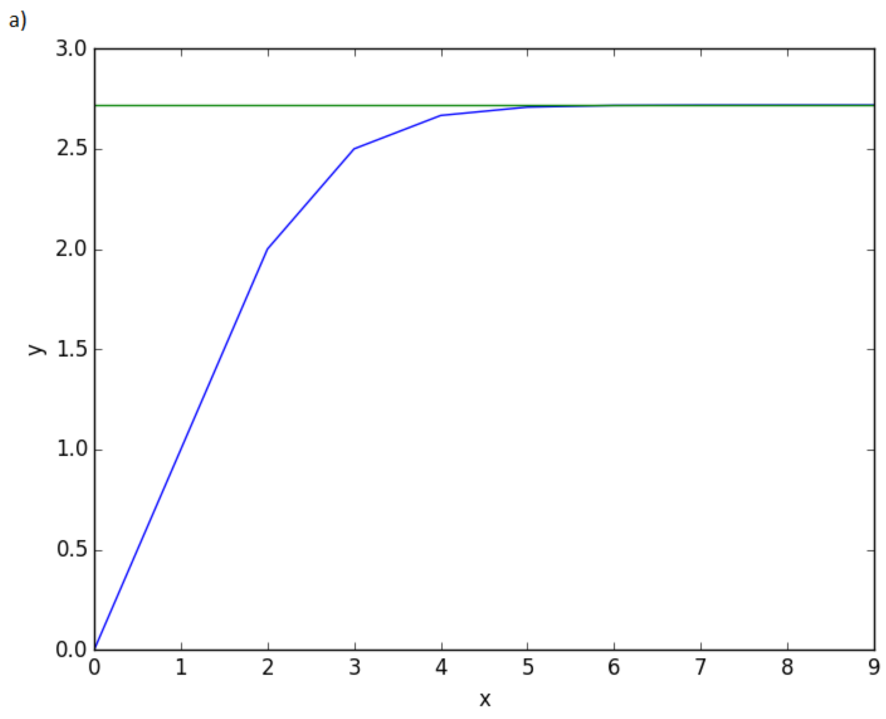


Figure 2: Graphs of both functions

As it can be seen, after $x = 5$ there's hardly any difference visible in the graph.

2.2.2 b)

```
def bFunc(degree, x):
        sum = x;

        if (degree >= 3): sum -= (1.0 / 6.0) * pow(x, 3);
        if (degree >= 5): sum += (1.0 / 120.0) * pow(x, 5);

        return sum;

def bFunc2(degree):
        nSteps = 40;
        xStep = 4 * math.pi / (nSteps - 1);
        x = math.pi * -2;

        xses = [];
        results = [];

        for i in range(nSteps):
                xses.append(x);
                results.append(bFunc(degree, x));
                x += xStep;
        return [xses, results];

for i in range(0, 7):
        taylorFunc = bFunc2(i);
        plt.plot(taylorFunc[0], taylorFunc[1]);

plt.xlabel('x');
plt.ylabel('y');
plt.show();
```

Listing 21: Problem 2.4 b)

The graph of the polynomials looks like the following:

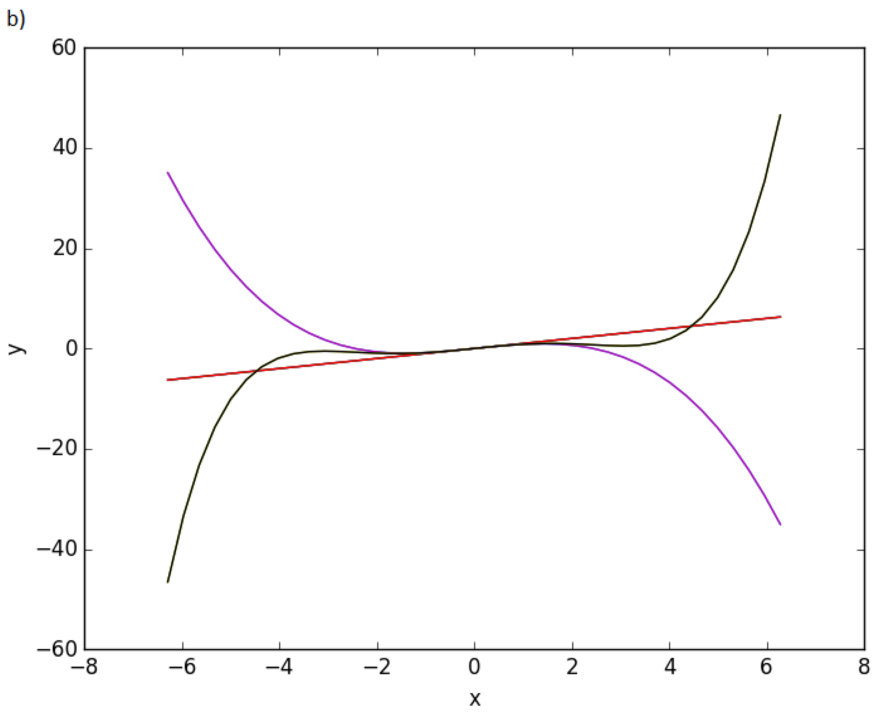


Figure 3: Graphs of all Taylor series

Since $sin^{(2n)}(0) = 0$ for $n \in \mathbb{N}$, the respective term won't show up in the Taylor polynomial. Therefore, the Taylor polynomial for degree $(2n-1)$ is the same as for degree $(2n)$ (with $n \in \mathbb{N}$). Because of that, only three different polynomials can be seen in the graph.

2.2.3 c)

```
ef eulerFunc(d, x):
```

```python
    if (d <= 0): return 1;

    sum = 1;

    for i in range(1, d + 1):
        sum += pow(x, i) / (factorialFunc(i) + 0.0);

    return sum;

def getRemainder(n, x):
    xabs = abs(x);

    return (math.exp(xabs) * pow(xabs, n + 1)) / float(
        factorialFunc(n + 1));

def getDegree(x):
    n = 0;

    while n < 100:
        remainder = getRemainder(n, x);

        if (remainder < 1e-11): return n;

        n = n + 1;

    return -1;

for x in range(1, 6):
    deg = getDegree(x);
    eul = eulerFunc(deg, x);

    print("x = " + str(x) + " | degree = " + str(deg) + " | e("
        + str(x) + ") = " + str(eul));
```

Listing 22: Problem 2.4 c)

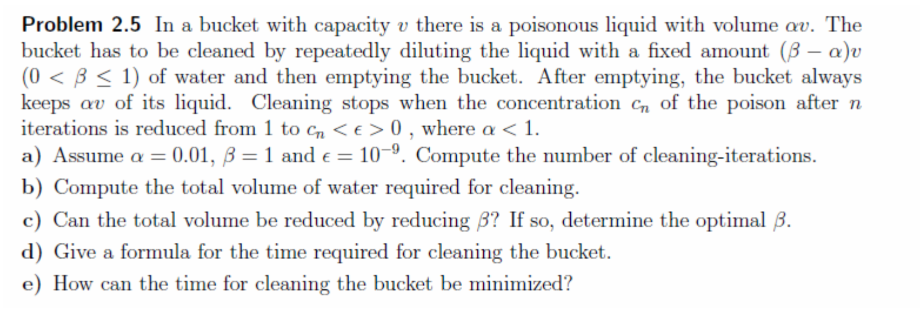
The resulting approximations as well as the necessary degree for the approximation to be as precise as wanted are the following:

```
x = 1 | degree = 14 | e(1) = 2.71828182846
x = 2 | degree = 19 | e(2) = 7.38905609893
x = 3 | degree = 23 | e(3) = 20.0855369232
x = 4 | degree = 28 | e(4) = 54.5981500331
x = 5 | degree = 32 | e(5) = 148.413159103
```

Listing 23: Result of 2.4 c)

As it can be seen, with every step of x the necessary degree has to increase by about 5.

2.3 Problem 2.5



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?

