Forside

UNIVERSITETET I OSLO
Det matematisk-naturvitenskapelige fakultet
Written exam in IN1900
2023 Fall

Date and time: Monday 4/12 from 09:00 to 13.00 (4 hours)

Permitted aides: None
A calculator is available in Inspera

It is important that you read this information carefully before you start answering the questions.

If you need to zoom in the exam set, hold ctrl and push + eller - on the numeric keyboard.

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 13 questions that should be answered in total, and in total 75 points available on these questions. Question 14 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. On multiple choice questions you get the same score (0) for a wrong answer and for not answering at all, so you should always mark an answer.

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 should 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 Hva skrives ut?

What is printed in the terminal window when the following code is run?

```
a = []
for i in range(3):
    a.append([i,i+1])

print(a[-1])
```

Select one alternative:

- [2, 3] ✔
- [3, 4]
- 4
- 3

Maximum marks: 2

2 Hva skrives ut?

What is printed in the terminal window when the following code is run?

```
import numpy as np

class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c0 + self.c1 * x + self.c2 * x**2

class Line(Parabola):
    def __init__(self, c0, c1):
        super().__init__(c0, c1, 0)

line1 = Line(1,1)
print(line1(1), isinstance(line1, Parabola))
```

Select one alternative:

- TypeError: 'Line' object is not callable
- 2 False
- 2 True ✔
- 2

Maximum marks: 2

3 Sortere en liste

We wish to sort a list L from the smallest to the largest value. The list contains only numbers. Below you find two alternative code segments for performing this task. Which of the codes is/are correct?

Alternative 1:

```
for i in range(len(L)):
    for k in range(len(L)-i-1):
        if L[k+1] <= L[k]:
            L[k], L[k+1] = L[k+1], L[k]
```

Alternative 2:

```
for i in range(len(L)):
    min_idx = i
    for k in range(i+1,len(L)):
        if L[k] < L[min_idx]:
            min_idx = k
    L[i], L[min_idx] = L[min_idx], L[i]
```

Select one alternative:

- ○ Alternative 2 is correct
- ○ Alternative 1 is correct
- ○ Both codes are correct ✔
- ○ None of the codes are correct

Maximum marks: 2

4 Lesing av fil

We have stored precipitation data in a file precip.txt, with the following contents:

```
Month Precip  Normal
Jan   100.5   58.0
Feb   45.9    46.0
Mar   72.6    41.0
Apr   99.7    48.0
May   17.0    60.0
Jun   39.9    80.0
```

Here the middle column is the measured precipitation and the right column is the normal precipitation for the given month. There are no empty lines in the file.

The code below is supposed to read this file and count the number of months when the measured precipitation exceeded the normal:

```
count = 0
with open('precip.txt', 'r') as infile:
    for line in infile:
        words = line.split()
        count += float(words[1]) > float(words[2])
```

Unfortunately the code does not work as intended. Write the answer to the following two questions in the text field below.

a) In which line will the code stop and print an error message?

b) Can you suggest a change in the code to make it work as intended?

Fill in your answer here

1

a) The code will stop on the line “count += float(…)”, when it attempt to read the first line of the file. In this line words[1] has the value “Precip”, and trying to convert this to a number with float will give a ValueError.

b) Inserting a infile.readline() before the loop will make the code work, since it will then skip the processing of the first line.

Maximum marks: 6

5 Hvilke hører sammen?

We have the following two lists: **x = [-1,0,1]** and **y = [0, 1, [0], [1], [[0], [1]]]**. On the left in the table below you find six expressions, and on top of the table you find six possible results from evaluating these expressions. Find the ones that fit together.

