ℹ Informasjon om eksamen

UNIVERSITY OF OSLO
Faculty of mathematics and natural sciences
Written exam in IN1900
2021 Fall

Time for exam: 14.12.21, 09:00 til 14.12.21, 13:00

Permitted aids: None.
A calculator is available in Inspera.

It is important to read this information carefully before you start.

The exam contains multiple choice questions, and text questions where you shall write short programs or read programs and write the output from the program. There are 15 questions that should be answered in total, and in total 75 points available on these questions. Question 16 shall not be answered, but is used by the examiners during the grading of the exam, to include the points from the mid term exams.

The number of points available is specified for each question. For questions with multiple sub-questions, each sub-question has the same number of points.

If you are missing information you can make your own reasonable assumptions, as long as they are in line with the "nature" of the question. In text questions you should then specify the assumptions you made, for instance in comments to the code.

All code in the question texts is written in Python 3.

Most of the questions lead to short code with little need for comments, unless you do something complicated or non-standard. (which is not recommended; but in this case the comments shall explain the ideas behind the program to make it easier to evaluate the code).

A question may ask you to write a function. A main program which calls the function is in this case not needed, unless it is specifically asked for in the question text.

1 Løkke som teller

What is the value of the variable n when these program statements have been executed?

```
n = 0
k = 10
while n < k:
    n = n + 2
    k = k + 1
```

Select one alternative:

- 10
- 12
- 14
- 16
- 18
- [✓ 20]
- 22
- 24

Maximum marks: 2

2 Hvilke hører sammen?

Assume that we have the list **x = [[-1],[1], 1, 2, 0].** To the left of the table below there are seven expressions, and at the top of the table there are six possible results of evaluating these expressions. Choose the correct alternative for each expression.

Please match the values:

	0	1	[-1]	[1]	[[-1]]	[[1]]
x[-1]	✓	○	○	○	○	○
x[x[-1]]	○	○	✓	○	○	○
x[x[2]-1]	○	○	✓	○	○	○
x[x[2]:x[3]]	○	○	○	○	○	✓
x[2:3]	○	○	○	✓	○	○
x[1:2]	○	○	○	○	○	✓
x[x[0][0]]	✓	○	○	○	○	○

Maximum marks: 3.5

3 Hvilken linje mangler?

The function **diff2nd(f,x,h)** computes an estimate of the second derivative of a function f in a point x:

```
def diff2nd(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/(h*h)
    return r
```

In the code defined below we want to use the function to estimate the second derivative of $f(x) = 2x^3 - 3x$ for x = 1.5, and to study how the estimate changes as we reduce h.

```
for k in range(1,8):
    h = 10**(-k)
    #Missing line goes here
    print(f'h={h}: {d2g}')
```

One line is missing in the code. Which of the following lines should be inserted to make the code do what we want?

Select one alternative:

[✓ d2g = diff2nd(lambda x: 2*x**3-3*x, 1.5, h)]

d2g = diff2nd(2*x**3-3*x, 1.5, h)

f = diff2nd(f=2*x**3-3*x, 1.5, h)

d2g = diff2nd(f=2*x**3-3*x, 1.5)

Maximum marks: 2

4 Funksjon med parametre

What is printed in the terminal when this program is run?

```
class F:
   def __init__(self, alpha, beta):
        self.alpha = alpha
        self.beta = beta

   def __call__(self, x):
      return self.alpha + self.beta * x

   def __add__(self, obj):
      return F(self.alpha+obj.alpha, self.beta+obj.beta)

a = F(0,1)
b = F(2,3)
c  = (a + b)(2)
print(c)
```

Select one alternative:

- ○ En feilmelding
- ○ c
- ○ 2
- ○ 8
- [✓ 10]
- ○ 12
- ○ 18

Maximum marks: 2

5 Logiske uttrykk

Decide for each of the logical expressions on the left side of the table what the result is when they are evaluated.