2.3.1 a)

With each cleaning iteration the concentration is multiplied by $\frac{\alpha}{\alpha+(\beta-\alpha)}$, which can be simplified to $\frac{\alpha}{\beta}$. The bucket can be defined as a class, with α, β and ϵ as attributes and the cleaning algorithm as function:

```
class Bucket:
        poison = 1;
        cleaningIterations = 0;
        waterUsed = 0;

        def __init__(self, alpha, beta, eps):
                self.alpha = alpha;
                self.beta = beta;
                self.eps = eps;

        def doOneClean(self):
                water = self.beta - self.alpha;
                self.waterUsed += water;

                self.poison = (self.alpha / float(self.beta)) *
                    self.poison;

        def clean(self):
                while (self.poison >= self.eps):
                        self.cleaningIterations += 1;
                        self.doOneClean();

bu = Bucket(0.01, 1, 10e-9);
bu.clean();
```

Listing 24: Problem 2.5 a)

The result is the following:

```
number of iterations: 5
```

Listing 25: Result of 2.5 a)

2.3.2 b)

The necessary adjustments to the class are already in the code fragment of the subtask a) (lines 4 and 13). The result of the calculation is:

```
water used: 4.95 * v
```

Listing 26: Result of 2.5 b)

2.3.3 c)

To calculate how much water is needed to reduce the poison to the required concentration, first a formula to calculate the number of required iterations is needed. The concentration after n iterations is $(\frac{\alpha}{\beta})^n$, which means the number of iterations to reach the required concentration ϵ is $\lceil log_{\frac{\alpha}{\beta}}(\epsilon)\rceil$.
The amount of water which is used per iteration is $(\beta - \alpha) * v$. Therefore, the amount of water required is $\lceil log_{\frac{\alpha}{\beta}}(\epsilon)\rceil * (\beta - \alpha) * v$.
To find out the optimal β with $\alpha = 0.01$ and $\epsilon = 10^{-9}$, the calculation for the water that is needed can be defined as a function $f(\beta) = \lceil log_{\frac{0.01}{\beta}}(10^{-9})\rceil * (\beta - 0.01)$ with $\beta \in (0, 1]$. Looking at the graph of this function shows that the lowest y value is approaching 0, however the exercise description stated that the water is dilluted with $((\beta - \alpha) * v)$ of water, so the value of β has to be greater than α. Therefore the optimal β in terms of water needed is the lowest possible value that is greater than α.

2.3.4 d)

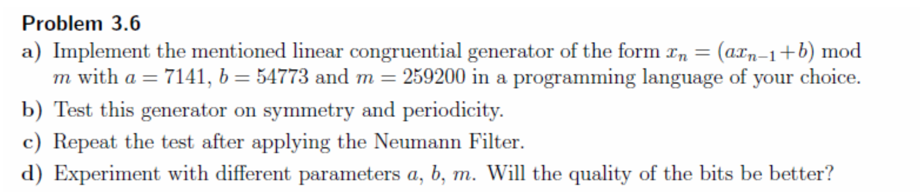
As already discovered in the previous subtask, the algorithm to determine the number of iterations is $\lceil log_{\frac{\alpha}{\beta}}(\epsilon)\rceil$. The cleaning time is the result of this calculation times the amount time required for one iteration.

2.3.5 e)

The number of iterations, which is proportional to the cleaning time, get smaller the smaller $\frac{\alpha}{\beta}$ is. Therefore, with α as small and β as big as possible the time for cleaning the bucket can be minimized. Assuming that α can't be adjusted the best possible value for β would be 1.

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?

3.1.1 a)

```python
class RNG:
    def __init__(self, pA = 7141, pB = 54773, pM = 259200):
        self.bits = [];
        self.x = 1;
        self.a = pA;
        self.b = pB;
        self.m = pM;

    def printParameters(self):
        print("a = " + str(self.a) + ", b = " + str(self.b) + ", m = " + str(self.m));

    def getRandomNumber(self):
        self.x = (self.a * self.x + self.b) % self.m;
        return self.x;

    def addNumbers(self, number):
        binNumber = bin(number)[2:];

        for bitNumber in binNumber:
            self.bits.append(int(bitNumber));

    def fillNumbers(self, iterations):
        for i in range(iterations):
            self.addNumbers(self.getRandomNumber());
```

Listing 27: Problem 3.6 a)

3.1.2 b)

To test for symmetry and periodicity, new functions can be added to the RNG class from a).

```python
    def makeSymmetryTest(self):
        sum = 0;

        for b in self.bits:
            sum += b;

        deviation = abs(0.5 - (sum / float(len(self.bits))));

        print("symmetry␣test:")
        print("deviation␣=␣" + str(deviation));

    def isPeriod(self, periodLength):
        nBits = len(self.bits);

        if periodLength * 2 > nBits: return False;

        iterations = int(math.floor(nBits / float(periodLength)));

        for i in range(periodLength):
            compareElement = self.bits[i];

            for j in range(1, iterations):
                key = j * periodLength + i;

                if key >= nBits: break;

                element = self.bits[key];

                if element != compareElement:
                    return False;

        return True;

    def makePeriodicityTest(self):
        periodLength = -1;

        maximumPeriod = self.m * int(math.ceil(math.log(self.m, 2)));

        for i in range(1, maximumPeriod):
            if self.isPeriod(i):
                periodLength = i;
                break;

        print("periodicity␣test:");

        if periodLength < -0.5:
            print("no␣period␣found.");
        else:
```

```
                        print("period length = " + str(periodLength
                            ));
```

Listing 28: Problem 3.6 b)

Running these tests with the values $a = 7141$, $b = 54773$ and $m = 259200$ lead to the following results.

```
symmetry test:
deviation = 0.0266565772348

perodicity test:
no period found.
```

Listing 29: Result of 3.6 b)

Note: The periodicty test had been aborted prematurely due to excessive run times of the program. With the large value m and the conversion of the numbers into binary the length of the period cannot be calculated within reasonable time.

3.1.3 c)

Applying the Neumann Filter can also be realized as an additional function for the RNG class:

```
        def applyNeumannFilter(self):
                tempBits = self.bits;
                self.bits = [];

                nIterations = int(math.floor(len(tempBits) / 2.0));

                for i in range(nIterations):
                        bit1 = tempBits[i * 2];
                        bit2 = tempBits[i * 2 + 1];

                        if (bit1 == 0 and bit2 == 1):
                                self.bits.append(0);
                        elif (bit1 == 1 and bit2 == 0):
                                self.bits.append(1);
```

Listing 30: Problem 3.6 c)

The results with the same values of b) after applying the Neumann Filter are:

```
symmetry test:
deviation = 0.000162239945826

perodicity test:
no period found.
```

Listing 31: Result of 3.6 c)

Again, the periodicity test has been aborted prematurely.

3.1.4 d)

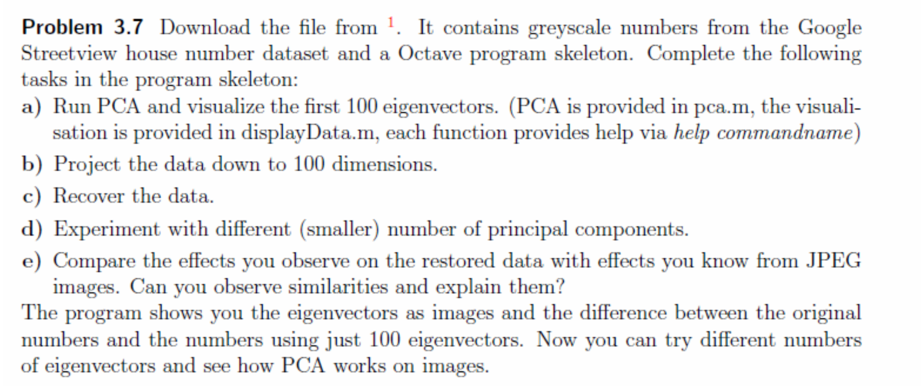
With the RNG class the values of a, b and m can easily be changed for further experiments:

```
a = 1, b = 1, m = 20
symmetry test:
deviation = 0.0769230769231

a = 20, b = 30, m = 100000
symmetry test:
deviation = 0.0624828117265

a = 214013, b = 2531011, m = 4294967296
symmetry test:
deviation = 0.0168803363686
```

Listing 32: Result of 3.6 d)

Looking at the results the deviation tends to get smaller the higher the value of m is. However, even with a value as little as 20 the deviation is somewhat reasonable.

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.