Please match the values:

	0	[0]	[[1]]	[[0]]	[1]	1
y[x[-2]:x[-1]]	○	○ ✔	○	○	○	○
y[x[0]][x[0]]	○	○	○	○	○ ✔	○
y[sum(x)]	○ ✔	○	○	○	○	○
y.index(1)	○	○	○	○	○	○ ✔
y[x[0]][x[-1]:]	○	○	○ ✔	○	○	○
y[3]	○	○	○	○	○ ✔	○

Maximum marks: 3

6 Feilhåndtering (exceptions)

We have the following program stored in the file multiply.py:

```
import sys
data = [0, 1, 2, 3, 4]

try:
    assert len(sys.argv) >= 3
    ind1 = int(sys.argv[1])
    ind2 = int(sys.argv[2])
    result = data[ind1] * data[ind2]
except IndexError:
    print("Error A")
    exit()
except AssertionError:
    print("Error B")
    exit()
except ValueError:
    print("Error C")
    exit()
```

We try to run the program from the command line with the alternatives shown below. For each row, choose what will be printed by the program, where "No output" means that nothing is printed.

Please match the values:

	Error A	No output	Error B	Error C
python multiply.py v1 v2	○	○	○	○ ✔
python multiply.py 2 3 5 7 9	○	○ ✔	○	○
python multiply.py 5 7	○ ✔	○	○	○
python multiply.py 12	○	○	○ ✔	○
python multiply.py 1 2	○	○ ✔	○	○
python multiply.py	○	○	○ ✔	○

Maximum marks: 3

7 Diskontinuerlig funksjon

The Heaviside function is defined as follows:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

a) Assume that the argument x is a number (not a number or a list) and implement the Heaviside function as a Python function **heaviside(x)**.

b) Write a test function which tests **heaviside(x)**, for at least two different values of the argument x.

c) Write a new implementation of the Heaviside function, which works when the argument x is a numpy array or a list. For this function you can assume that the argument is always a list or array, not a scalar. The function shall return a list or array of the same length as the input argument. Call the function **heaviside2(x)**.

Fill in your answer here

```
1
```

Solution at the end of the exam set.

Maximum marks: 9

Solution at the end of the exam set

8 Sum implementert som en klasse

The hyperbolic sine function sinh(x) can be approximate by a series:

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

We want to implement a Python class **SinHyp**, which calculates this series. We write the class in a file/module named hyperbolic.py, and want it to support the following use:

from hyperbolic import SinHyp
sinh_approx = SinHyp(N=4)
print(sinh_approx(x=1.0))
print(sinh_approx)

This code should give the following output:
1.1752011684303352
Approximate sinh, N = 4
(The first line is the value of the sum for N=4 and x=1.0)

a) Write the code for the class **SinHyp.** The operation ! (factorial) is found in the **math** module with the name **factorial**.

b) As we increase N, the approximation of sinh(x) improves. This can be demostrated by computing the distances

$$d_N = sinh(x) - \sum_{k=0}^{N} \frac{x^{2k+1}}{(2k+1)!}$$

between the true value sinh(x) and the approximate values, for different values of N (for instance for $N = 1, 2, \ldots, 30$) for a given value of x, and plotting the points (N, d_N). Write code for computing these distances and plotting them. You can assume that the true value of sinh(x) can be computed with the function **sinh(x)** in the **math** module.

Fill in your answer here

```
1
```

Solution at the end of the exam set.

Maximum marks: 8

9 Klasse for polynomer

A general polynomial of degree N can be written on the form

$$p(x) = \sum_{i=0}^{N} c_i x^i$$

Such a polynomial can be represented as a list of length *N+1*, where the element with index *i* is the coefficient in front of the term of degree *i* (c_i).

A class that uses this data structure to represent a polynomial can be implemented as follows:

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

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

The constructor takes a list of coefficients as input and stores the list as an attribute. The method **__call__** evaluates the polynomial for a given x.

a) We want the class to support the following use:

```
p1 = Polynomial([0,1,2])
p2 = Polynomial([0,0,1])
p3 = p1 + p2
```

Here, **p3** should be an instance of the class **Polynomial**, which represents the sum of p1 and **p2**. Explain how the class needs to be extended for this code to work. Which method must be added? What arguments should it accept, and what should it return? You do not need to write code, but explain the extension as precisely as possible.

b) Write the code for the extension you described in a) Remember that the two polynomials to be added do not necessarily have the same degree.

Fill in your answer here

```
1
```

Solution at the end.