Please match the values:

	False	True
True or False and True	○	[✓ X]
True and (not False or True)	○	[✓ X]
not (False and True)	○	[✓ X]
[e for e in [False,True] if not e][-1]	[✓ X]	○
not ((2==3) and not(2==4))	○	[✓ X]

Maximum marks: 2.5

6 Finne største verdi

We have a list x containing numbers (for instance x = [-5, -2, -6, -3, -8, -3, -5, -16]) , and we want to find the larges value in this list. Which of the code alternatives on the left will result in y being equal to the largest value in x? (answer "Yes" for the alternatives that you believe result in y being equal to the largest value in x, and answer "No" for the alternatives you believe do not result in y being equal to the larges value in x).

Please match the values:

	No	Yes
Alternative D	✓	
Alternative C		✓
Alternative G		✓
Alternative A	✓	
Alternative F		✓
Alternative E		✓
Alternative B		✓

Maximum marks: 3.5

```
# Alternative A
y = -1
for e in x:
   if e > y:
      y = e

# Alternative B
y = x[-1]
for i in range(len(x)):
   y = max(y,x[i])

# Alternative C
for i in range(len(x)-1):
   if x[i+1] < x[i]:
      x[i+1] = x[i]
y = x[-1]

# Alternative D
v = [x[i] for i in range(1, len(x)) if x[i] > x[i-1]]
y = v[-1]

# Alternative E

import numpy as np
v = np.asarray(x)
y = -min(-v)

# Alternative F
v = [x[0]]
for i in range(len(x)):
   if v[-1] <= x[i]:
      v.append(x[i])
y = v[-1]

# Alternative G
s = f"y=max({x})"
exec(s)
```

7 Feilhåndtering (exceptions)

One line is missing in the following program:

```
import sys
try:
   # One line is missing here
except IndexError:
   print("Feil A")
   sys.exit()
except ValueError:
   print("Feil B")
   sys.exit()
except ZeroDivisionError:
   print("Feil C")
   sys.exit()
```

We replace the missing line with one of the alternatives on the left side of the table below, and run the program with the command

```
Terminal> python check.py 4 0
```

Decide for each row in the table what is printed by the program.

Please match the values:

	Feil A	Feil B	Feil C	No output
j = sys.argv[1]	○	○	○	✓
j,k = sys.argv[2], sys.argv[3]	✓	○	○	○
j = float(sys.argv[0])	○	✓	○	○
h = 1/int(sys.argv[2])	○	○	✓	○
k = sys.argv[:3]	○	○	○	✓

Maximum marks: 2.5

8 Numerisk derivasjon

The derivative of a function $f(x)$ can be estimated with a centered difference:

$$f'(x) \approx \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}$$

where $\Delta x > 0$ is a constant.

Write a Python function **midpoint(f, x0, dx=1e-4)**, which uses this formula to estimate the derivative of the function f(x) in the point x0. The function shall return the estimated value. Include code to call the function to estimate the derivative of *sin(x)* in the point $x = \pi/4$. Include necessary import.

Fill in your answer here

```

def midpoint(f, x0, dx=1e-4):
  return (f(x0+dx)-f(x0-dx))/(2*dx)

import math
estimate = midpoint(math.sin, math.pi/4)
```

Maximum marks: 3

9 Beregning av en sum

The inverse sine function $\arcsin x$ can be approximated by:

$$\arcsin x \approx \sum_{n=0}^{N} \frac{(2n)!}{2^{2n}(n!)^2} \frac{x^{2n+1}}{2n+1}$$

where $n!$ is the factorial of n.

a) Write a Python function **arcsin_approx(x,N)** which for given x and N computes the sum above and returns it. Use the function **factorial(n)** from the **math** module to compute the factorial (n!).

b) To study how good the approximation above is, we want to compare the answer we get with the answer from the builtin function **asin(x)** in the **math** module. Write a program which uses the function from question a) to approximate the inverse sine function for x = 0.5 og N=1, 2, ..., 20 and prints a table with three columns: **N**, **arcsin_approx(x,N)** and the error **abs(asin(x) - arcsin_approx(x,N)**).