The first 100 numbers of the data looks like this:

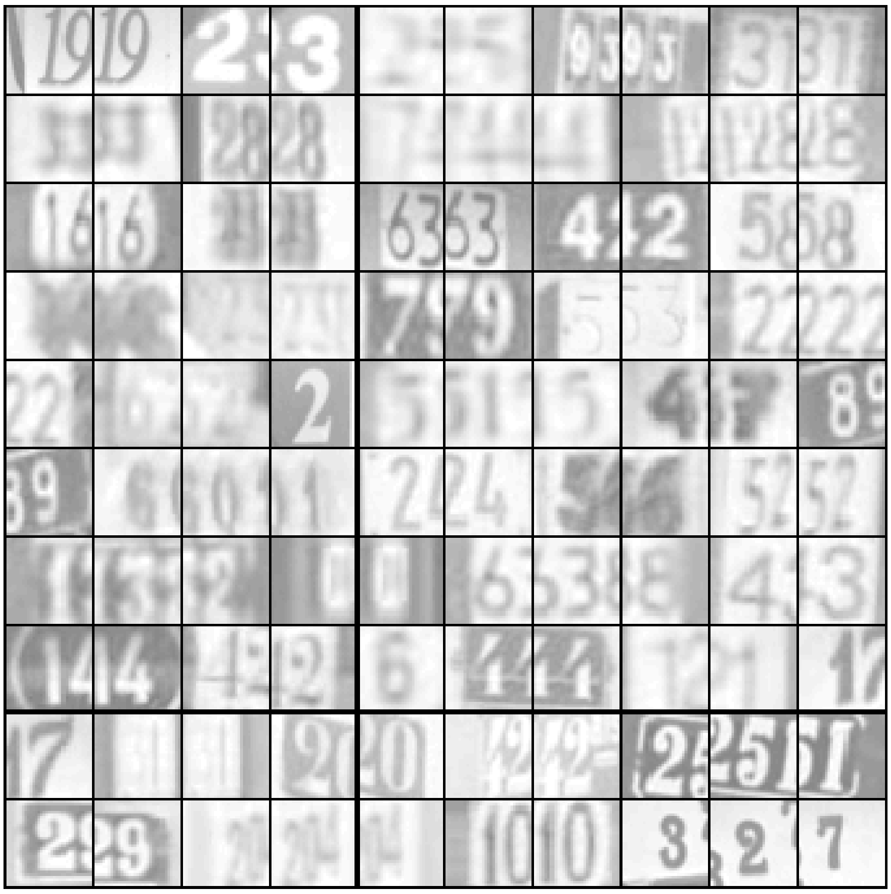


Figure 4: The first 100 numbers of the data

3.2.1 a)

The top 100 eigenvectors can be dispayed the following way:

```
[U, S] = pca(X_norm);
eigenVectors = U(:,1:100);

displayData(eigenVectors');
```

Listing 33: Problem 3.7 a)

The resulting visualization is:

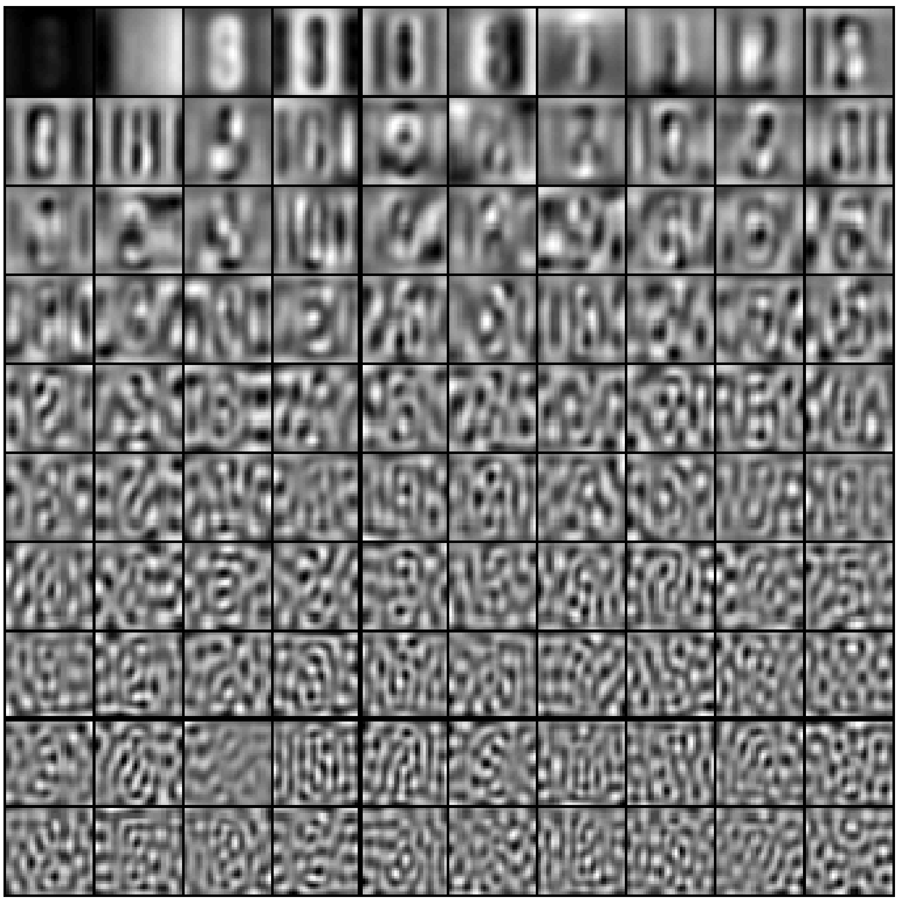


Figure 5: The top 100 eigenvectors

3.2.2 b)

The data can be projected down to 100 dimensions in the following way:

```
Z = X_norm * eigenVectors;
```

Listing 34: Problem 3.7 b)

3.2.3 c)

Recovering the data:

```
recoveredData = Z * eigenVectors';

displayData(X_norm(1:100,:));
displayData(recoveredData(1:100,:));
```

Listing 35: Problem 3.7 c)

The recovered numbers look like this:

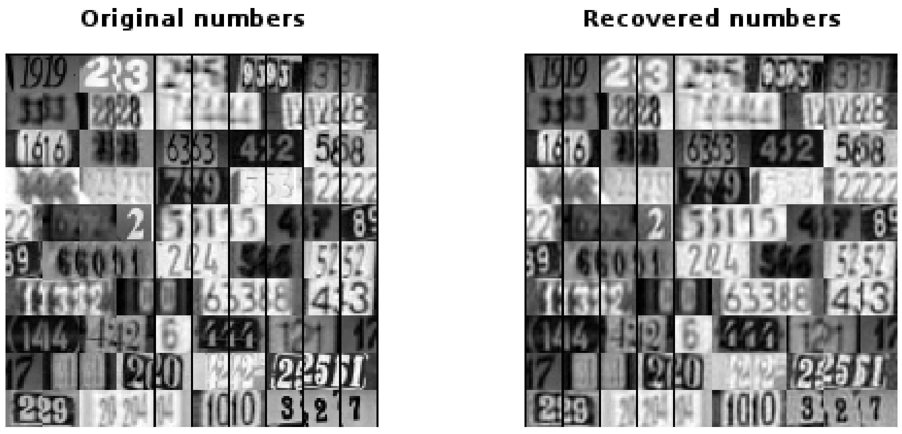


Figure 6: The data using 100 principal components

3.2.4 d)