Maximum marks: 8

10 Implementere et kunderegister

A company keeps a register of unpaid invoices in a text file *register.txt,* with the following format:

Invoice id. Customer id. Name Amount Due date
57893; 101456; Ole Olsen; 999.00 ; 31.01.2023
34216; 10343; Knut Knutsen; 1499.00; 21.01.2023
89134; 101456; Ole Olsen; 499.00; 20.01.2023

Each line contains information about a single invoice, but multiple invoices can be registered to the same customer, as indicated in the file above. There are no blank lines in the file. Your task is to make Python functions for reading and updating such a register.

a) Write a function **process_invoice(inv_line)**, which takes a string with information about an invoice as input, and returns a dictionary with the invoice information (Remark: in this sub-question you shall not use the file *register.txt*). You can assume that the string has the same format as a single line in the file given above (except for the header line). The dictionary shall have keys 'inv_id', 'cust_id', 'name', 'amount', and 'date', and the corresponding values shall be taken from the string. For instance, it shall be possible to use the function in the following way:

```
line = "57893;      101456;      Ole Olsen;      999.00 ;      31.01.2023"
invoice = process_invoice(line)
```

After this call, the dictionary **invoice** shall have the value

{'inv_id': 57893, 'cust_id': 101456, 'name': 'Ole Olsen', 'amount': 999.0, 'date': '31.01.2023'}

b) Write the function **read_register(filename)**, which reads a file with the same format as the file *register.txt* above, and returns a dictionary with the information in the file. This dictionary shall have one key/value pair for each customer. For instance, the file *register.txt* listed above contains information about two customers (one customer has two invoices), and the resulting dictionary should then contain two key/value pairs. Each key shall be the customer id and the corresponding value shall contain information about all the invoices for this customer. To achieve this, each value in the dictionary shall be a new dictionary with keys 'name' og 'invoices', where the value of 'name' is the customer name og 'invoices' is a new dictionary with the invoice id (inv_id) as key and a tuple with the amount and date as value. For example, if the file *register.txt* has the contents listed above, the function call

```
register = read_register('register.txt')
```

shall result in a dictionary with the following contents:

```
{101456: {'name': 'Ole Olsen', 'invoices': {57893: (999.0, '31.01.2023'), 89134: (499.0,
'20.01.2023')}}, 10343: {'name': 'Knut Knutsen', 'invoices': {34216: (1499.0,
'21.01.2023')}}}
```

Note that each customer only shows up once, but each customer can have one or more invoices. You can reuse the function **process_invoice** from sub-question a) to process the file.

Fill in your answer here

```
1
```

Solution at the end.

Maximum marks: 8

11 Differenslikninger

We have the following system of difference equations for $n \geq 0$:

$$x_n = x_{n-1} + ax_{n-1} - bx_{n-1}y_{n-1}, y_n = y_{n-1} + dbx_{n-1}y_{n-1} - cy_{n-1}.$$

The system is a variant of the famous Lotka-Volterra model, which describes the dynamics in a population of predators and prey, where x_n is the number of prey and y_n the number of predators at a given time t_n. The parameters *a,b,c* and *d* are positive constants.

Write a function **LotkaVolterra(N, a, b, c, d, x0, y0)** which computes the solution (x_n, y_n) of the difference equation above for $n = 1, 2, \ldots, N$. The other arguments to the function are the parameters *a,b,c,d* and the initial values x_0 og y_0. The function shall return the solution as a tuple (x, y), where x is a list (or array) with solution values $x_0, x_1, x_2, \ldots, x_N$ and y is a list (or array) with solution values $y_0, y_1, y_2, \ldots, y_N$.

Fill in your answer here

```
1
```

Solution at the end.