Fill in your answer here

```

# Question a)
def arcsin_approx(x,N):
  from math import factorial as fac
  s = 0
  for n in range(N+1):
    nn = 2*n
    s += fac(nn)/(2**nn*fac(n)**2) * x**(nn+1)/(nn+1)
  return s

# Question b)
from math import asin
x = 0.5
print(" N   Approx       Error")
for N in range(1,21):
    approx = arcsin_approx(x,N)
    error = abs(asin(x)-arcsin_approx(x,N))
    print(f"{N:2d}   {approx:2.8f}   {error:2.8f}")
```

Maximum marks: 6

10 Klasse for tilfeldige tall

We cannot make the computer generate truly random numbers, but there are several so-called *pseudo random number generators* (PRNGs) which can generate sequences of numbers that in practice can be assumed to be random. The function below implements such a PRNG. The argument n specifies how many random numbers we want to generate, and the other arguments seed seed, m, a og b can be used to adjust how the algorithm behaves.

```
def rand_gen(n, seed, m, a, b):
    res = [ ]
    x = seed
    for i in range(n):
        x = (a * x + b) % m
        res.append(x/(m-1))
    return res
```

a) We want to implement the random number generator above as a function of n alone, with seed, m, a og b as parameters to the function. Do this by implementing the random number generator as a class **RandomGenerator** with two special methods, so that the following code works:

```
rand = RandomGenerator(seed=0.01, m=2**16+1, a=75, b=74)
x = rand(50)   # Shall generate a list of 50 random numbers
```

b) Extend the class **RandomGenerator** with a new special method so that the following code works:

```
rand = RandomGenerator(seed=0.01, m=2**16+1, a=75, b=74)
print(rand)   # Prints the values of the parameters seed, m, a and b
```

Fill in your answer here

```python
class RandomGenerator:

    def __init__(self, seed, m, a, b):
        self.seed, self.m, self.a, self.b = seed, m, a, b

    def __call__(self, n):
        res = []
        x = self.seed
        for i in range(n):
            x = (self.a * x + self.b) % self.m
            res.append(x/(self.m-1))
        return res

    def __str__(self):
        return f"seed={self.seed}, m={self.m}, a={self.a}, b={self.b}"
```

Maximum marks: 6

11 DNA-sekvenser

A DNA sequence is a text string of arbitrary length which only contains the four letters A, C, G, T. An example of a DNA sequence is "CGTACGCCGA". In this exercise you shall make a program to analyze such DNA sequences. Initially, the DNA sequence is stored in a text file and needs to be read from this. Such a file includes a single DNA sequence, but for readability the sequence is usually divided into several numbered lines. An example of the contents of such a file (note: the numbers on the left are also part of the file):

```
1       CGTACGCCGACGTACGCCGA
2       CGCCGACGTACGCGCCGACG
3       GTACGCGCCGACGGTACGCG
```

In this specific case the DNA sequence is divided into three lines, but the number of lines can vary from file to file. Each line contains a line number, followed by one or more blanks, and the a part of the DNA sequence. The complete DNA sequence is obtained by joining (concatenating) these partial sequences in the order they appear in the file.

a) Write a function **read_DNA(filename)** which reads a DNA sequence from a file on the format given above, and returns a text string containing the DNA sequence. You can assume that the file has the correct format. Remember that the line numbers and the blanks on each line shall not be part of the text string returned by the function.

b) Write a function **find_CG(dna)** which takes a text string with a DNA sequence as input, and returns a list with all the positions where the sequence "CG" occurs. If, for instance, **dna** is the sequence "CGATCGCG", the function shall return the list [0, 4, 6], and if **dna** is the sequence "AAT" it shall return the empty list [].

c) Write a test function **test_find_CG()** for the function in question b). The test function shall verify that **find_CG** returns the correct result for the two sequences "TATACG" og "CGCGCG".

d) When the sequence "CG" occurs k times consecutively in a DNA sequence, we call it a CG repetition of length k. For instance, the sequence "CGAACGCGCG" contains one CG repetition of length 1 and one CG repetition of length 3. Write a function **longest_CG_repetition(dna)** which takes a text string with a DNA sequence as input, and returns the lenght of the longest CG repetition in the sequence. For instance, the function shall return the value 3 for the DNA sequence "CGAACGCGCG" and the value 0 for the DNA "AAGCTCA".