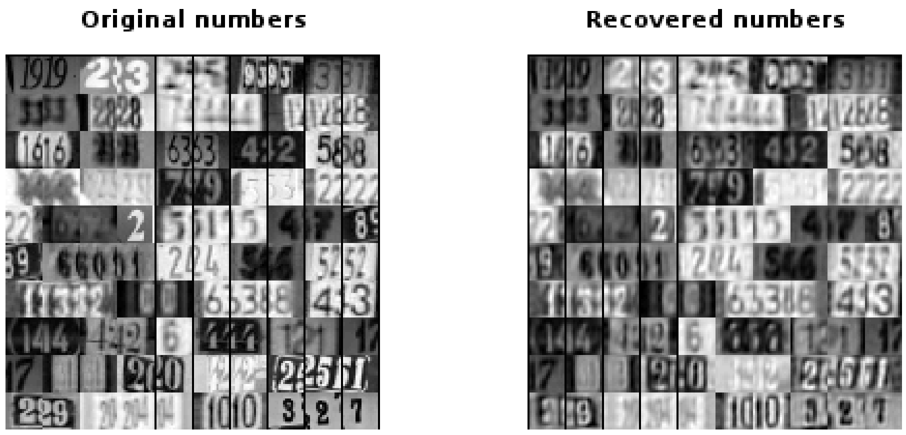


Figure 7: The data using 50 principal components

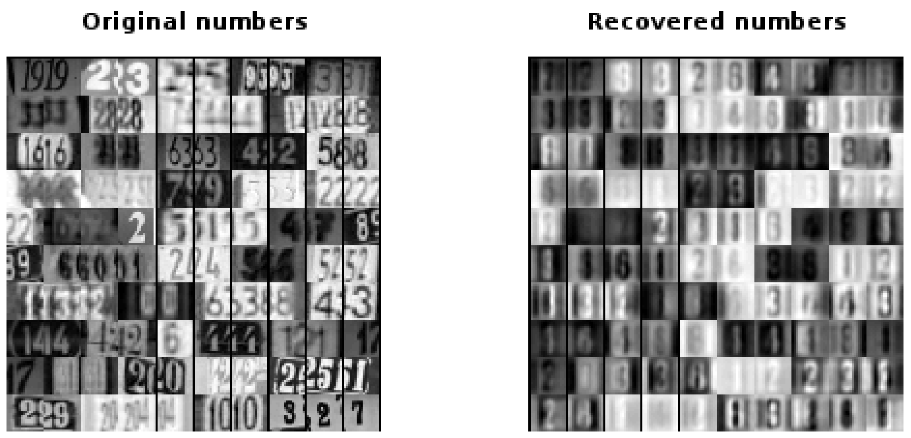


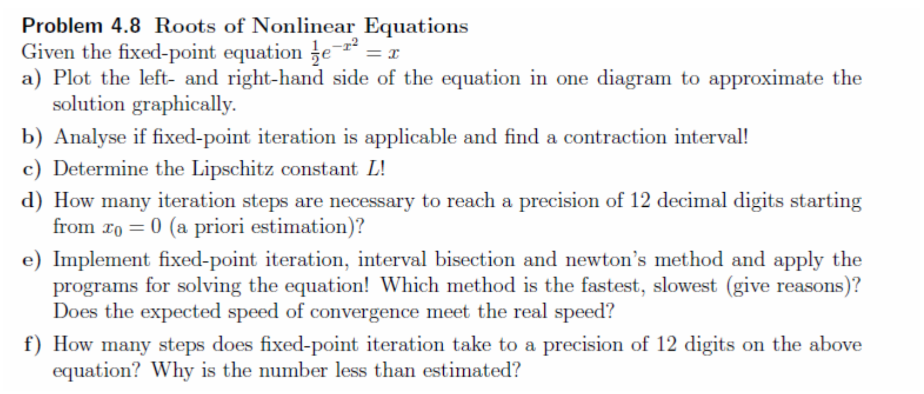
Figure 8: The data using 10 principal components

3.2.5 e)

By reducing the number of dimensions the detail gets lost in a similar way like the JPEG compression. This can be observed especially between two areas with a high color contrast.

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?

First the function needs to be defined. The're are two ways to find a root, setting function equal to 0 and setting the function equal to x. The function which is to be set equal to 0 is going to be called $f(x)$, while the function to be set equal to x is going to be called $fFixed(x)$, with their derived functions $fDerived(x)$ and $fFixedDerived(x)$ respectively:

```
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 36: Different functions for the equation

These functions will be used in the code of the following subtasks, where method to find a root determines which of them is going to be used.

4.1.1 a)

With the following code both sides of the equation will be plotted:

```
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 37: Problem 4.8 a)

The resulting plot of both sides of the equation looks like the following:

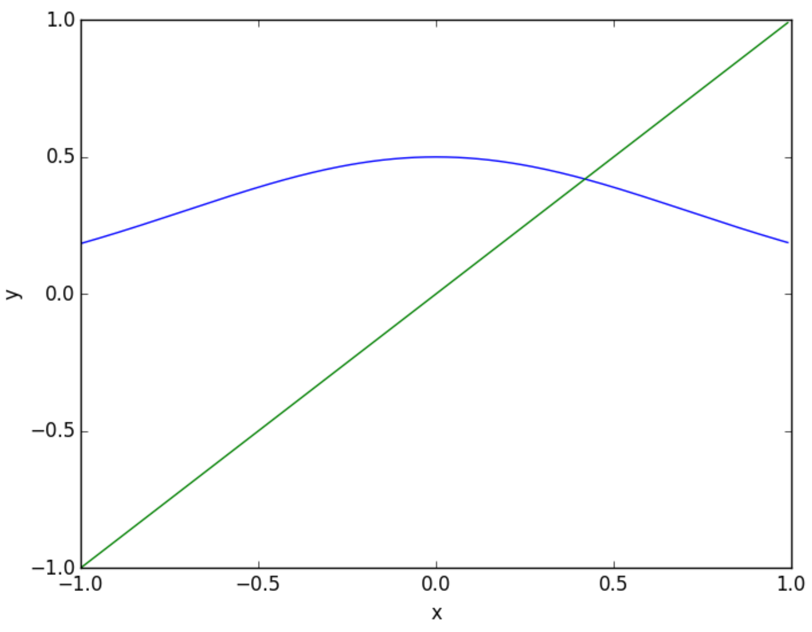


Figure 9: Graphical solution for the root

The approximate solution can be found graphically by looking for an intersection of both functions. The functions intersect close to the point (0.45 | 0.45), therefore the root of the function is approximately 0.45.

4.1.2 b)

To use the fixed point interation method for finding a root of a function within an interval $[a, b] \to [a, b]$, the function has to be a contraction within this interval. A function is a contraction if the Lipschitz constant L is greater than 0 and smaller than 1. Within the interval $[0, 1] \rightarrow [0, 1]$, where we graphically estimated the root in the sub task a), the Lipschitz constant is about 0.4289 (see sub task c)), therefore the condition holds and fixed point interation is applicable.

4.1.3 c)

The Lipschitz constant is the highest increase or decrease of a function, or in other words the maximum of the absolute value of the minimum and maximum of the first derivative of said function. The second derivative of the function has two roots, at $x_1 = -0.70711$ and $x_2 = 0.70711$. These are the maxima and minima of the first derivative, and therefore the highest increase of the function and the Lipschitz constant L. Since only x_2 is in the contraction interval, the Lipschitz constant can simply be calculated by plugging it into the function f.

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

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

Listing 38: Problem 4.8 c)

The resulting L is:

```
L = 0.428881942471
```

Listing 39: Result of 4.8 c)

4.1.4 d)

With the Lipschitz constant from c), the maximum number of iterations required to find a root with a given precision by using the fixed point iteration method can be calculated with the a priori estimation:

```
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 40: Problem 4.8 d)

The result of this estimation is:

```
33 iteration steps required
```

Listing 41: Result of 4.8 d)

Therefore, to get a precision of 12 decimal digits for our function f a maximum of 33 iterations are needed.

4.1.5 e)

First we need a function to see whether our calculated value is precise enough:

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

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

Listing 42: Determine if an approximation is precise enough

The implementation of fixed point iteration could be realized in the following way:

```python
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;
```

Listing 43: Fixed point iteration method

The interval bisection method can be implemented like this:

```python
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;
```

Listing 44: Interval Bisection method

Finally a function to approximate the root of a function with Newton's method could be realized the following way:

```
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;
```

Listing 45: Newton's method

Afterwards, the three methods can be compared regarding how much time is needed to find the root of the function with the desired precision.

```
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);

# get a precise enough approximation of the root
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("newtons 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("newtons method:");
findRoot(2, fP33, precision, 0, -1);
```

Listing 46: Testing the three methods

The results of the tests are the following:

```
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

newtons 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

newtons 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

newtons method:
steps taken: 5
average time taken: 0.0377 milliseconds
```

Listing 47: Result of 4.8 e)

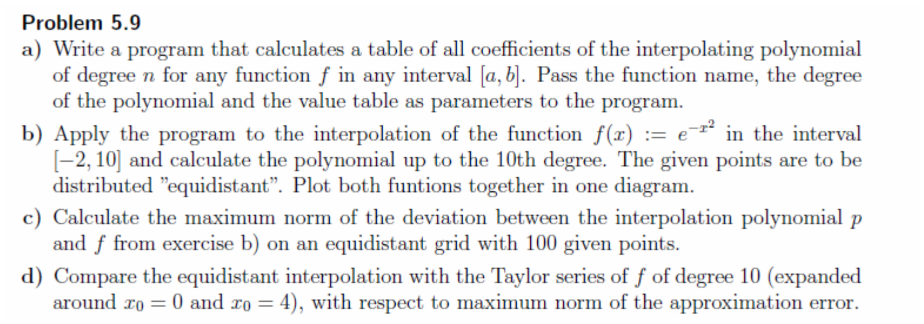
As seen Newton's method is much faster than the other two methods, with up to almost a tenth of computational time. The second fastest method is fixed point iteration, while the interval bisection method is the slowest.
The reason why Newton's method is the fastest is because it also utilizes the first derivative of the function, which is a valuable information to find the root quicker. Interval bisection is the slowest method because it doesn't even use the function value unlike fixed point iteration. Instead, the interval is halved halved with each step, no matter how close one bound is to the root of the function (except when the approximation is already precise enough of course).

4.1.6 f)

To get a precise enough approximation of the root, 27 steps were needed instead of the estimated 33 steps in d). The reason it took less iterations is 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)

To solve the linear equation for the coefficients the *numpy* library can be used again.

```
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 48: Problem 5.9 a)

This function will be used in the following subtasks.

5.1.2 b)

To use the coefficients as a function, a python function for the calculation of the y values has to be defined as well:

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

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

	return sum;
```

Listing 49: Function which uses the calculated coefficients

Afterwards, the function values for the coefficients can be calculated and the result plotted.