Maximum marks: 6

12 Fitzhugh-Nagumo-modellen I

The Fitzhugh-Nagumo model is a famous ODE-model that describes the electrical signal in cells like nerve cells or heart cells. The model is given by:

$$v'(t) = c_1 \cdot v(t) \cdot (1 - v(t)) \cdot (v(t) - a) - c_2 w(t),$$
$$w'(t) = b(v(t) - c_3 w(t))$$

The parameters c_1, c_2, c_3, a, b are constants that can be adjusted to control the behavior of the model.

a) Write a class that implements this model. The class shall have a constructor, which takes the five model constants as arguments and stores them as attributes, and a **__call__** method that implements the right hand side of the model. The **__call__** method must be defined so that the model can be solved with the solvers in the **ODESolver** class hierarchy (attached).

b) Write code to solve the model for the time interval $t = 0$ to $t = 400$ with 400 time steps, using the **RungeKutta4** class from the **ODESolver** hierarchy, and plot the solutions v and w in the same window. Use the parameter values $c_1 = 0.26, c_2 = 0.1, c_3 = 1.0, a = 0.13, b = 0.013$, and initial values $v(0) = 0.25, w(0) = 0$.

Fill in your answer here

```
1
```

Solution at the end.

Maximum marks: 10

13 Fitzhugh-Nagumo-modellen II

The SciPy library includes a function **root(fun, x0)**, which can be used to solve non-linear equations on the form $f(x) = 0$. The function can be imported with

```
from scipy.optimize import root
```

and the two arguments to the function are as follows:

fun - a Python function that implements the mathematical function f(x). The function must take a single argument (x), which is an array of arbitrary length, and return an array of the same length as x.

x0 - start value (initial guess) for the solution. This should be an array of the same length as the argument to the function f.

The function **root** returns the solution of the equation as an array **x**.

a) Write code that uses the **root** function to find an equilibrium point for the Fitzhugh-Nagumo model from the previous question, that is, values for $v(t)$ and $w(t)$ such that $v'(t) = w'(t) = 0$. You can use the same parameter values as in Question 12, and you can assume that an equilibrium point exists close to the initial values used in Question 12. You can also assume that you write the code in the same file as the code from Question 12, so there is no need to import or duplicate the code here.

b) Equilibrium points for an ODE system can be stable or unstable. A bit simplified, an equilibrium point v_0, w_0 is stable if the solution of the ODE system approaches v_0, w_0 for initial values $v_0 + \epsilon, w_0 + \epsilon$, where ϵ is a small number. If v_0, w_0 is an unstable equilibrium point, the initial values $v_0 + \epsilon, w_0 + \epsilon$ will give a solution that moves away from v_0, w_0. Explain how you can use the code you wrote in question 12 to determine if the equilibirum point you found in a) is stable or unstable. You do not have to write code, but explain as precisely as possible how you would perform this task.

Fill in your answer here

```
1
```

Solution at the end.

Maximum marks: 8

14 Poeng fra midtveis, til bruk for sensor

This question is not to be answered. It is used by the graders to include points from the midterm exam.

Fill in your answer here

Format

Words: 0

Maximum marks: 25

Question 12
Attached

```
import numpy as np


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

    def set_initial_condition(self, u0):
        if np.isscalar(u0):              # 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, t_span, N):
        t0, T = t_span
        self.dt = (T - t0) / N
        self.t = np.zeros(N + 1)  # N steps ~ N+1 time points
        if self.neq == 1:
            self.u = np.zeros(N + 1)
        else:
            self.u = np.zeros((N + 1, self.neq))

        msg = "Please set initial condition before calling solve"
        assert hasattr(self, "u0"), msg

        self.t[0] = t0
        self.u[0] = self.u0

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

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

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

```python
#Question 7:
#a)
def heaviside(x):
    if x < 0:
        return 0
    return 1

#b)
def test_heaviside():
    args = [-1, 0, 1]
    expected = [0, 1, 1]

    for x, e in zip(args, expected):
        assert heaviside(x) == e

#c)
def heaviside2(x):
    result = []
    for x_ in x:
        if x_ < 0:
            result.append(0)
        else:
            result.append(1)
    return result


#Question 8:
from math import factorial, sinh
import matplotlib.pyplot as plt

#a)
class SinHyp:
    def __init__(self, N):
        self.N = N

    def __call__(self,x):
        s = 0
        for k in range(self.N+1):
            s += x**(2 * k + 1)/factorial(2 * k + 1)
        return s

    def __str__(self):
        return f'Approximate sinh, N = {self.N}'


sinh_approx = SinHyp(N=4)
print(sinh_approx(x=1.0))
print(sinh_approx)