Fill in your answer here

```
def read_DNA(filename):
    infile = open(filename, 'r')
    dna_seq = []
    for line in infile:
        words = line.split()
        dna_seq.append(words[1])
    return ''.join(dna_seq)

def find_CG(dna):
    pos = []
    for i in range(len(dna)-1):
        if dna[i:i+2]=="CG":
            pos.append(i)
    return pos

def test_find_CG():
    s = "TATACG"
    expected = [4]
    computed = find_CG(s)
    assert expected==computed, f"Error detected: {s}"
    s = "CGCGCG"
    expected = [0,2,4]
    computed = find_CG(s)
    assert expected==computed, f"Error detected: {s}"
```

```
def longest_CG_repetition(dna):
    pos = find_CG(dna)
    res = count = 1
    for k in range(1,len(pos)):
        if pos[k]==pos[k-1]+2:
           count += 1
           res = max(res,count)
        else:
           count = 1
    return min(res,len(pos))
```

Maximum marks: 12

12 Nullpunkter

Assume that we numerically want to find a root (zero) of a given function f, i.e., we want to find x such that $f(x) = 0$. The secant method does this by solving the following difference equation for $n = 2, 3, \ldots, N$:

$$x_n = \frac{f(x_{n-1})x_{n-2} - f(x_{n-2})x_{n-1}}{f(x_{n-1}) - f(x_{n-2})}$$

where x_0 and x_1 are initial conditions. If we believe the solution is in the interval $[0, 5]$, we can for instance choose $x_0 = 0$ og $x_1 = 5$.

a) Write a function **secant(f, x0, x1, N)** which implements the secant method for a given function f, with initial conditions x0 and x1, and where N is the upper limit as indicated above. The function shall return the last computed value x_N.

b) In practice we do not know how many terms that must be computed (i.e., how large N must be) to achieve $f(x_N) \approx 0$. It is often more useful to set a tolerance value $\varepsilon > 0$, and then compute exactly the number of terms N that are needed to achieve $|f(x_N)| < \varepsilon$. Write a function **secant_tol(f, x0, x1, eps, Nmax)** which does this. Here f is the function for which we want to find the root, x0 and x1 are initial values, eps is the tolerance, and Nmax is the maximum number of terms to be computed in the sequence x_n. The function shall return the last computed value of the sequence, regardless of whether this satisfies the tolerance criterion or not. Note: the function shall compute *exactly the numbers of terms* in the sequence that is needed to satisfy the tolerance criterion, but if the criterion is still not satisfied after computing Nmax terms, the function shall return the value x_{Nmax}.

Fill in your answer here

```

def secant(f,x0,x1,N):
    import numpy as np
    x = np.zeros(N+1)
    x[0],x[1] = x0,x1
    for n in range(2,N+1):
        z1,z2 = x[n-1],x[n-2]
        f1,f2 = f(z1),f(z2)
        x[n] = (f1*z2-f2*z1)/(f1-f2)
    return x[N]

def secant_tol(f,x0,x1,eps,Nmax):
    import numpy as np
    x = np.zeros(Nmax+1)
    x[0],x[1] = x0,x1
    n = 2
    while n<Nmax+1 and abs(f(x[n-1]))>=eps:
        z1,z2 = x[n-1],x[n-2]
        f1,f2 = f(z1),f(z2)
        x[n] = (f1*z2-f2*z1)/(f1-f2)
        n += 1
    return x[n-1]
```

Maximum marks: 6

13 ODE-system for været

We will study a system of differential equations called the Lorenz model. The model is meant to give an extremely simplified description of the weather, and in particular to model unpredictable behaviour of the weather. The model consists of three functions $x(t)$, $y(t)$ og $z(t)$, where $t \geq 0$. Somewhat simplified, $x(t)$ is a measure of the motion in the atmosphere, while $y(t)$ og $z(t)$ describes the horizontal and vertical temperature distribution in the atmosphere. The following system of differential equations describes how $x(t)$, $y(t)$ og $z(t)$ evolve over a time interval [0,T]:

$$\begin{aligned} x'(t) &= a(y(t) - x(t)) \\ y'(t) &= bx(t) - y(t) - x(t)z(t) \\ z'(t) &= x(t)y(t) - cz(t) \end{aligned}$$

At time t=0 we have the initial conditions x(0)=x0, y(0)=y0 og z(0)=z0. The constants a, b, c are all positive, and in addition we have $a > c + 1$.