```python
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 50: Problem 5.9 b)

The resulting graph looks like the following:

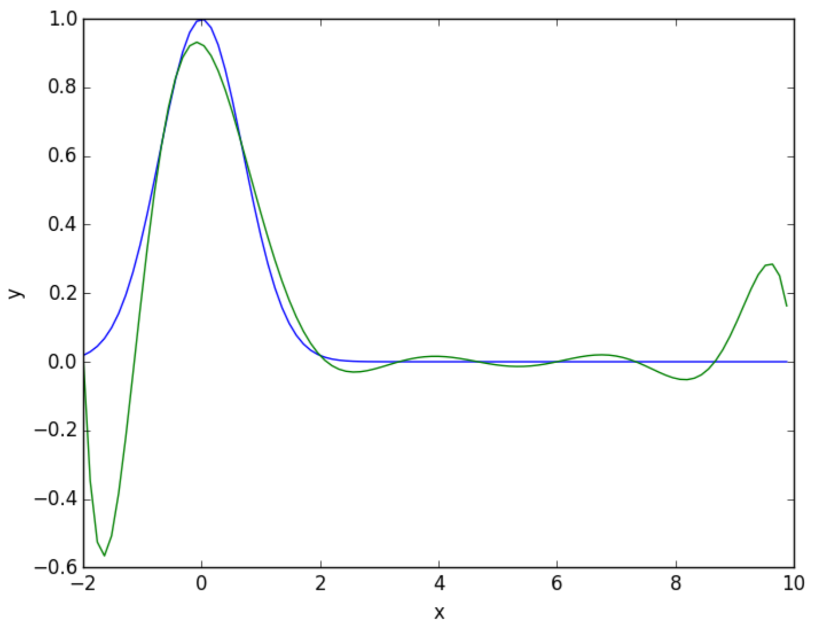


Figure 10: Plot of both functions

The blue curve represents the real values, while the green curve represents the graph of the polynomial approximation. Within the interval $[-1, 8]$ the deviations to be relatively small, outside of the interval however there seem to be bigger differences between both functions.

5.1.3 c)

```
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 51: Problem 5.9 c)

The results of the calculation of the maximum deviation are:

```
maximum deviation: 0.634025332357
```

Listing 52: Result of 5.9 c)

5.1.4 d)

First the functions to get the y values for both Taylor series have to be defined:

```
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));
```

Listing 53: Definition of the Taylor functions

Afterwards the deviation of both functions to the original function can be calculated:

```
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 54: Problem 5.9 d)

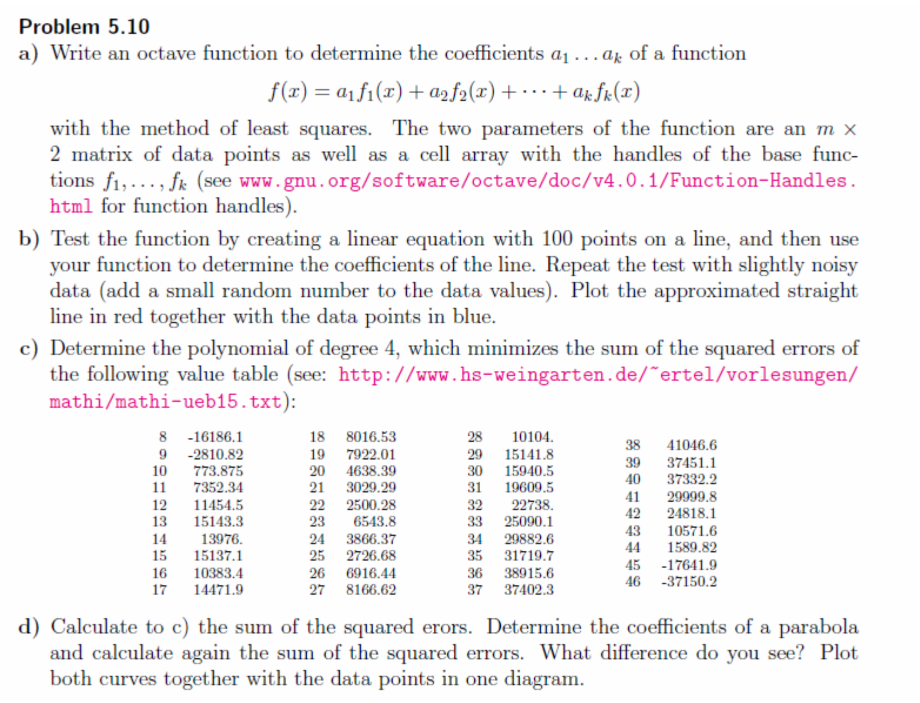
The results of the calculation are:

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

Listing 55: Result of 5.9 d)

As it can be easily seen, the Taylor series expanded around $x_0 = 0$ has a much bigger deviation than the series expanded around $x_0 = 4$. The reason for that is that the y value of a Taylor series tends to deviate stronger as further the respective x is from the expantion point x_0. With $x_0 = 4$ the maximum Δx ($\Delta x = |x - x_0|$) is 6, which is as little as possible since 4 is in the middle of the interval $[-2, 10]$. With $x_0 = 0$ however the maximum Δx is 10, therefore the Taylor series expanded there is deviating much more.

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)

```
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 56: Problem 5.10 a)

5.2.2 b)

Wth the function from a), the linear equations can be easily approximated by using two base functions:

```
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 57: Problem 5.10 b)

The coefficients calculated from the random data points are the following:

```
coefficients:
[ 0.64    0.23 ]

approximated coefficients:
[ 0.63139257    2.33442881 ]
```

Listing 58: Result of 5.10 b)

As it can be seen, the approximated coefficients deviate very little from the actual linear equation.
The exact linear equation and the data points are visualized in the following graph:

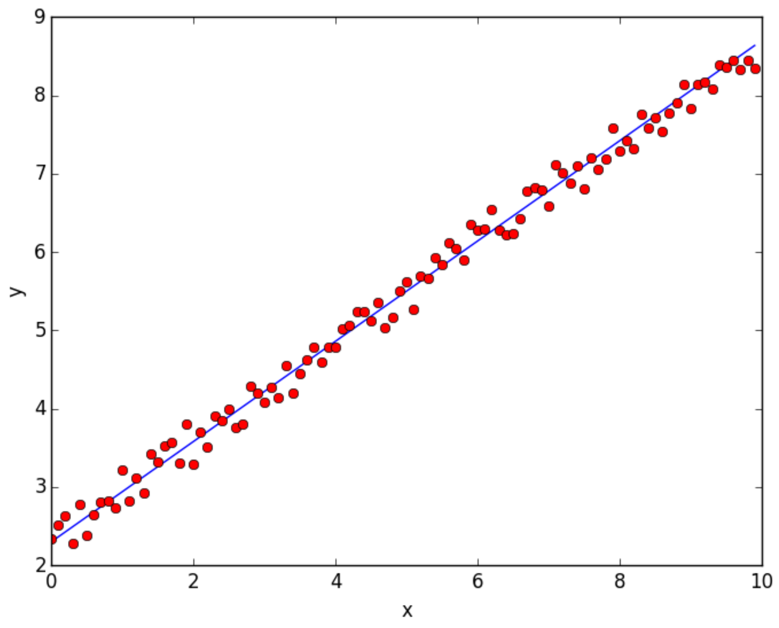


Figure 11: The linear equation as well as the data points

5.2.3 c)

Using the function from a), new data points and new base function, a polynomial of degree 4 (as well as degree 2 for the subtask d)) can be easily created:

```
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]);
# [...] (I left out the other values in the paper to save space)
txtPoints.append([45, -17641.9]);
txtPoints.append([46, -37150.2]);

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

txtPointsXses = [];
txtPointsYses = [];
```

Listing 59: Problem 5.10 c)

5.2.4 d)

With both polynomial functions the squared error can now be calculated:

```
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));
```

Listing 60: Calculating the squared errors of both polynomials

The resulting errors are:

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

Listing 61: Squared errors of both polynomials

As it can be seen, the error of the degree 4 polynomial is over 70 times smaller than the degree 2 polynomial. This is mainly due to the fact that the data points lay suitable for a degree 4 polynomial.

```
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 62: Plotting both polynomials

The plot of the polynomials look like this:

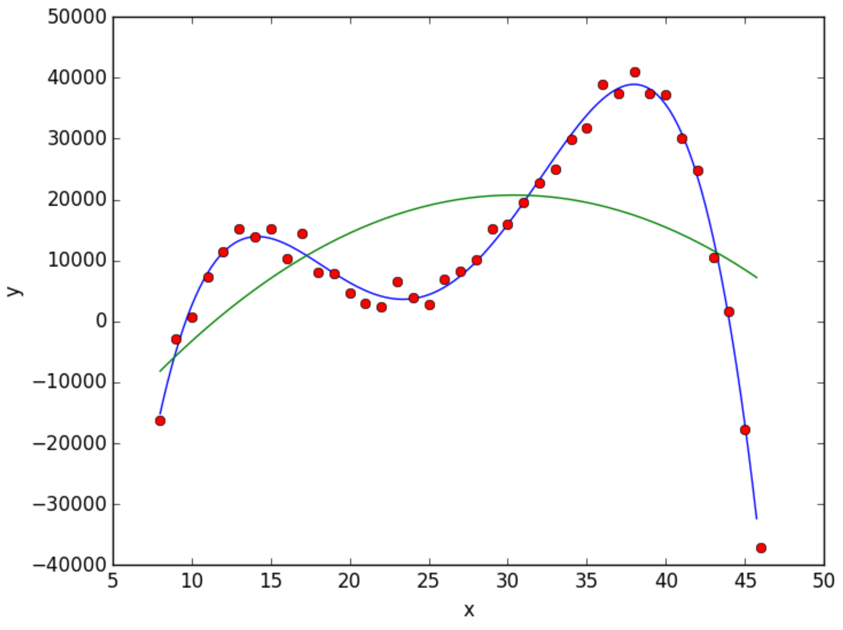
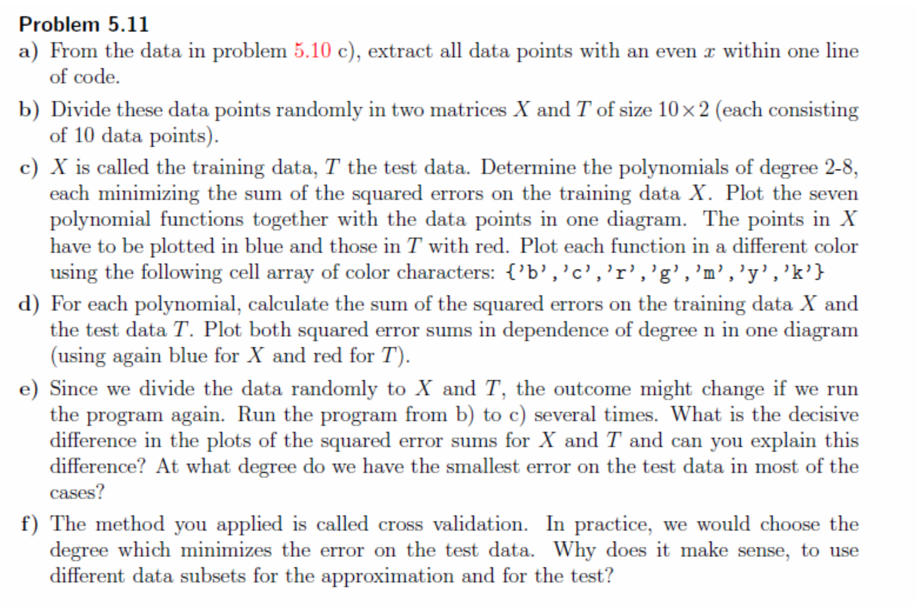


Figure 12: Result of Problem 5.10 d)

5.3 Problem 5.11



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?

The previously defined functions $getCoefficients$ and $polynomialF$ as well as the data points *txtPoints* from problem 5.10 will be used in this exercise too.

5.3.1 a)

```
evenPoints = [];

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

Listing 63: Problem 5.11 a)

5.3.2 b)

```
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);
```

Listing 64: Problem 5.11 b)

5.3.3 c)

To get the polynomials up to degree 8, first the base functions have to be defined:

```
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;
```

Listing 65: Base functions up to degree 8

By using these base functions the coefficients can be easily calculated with the *getCoefficients* function. Afterwards, the *y* values of those functions can be calculated with the *polynomialF* function in order to plot them.

```
xses = [];
allYses = [];

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 = ["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();
```

Listing 66: Problem 5.11 c)

The resulting graph looks like this.

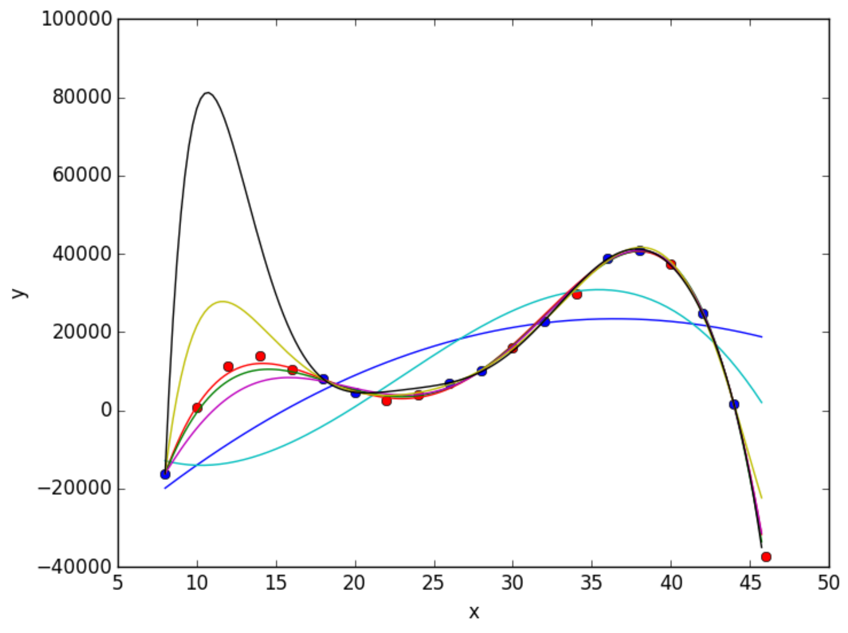


Figure 13: All 7 polynomials

5.3.4 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 67: Problem 5.11 d)

The resulting errors look the like the following:

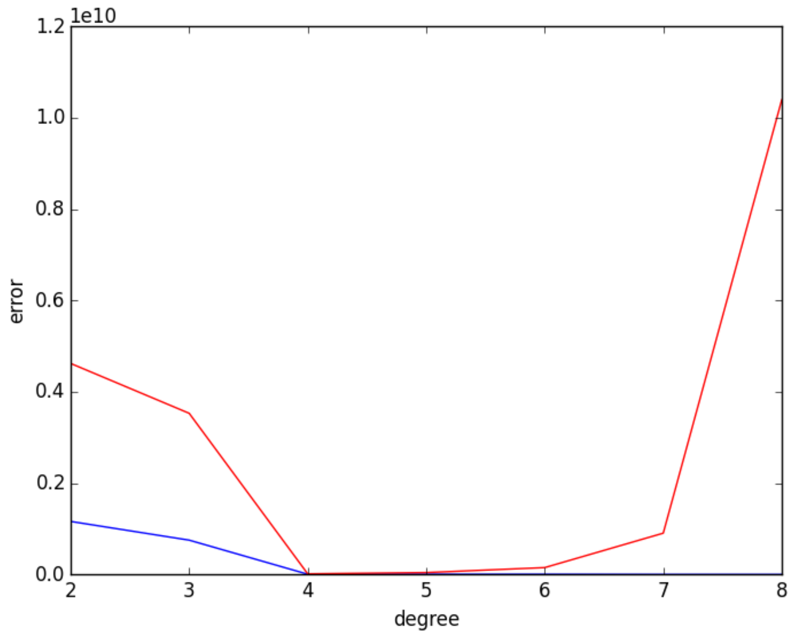


Figure 14: The errors of the polynomials

5.3.5 e)

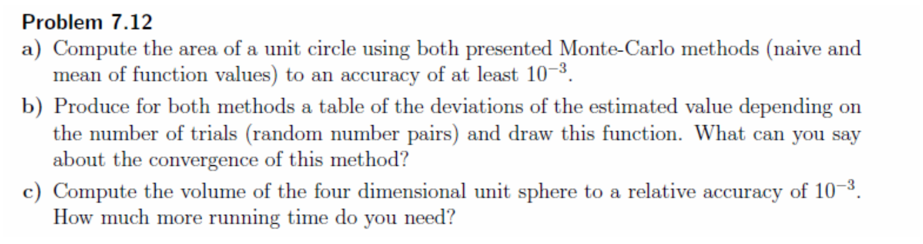
The deviation to the original approximated function (where all data points were used) minimizes the more evenly the points of X and T are distributed. The reason for that is that when there's a large interval of test points only, the approximated function has no bounds at all in that interval and can deviate as much as it needs to, without increasing the error.
For the training data X the error decreases the higher the degree of the polynomial is. For the test data T however the error first decreases up degree 4 and 5, but then rapidly increases with each additional degree. A reason for that could be that functions of higher degrees tend to have bigger distortions, and since there are no bounds on the test data points they are free to do so.
In most of the cases the polynomials of degree 4 and 5 had the best results.

5.3.6 f)

It makes sense because the approximated function is supposed to be close to all data values, 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?