#b)
dist = []
for N in range(1,31):
```

```
    sinh_approx = SinHyp(N)
    dist.append(sinh(1.0) - sinh_approx(1.0))

plt.plot(dist)
plt.show()


#Question 9:
#a)
"""
The class needs to be extended with a special method __add__. It needs to take one
Polynomial
as argument, in addition to "self", and must return a Polynomial instance
representing
the sum of the two polynomials.
"""

#b)
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

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

    def __add__(self, other):
        # return self + other
        # start with the longest list and add in the other:
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
        else:
            coeffsum = other.coeff[:] # copy!
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]
        return Polynomial(coeffsum)



#Question 10:
#a)
def process_invoice(inv_line):
    words = [w.strip() for w in inv_line.split(';')]
    inv_id = int(words[0])
    cust_id = int(words[1])
    cust_name = words[2]
    amount = float(words[3])
    date = words[4]
    invoice =
{'inv_id':inv_id,'cust_id':cust_id,'name':cust_name,'amount':amount,'date':date}
    return invoice
```

```
#b)
def read_register(filename):
    customers = {}
    with open(filename) as infile:
        infile.readline()
        for line in infile:
            invoice = process_invoice(line)
            cust_id = invoice['cust_id']
            inv_id = invoice['inv_id']
            info = (invoice['amount'],invoice['date'])
            if cust_id in customers:
                customers[cust_id]['invoices'][inv_id] = info
            else:
                customers[cust_id] = {'name':invoice['name'],'invoices':
{inv_id:info}}

    return customers


#Question 11
def LotkeVolterra(N,a, b, c, d,x0,y0):
    x = [0]*(N+1); y = [0]*(N+1)
    x[0] = x0; y[0] = y0
    for i in range(N):
        x[i+1] = x[i] + a * x[i] - b * x[i] * y[i]
        y[i+1] = y[i] + d * b * x[i] * y[i] - c * y[i]

    return x,y


#Question 11:
from ODESolver import RungeKutta4
import matplotlib.pyplot as plt
import numpy as np

#a)
class FHN:
    def __init__(self, c1, c2, c3, a, b):
        self.c1, self.c2, self.c3 = c1, c2, c3
        self.a, self.b = a, b

    def __call__(self, t, u):
        c1, c2, c3 = self.c1, self.c2, self.c3
        a, b = self.a, self.b
        v, w = u
        dv = c1 * v * (1 - v) * (v - a) - c2 * w
        dw = b * (v - c3 * w)
        return dv, dw

#b)
fhn = FHN(0.26, 0.1, 1.0, 0.13, 0.013)
solver = RungeKutta4(fhn)
solver.set_initial_condition([0.25,0])
t,u = solver.solve([0,400],400)
```

```
plt.plot(t,u[:,0])
plt.show()

#Question 13:
#a)
"""
We want to reuse the right-hand-side function of the FHN class,
since this defines the function for which we want to find the roots.
However, this function was defined to be used with the ODESolver class
and therefore takes two arguments, while root expects a function that
takes only one. We can solve this in many ways, for instance writing
a sub-class to FHN or write a new function which calls the FHN instance.
"""
from scipy.optimize import root
init = np.array([0.13,0])

#Simplest solution: wrap an existing FHN instance in a function
def fhn_fun(u):
    return fhn(0.0,u)

solution = root(fhn_fun,init)

#Slightly more advanced, avoids using a global variable
class FHN_eq(FHN):
    def __call__(self,u):
        return super().__call__(0,u)

fhn_eq = FHN_eq(0.26, 0.1, 1.0, 0.13, 0.013)
solution = root(fhn_eq,init)
print(solution)

#b)
"""
One way to determine if an equilibrium point is stable
or unstable is to start with the solution from a)
(i.e., the equilibrium point), perturb it slightly and use it
as an initial condition. If the solution returns to the initial condition
it is a stable equilibrium point, if it doesn't it is unstable.
"""

#possible (simple) solution
eps = 1.0e-2
i_vals = solution.x + eps   #add eps to both components of x
solver.set_initial_condition(i_vals)
t,u = solver.solve([0,400],400)
plt.plot(t,u)
plt.show()
```