a) Write a function **Lorenz(x0, y0, z0, a, b, c, T, N)** which takes initial values x0, y0 and z0, parameters a, b and c, the final time T, and the number of time points N som arguments, og computes and returns the solution for $t_k = k \cdot \Delta$ where $k = 0, 1, \ldots, N-1$ and $\Delta = T/(N-1)$. Use the RungeKutta4 method in the ODESolver module to solve the equations. The relevant parts of the ODESolver module are included as an attachment to this question. The function **Lorenz** shall return four numpy-arrays, all having length N:

t, which contains the time points t_k where the numerical solution is computed
x, which contains $x(t_0), x(t_1), \ldots, x(t_{N-1})$
y, which contains $y(t_0), y(t_1), \ldots, y(t_{N-1})$
z, which contains $z(t_0), z(t_1), \ldots, z(t_{N-1})$.

b) We want to solve the model with initial values $x_0 = 0,\ y_0 = 2,\ z_0 = 0$ and with parameter values $a = 4,\ b = 16,\ c = 0.5$. Write code to call the function **Lorenz** with the given parameters and with T=100 and N=500. Also include code to plot the solutions $x(t)$, $y(t)$ og $z(t)$. The three curves shall be plotted with different colours, in the same window, and with a legend which explains which colour is $x(t)$, $y(t)$ and $z(t)$, respectively. Include necessary import.

c) It is well known that the Lorenz is very sensitive to the initial conditions, i.e., that very small perturbations of x(0), y(0) or z(0) can have a large impact on the solutions $x(t)$, $y(t)$ og $z(t)$ at a later time t. To investigate this behaviour we shall keep the values of the parameters a, b, c and the two last initial values y_0, z_0 fixed, so that the values of $x(t)$, $y(t)$ og $z(t)$ at time $t = T$ are determined from (i.e., are a function of) the initial condition $x_0$only. Use the same values for the parameters a, b, c and for the two last initial conditions y_0, z_0 as in question b). Also let T og N have the same values as in question b). Write code which chooses 100 different values for the initial value x_0 on the interval [0, 0.01] and for each such value solves the Lorenz model and stores the value of $x(t)$ at time t=T. Then write code to plot the value of $x(T)$ as a function of the x_0 value.

Fill in your answer here

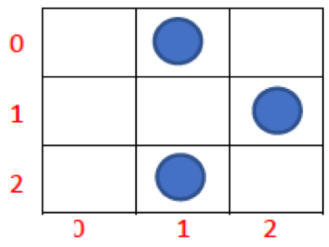
```python
from ODESolver import *
import numpy as np

# Question a)
def Lorenz(x0,y0,z0,a,b,c,T,N):
    def f(u,t):
        x,y,z = u[0],u[1],u[2]
        return [a*y-a*x, b*x-y-x*z,x*y-c*z]
    rk = RungeKutta4(f)
    rk.set_initial_condition([x0,y0,z0])
    u,t = rk.solve(np.linspace(0,T,N))
    return t,u[:,0],u[:,1],u[:,2]

# Question b)
import matplotlib.pyplot as plt
t,x,y,z = Lorenz(0,2,0,4,16,0.5,100,500)
plt.plot(t,x,t,y,t,z)
plt.legend(['x','y','z'])
plt.show()

# Question c)
x0,xN = np.linspace(0,0.01,100), np.zeros(100)
for i in range(100):
    t,x,y,z = Lorenz(x0[i],2,0,4,16,0.5,100,500)
    xN[i] = x[-1]
plt.plot(x0,xN)
plt.show()
```

Maximum marks: 9

```python
import numpy as np

class ODESolver:
    def __init__(self, f):
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # scalar ODE
            self.neq = 1                 # no of equations
            U0 = float(U0)
        else:                            # system of ODEs
            U0 = np.asarray(U0)
            self.neq = U0.size           # no of equations
        self.U0 = U0

    def solve(self, time_points):
        self.t = np.asarray(time_points)
        N = len(self.t)
        if self.neq == 1:  # scalar ODEs
            self.u = np.zeros(N)
        else:              # systems of ODEs
            self.u = np.zeros((N,self.neq))

        self.u[0] = self.U0

        for n in range(N-1):
            self.n = n
            self.u[n+1] = self.advance()
        return self.u, self.t

class ForwardEuler(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = t[n+1] - t[n]
        unew = u[n] + dt*f(u[n], t[n])
        return unew

class RungeKutta4(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = t[n+1] - t[n]
        dt2 = dt/2.0
        k1 = f(u[n], t[n])
        k2 = f(u[n] + dt2*k1, t[n] + dt2)
        k3 = f(u[n] + dt2*k2, t[n] + dt2)
        k4 = f(u[n] + dt*k3, t[n] + dt)
        unew = u[n] + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)
        return unew
```