6.1.1 a)

First we need a function whose integral over a certain interval is equal the area of the unit circle. The integral of the function $\sqrt{1-x^2}$ over the interval $[-1,1]$ equals half the area of the unit circle, so with a little tweaking of the parameter x we can modify the function to have an integral equal to the unit circle over the interval $[0,4]$. The function in Python looks the following way:

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

Listing 68: Function for the unit circle area

Afterwards, we can use this function for the calculation of the area with both the naive and mean Monte Carlo method:

```python
goal = math.pi;

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

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

        return circleF(randX) > randY;

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

        while abs(areaEstimated - goal) > precision:
                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:
            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("mean value method:");
print("estimated area: " + str(meanMonteCarlo[0]));
print("tries: " + str(meanMonteCarlo[1]));
```

Listing 69: Problem 7.12 a)

The results of those calculations are:

```
naive method:
estimated area: 3.14241702843
tries: 7141.0

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

Listing 70: Result of 7.12 a)

6.1.2 b)

To calculate the deviation we can use a slightly alternative version of the Monte Carlo functions of a).

```
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 71: Problem 7.13 b)

The resulting graph looks like this:

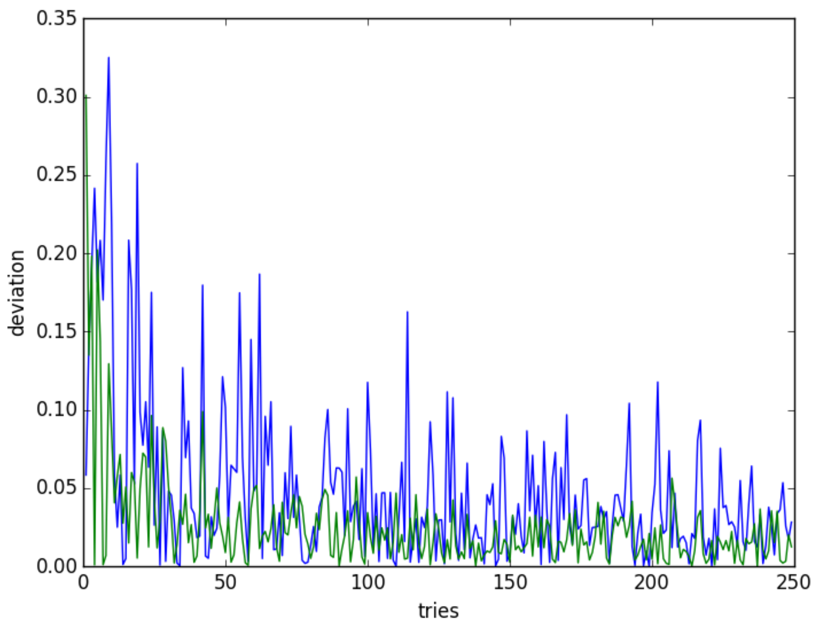


Figure 15: Deviation of the Monte Carlo methods

Although the deviation tends to get smaller and smaller there is no convergence. Due to the random factor both functions have there is no guarantee the y values get close to a certain value.

6.1.3 c)

To calculate the volume we again need a function to check whether a random point is within our body of choice. For an n-dimensional unit sphere we can simply check whether the length of the n-dimensional vector of the coordinates is shorter than 1. If so, the point is within the sphere. The Python function for the four-dimensional sphere can be implemented in the following way:

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

Listing 72: Function for the 4D unit sphere volume

Again we use this function for the Monte Carlo method to calculate the volume the same way we did in the subtask a):

```
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:
                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]));
```

Listing 73: 4D unit sphere volume calculation

The result of the calculation is the following:

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

Listing 74: Result of the 4D unit sphere volume calculation

Afterwards we can analyze the computational time of both methods:

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

Listing 75: Problem 7.12 c)

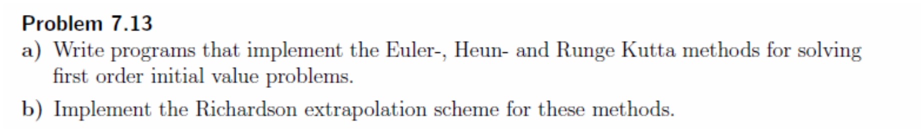
The results of these calculations are:

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

Listing 76: Result of 7.12 c)

Looking at the results, it takes about 3 times as much computational time to calculate the volume of a 4 dimensional unit sphere than it is to calculate the area of a 2 dimensional unit circle.

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)

To solve an ODE, first a general function can be defined for the basic approach. To make this function applicable for different approximation methods (like Euler method and Runge-Kutta method) the function which defines how to calculate the y_{n+1} value will be passed as an argument. This abstract function will be called *yp1Method* in this exercise.

```python
def odeSolver(f, a, b, y0, h, yp1Method):
        x = a;
        y = y0;

        yses = [];
        yses.append(y);

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

                x += h;

        return yses;
```

Listing 77: Function to solve ODE

Afterwards, we need to define the *yp1Method* functions for the Euler, Heun, and Runge-Kutta methods.

```python
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);
```

Listing 78: YP1Methods

The final ODE solving functions for each method can now call the basic *odeSolver* function and pass the respective *yp1Method*:

```python
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);
```

Listing 79: ODE solving functions for each method

To solve an ODE system the function has to be altered a little bit.

```python
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;
```

Listing 80: Function to solve an ODE system

Also a new *yp1Method* has to be defined:

```
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 81: YP1Method for solving an ODE system

6.2.2 b)

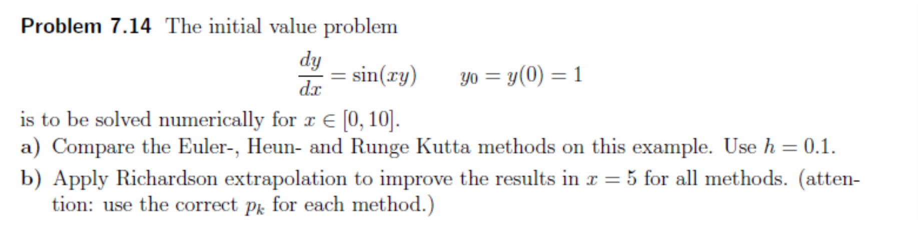
For the Richardson extrapolation we need to define new functions for the different *yp1Methods* again to calculate the value of the next y value. However, the basic ODE solving function can be reused.

```
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 82: Problem 7.13 b)

The required values for q and p_k have been empirically determined for each method.

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.)

6.3.1 a)

To calculate the respective y values we can simply use the functions from Problem 7.13:

```python
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 83: Problem 7.14 a)

The resulting graph of each method looks like the following:

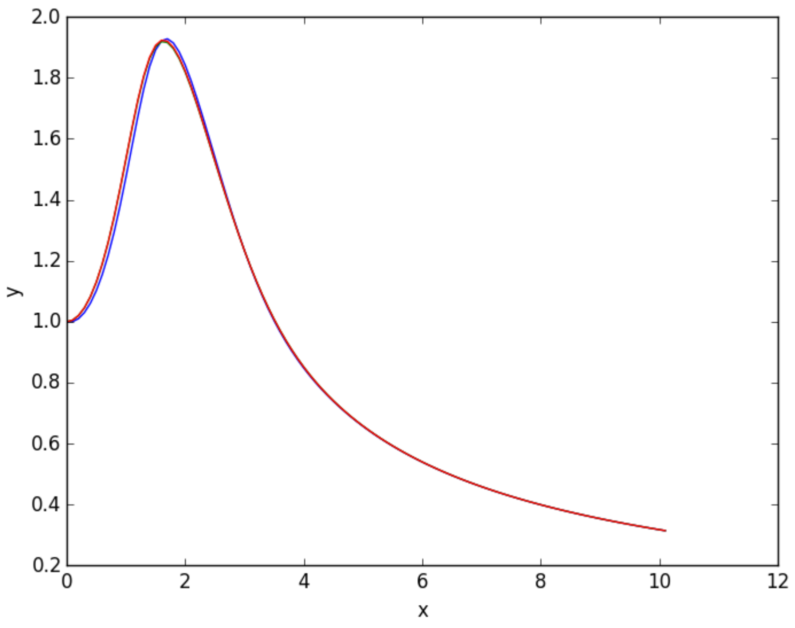


Figure 16: Graph of the results of the 3 methods

As it can be seen, the graphs of the functions are very close to each other and there is hardly any difference noticeable.

6.3.2 b)

Again, using the functions from Problem 7.13 we can simply calculate the y values with the Richardson extrapolation:

```
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 84: Problem 7.14 b)

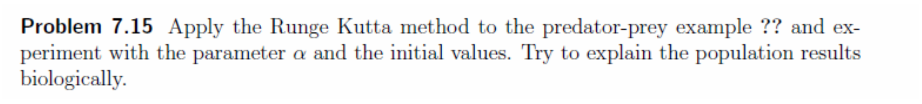
The result of these calculations are:

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

Listing 85: Result of 7.14 b)

As expected, the value approximated with Euler's method is slightly below the other method's values, however with the Richardson extrapolation this error is not as critical as it would have been if the regular methods had been used.