14 Brettspill

We will here program a simple board game. Let N > 1 be an integer. We have a board consisting of N x N squares, where rows and columns are numbered from 0 to N-1. We also have N pieces placed on the board (never two in the same square). Here's an example with N = 3:



The size of the board and the initial position of the pieces are stored in the text file start.txt. The first line gives N, and the next lins give the piece type, row number and column number (separated by commas) for each piece. There are two different types of pieces and these are given as 1 or 2 in the file. This is what the file would look like for the example above:

```
3
2,0,1
1,1,2
1,2,1
```

Here the first piece is of type 2 and the two next ones are of type 1.

Vi skal implementere spillet som en klasse:

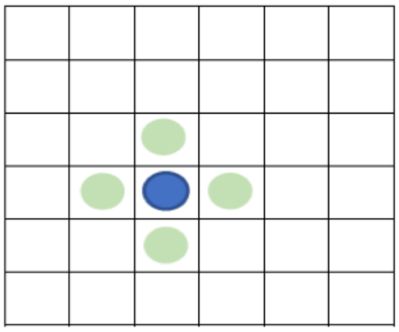
```
class Game:
  def __init__(self):
     # Insert code here (see question a below)

  def move1(self, row0, col0, row1, col1):
     # Insert code here (see question b below)

  def move2(self, row0, col0, row1, col1):
     # Insert code here (see question c below)
```

a) Write the constructor of the class **Game**, so that it reads the file start.txt and saves the positions of the pieces in a 2-dimensional numpy-array **board** with N rows og N columns. Thsi variable should be an instance variable and can be created like this (after N has been read from the file):

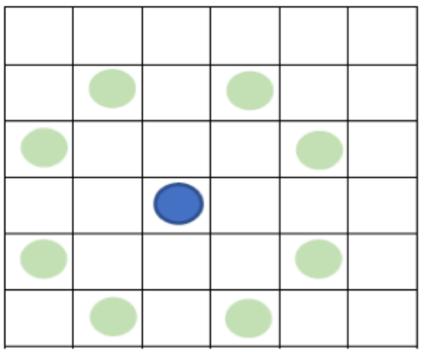
```
import numpy as np
self.board = np.zeros((N,N), 'int')
```

When you have saved the positions of the pieces, self.board[i,j] == 0 if there is is no piece in the square on the i'th row and j'th column, self.board[i,j] == 1 if the square contains a piece of type 1, and self.board[i,j] == 2 if the square contains a piece of type 2.

b) Write the function **move1(self, row0, col0, row1, col1)** which moves a piece of type 1 from the position row0, col0 to the position row1, col1, by making the necessary changes to the instance variable **board**. First the function needs to check that the move is legal. The criteria for a legal move are (a) the square row0, col0 must contain a piece of the correct type and the new square row1, col1 must be empty (b) pieces can only move 1 position horizontally (left or right) or one position vertically (up or down), and (c) the piece must not be moved out of the board. The figure below shows an example with all the legal moves for such a piece. Here we have N=6, blue indicates the start position, and green indicates possible positions after the move. If the move is illegal the function shall end without altering the variable **board**, but it is not necessary to print an error message or a message that the move is illegal.



c) Write the function **move1(self, row0, col0, row1, col1)** which moves a piece of type 2 from the position row0, col0 to the position row1, col1 by making the necessary changes in the instance variable **board.** Here you must also first check that the move is legal. The criteria are that there must be a piece of type 2 in the square row0, col0, and the new square row1, col1 must be empty. Pieces of type 2 always move two squares vertically and one square horizintally, or two squares horizontally and one square vertically. You must also check that the piece is not moved outside the board. The figure below shows an example with all legal moves for such a piece. Here we have N=6, blue indicates the start position, and green indicates possible positions after the move. If the move is illegal the function shall end without altering the variable **board**, but it is not necessary to print an error message or a message that the move is illegal.