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.

To simulate the predator-prey-model an ODE system has to be created, with one equation for the sheep and one for the wolfs. Afterwards, the solution can be simply calculated with the script from Problem 7.13:

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

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

fses = [];
y0ses = [];

fses.append(sheepFunc);
fses.append(wolfsFunc);
y0ses.append(3);
y0ses.append(1);

yVectors = dif.rungeKuttaSystem(fses, 0, 5, y0ses, 0.05);

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

x = 0;

for yVector in yVectors:
        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();
```

Listing 86: Problem 7.15

Depending on α and the initial number of sheep and wolfs the graphs look like this:

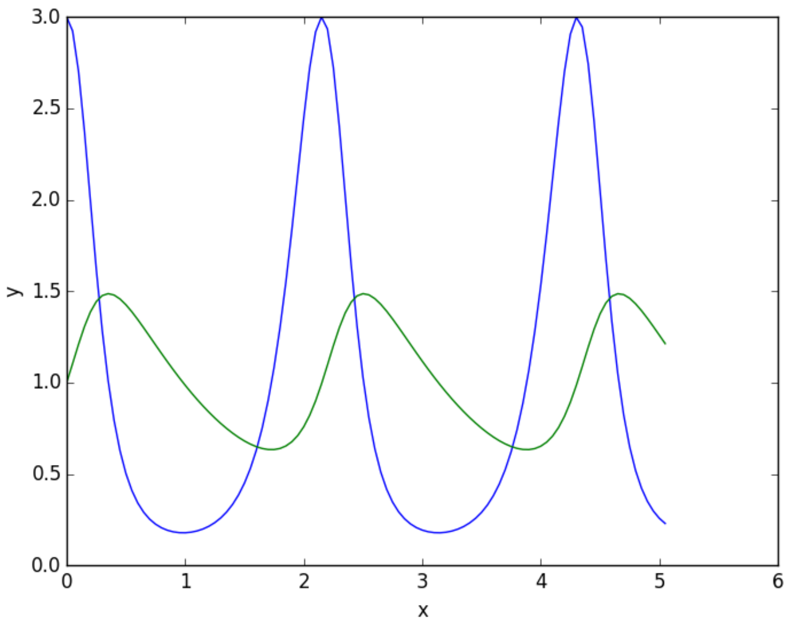


Figure 17: $\alpha = 10$, initially 3 sheep and 1 wolfs

Biologically, this could be explained by the balance between predator and prey which is mandatory for both species to survive.

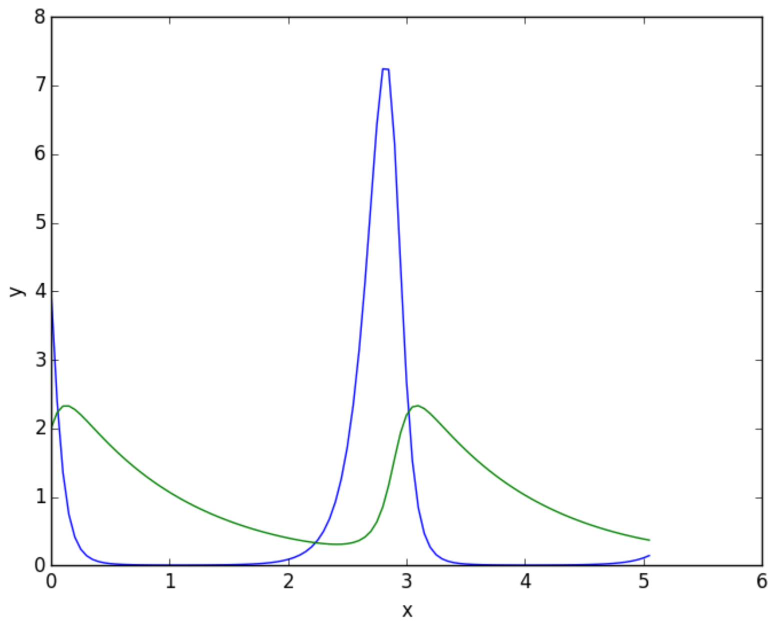


Figure 18: $\alpha = 9$, initially 4 sheep and 2 wolfs

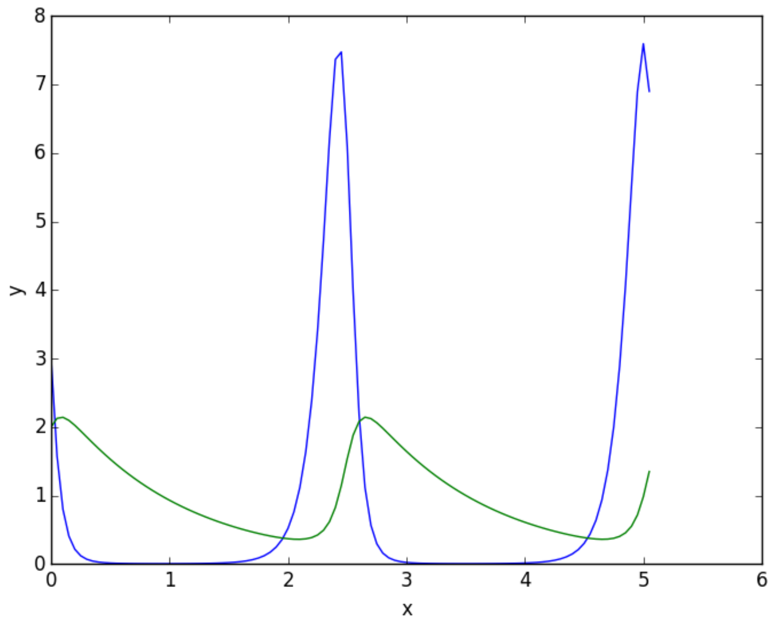
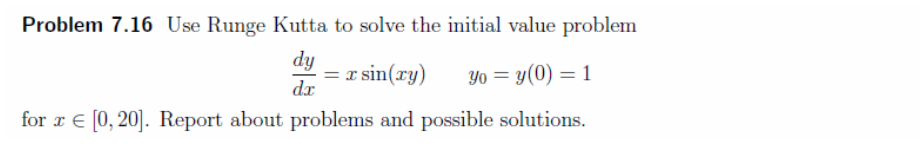


Figure 19: $\alpha = 12$, initially 3 sheep and 2 wolfs

6.5 Problem 7.16



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.

```python
def myF(x, y):
        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 87: Problem 7.16

The resulting graph looks like:

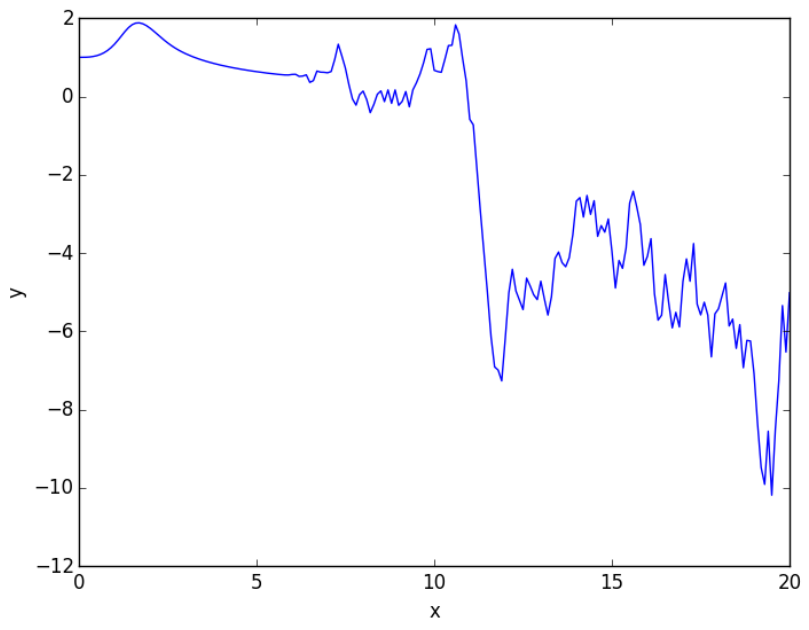


Figure 20: Graph of Problem 7.16

Since the x parameter is a factor both inside and outside of the sine, the frequency as well as the amplitude increase as x grows. Hence, the fluctuation increases and the slopes get steeper and steeper. Because the step size stays the same, the curve gets “spikier” and slightly inaccurate. A possible solution could be to decrease the step size.