Fill in your answer here

```
import numpy as np

class Game:
    def __init__(self):
        infile = open('start.txt', 'r')
        self.N = int(infile.readline())
        self.board = np.zeros((self.N,self.N), 'int')
        for line in infile:
            words = line.split(',')
            a,b,c = int(words[0]),int(words[1]),int(words[2])
            self.board[b,c] = a

    def move1(self, row0, col0, row1, col1):
        crit0 = 0<=row0 and row0<self.N and 0<=col0 and col0<self.N
        crit1 = 0<=row1 and row1<self.N and 0<=col1 and col1<self.N
        crit2 = abs(row0-row1) + abs(col0-col1) == 1
        crit3 = self.board[row0,col0]==1
        crit4 = self.board[row1,col1]==0
        if crit0 and crit1 and crit2 and crit3 and crit4:
            self.board[row0,col0] = 0
            self.board[row1,col1] = 1

    def move2(self, row0, col0, row1, col1):
        crit0 = 0<=row0 and row0<self.N and 0<=col0 and col0<self.N
        crit1 = 0<=row1 and row1<self.N and 0<=col1 and col1<self.N
        crit2_1 = abs(row0-row1)==1 and abs(col0-col1)==2
        crit2_2 = abs(row0-row1)==2 and abs(col0-col1)==1
        crit3 = self.board[row0,col0]==2
        crit4 = self.board[row1,col1]==0
        if crit0 and crit1 and (crit2_1 or crit2_2) and crit3 and crit4:
            self.board[row0,col0] = 0
            self.board[row1,col1] = 2
```

Maximum marks: 9

15 B-splines

B-splines is a type of function which is used in a number of applications, for instance in computer graphics and numerical derivation. In this question you will write Python functions that compute the values of such B-splines.

The starting point for B-splines is a set of values (herafter called nodes) $x_{-m} < x_{-m+1} < \ldots < x_{n+m}$ where $m > 0$ and $n > 0$ (notice that the first m elements have negative index). *The B-splines functions of degree 0 are called* $B^0_{-m}(t), B^0_{-m+1}(t), \ldots, B^0_{n+m-1}(t)$ and are defined as follows:

$$B^0_p(t) = \begin{cases} 0, & t < x_p \text{ or } t > x_{p+1} \\ (x_{p+1} - x_p)^{-1}, & x_p < t < x_{p+1} \\ \frac{1}{2}(x_{p+1} - x_p)^{-1}, & t = x_p \text{ or } t = x_{p+1} \end{cases}$$

where $p = -m, -m+1, -m+2, \ldots, n+m-1$ and where t is a real number.

a) Write a Python function **Bspline0(t, p, x)** which computes and returns the value of $B^0_p(t)$. The arguments to the function are t (a float), p (an integer), og x (a dictionary with all the nodes, given so that the values $x_{-m}, x_{-m+1}, \ldots, x_{n+m}$ can be obtained as x[-m], x[-m+1], ..., x[n+m]). You can easily find the values of *m* and *n* by looking at the smallest and largest key value in x. (these are, respectively, equal to -*m* and *n+m*). You can assume that p is n the correct interval, i.e. that p has one of the values $-m, -m+1, -m+2, \ldots, n+m-1$.

The B-splines functions of degree $k > 0$ are called $B^k_{-m}(t), B^k_{-m+1}(t), \ldots, B^k_{n+m-k-1}(t)$, and these are defiined as follows:

$$B^k_p(t) = \frac{(t-x_p)B^{k-1}_p(t)+(x_{p+k+1}-t)B^{k-1}_{p+1}(t)}{x_{p+k+1}-x_p}$$

where $p = -m, -m+1, -m+2, \ldots, n+m-k-1$.

b) Write a Python function **Bspline(t, k, p, x)** which computes and returns the value $B^k_p(t)$. The arguments to the function are t (a float), k (a non-negative integer), p (an integer), and x (rgumentene til funksjonen er t (en float-verdi), k (et ikkenegativt heltall), p (et heltall), og x (a dictionary with all the nodes, given so that the values $x_{-m}, x_{-m+1}, \ldots, x_{n+m}$ can be obtained as x[-m], x[-m+1], ..., x[n+m]). As in question a) you can assume that p is in the correct interval (that is, p has one of the values $-m, -m+1, -m+2, \ldots, n+m-k-1$).

Fill in your answer here

```

def Bspline0(t,p,x):
   if t<x[p] or t>x[p+1]:
      return 0
   elif x[p]<t and t<x[p+1]:
      return 1/(x[p+1]-x[p])
   else:
      return 0.5/(x[p+1]-x[p])

def Bspline(t,k,p,x):
   if k==0:
      return Bspline0(t,p,x)
   elif k>0:
      a =(t-x[p])*Bspline(t,k-1,p,x)+(x[p+k+1]-t)*Bspline(t,k-1,p+1,x)
      b = x[p+k+1]-x[p]
      return a/b
```

Maximum marks: 6

```python
"""
Here are some alternative solutions to question 15. The one
based on recursion (also listed above) is the simplest and most elegant,
but recursion was not part of the IN1900 curriculum in 2021. The
question can also be solved with a loop, but this is more complicated
and makes this one of the most difficult questions on the exam.

In the exam setting, with limited time and no possibility to run the
code, it is very difficult to get all the indices correct in the loop
version. A good score is given to all solutions that get the overall loop
structure correct, even with some minor errors in indices etc.
"""
def Bspline0(t,p,x):
    if t<x[p] or t>x[p+1]:
        return 0
    elif x[p]<t and t<x[p+1]:
        return 1/(x[p+1]-x[p])
    else:
        return 0.5/(x[p+1]-x[p])

def Bspline(t,k,p,x):
    """The simplest solution, using recursion"""
    if k==0:
        return Bspline0(t,p,x)
    elif k>0:
        a =(t-x[p])*Bspline(t,k-1,p,x)+(x[p+k+1]-t)*Bspline(t,k-1,p+1,x)
        b = x[p+k+1]-x[p]
    return a/b



def Bspline_loop1(t,k,p,x):
    """
    To compute 1 Bspline function of degree k, we need
    2 Bsplines of degree k-1, 3 of degree k-2, ..., and k+1
    of degree 0. In this version we store the
    function values in a list of dictionaries."""

    if k == 0:
        return Bspline0(t,p,x)

    bsplines = []
    bsplines.append({p+k_: Bspline0(t,p+k_,x) for k_ in range(k+1)})
    for i in range(1,k+1):
        prev = bsplines[i-1]
        new = {}
        for j in range(k+1-i):
            a = (t-x[p+j])*prev[p+j]+(x[p+i+1+j]-t)*prev[p+j+1]
            b = x[p+i+1+j]-x[p+j]
            new[p+j] = a/b
        bsplines.append(new)

    return bsplines[k][p]
```

```python
def Bspline_loop2(t,k,p,x):
    """
    Alternative version, nearly identical to the one above,
    but instead of storing the function values for all degrees k
    in a list of dictionaries, we only store for k-1 (prev) and
    k (new) in two lists. """

    if k == 0:
        return Bspline0(t,p,x)

    prev = [Bspline0(t,p+k_,x) for k_ in range(k+1)]
    for i in range(1,k+1):
        new = []
        for j in range(k+1-i):
            a = (t-x[p+j])*prev[j]+(x[p+i+1+j]-t)*prev[j+1]
            b = x[p+i+1+j]-x[p+j]
            new.append(a/b)
        prev = new

    return new[0]


x = {-1:0,0:1.0,1:1.3,2:2.0,3:2.5,4:3.0,5:3.1,6:3.5,7:4.0}
t = 1.5
p=1

for k in range(4):
    print(f'Rec degree {k}: {Bspline(t,k,1,x)}')
    print(f'Loop 1 degree {k}: {Bspline_loop1(t,k,1,x)}')
    print(f'Loop 2 degree {k}: {Bspline_loop2(t,k,1,x)}')
```

16 Skal ikke besvares, til bruk under sensuren

This question is not to be answered, but is used by the examiners during the marking of the exam, to add the points from the midterm exam.

Not to be answered.

Format

Words: 0

Maximum marks: 25