UiO : Universitetet i Oslo

INF1100 1 Grunnkurs i programmering for naturvitenskapelige anvendelser

Oppgaver	Oppgavetype	Vurdering
Front page	Dokument	Automatisk poengsum
1 What is printed?	Skriveoppgave	Manuell poengsum
2 What is printed?	Skriveoppgave	Manuell poengsum
3 What is printed?	Skriveoppgave	Manuell poengsum
4 What is printed?	Skriveoppgave	Manuell poengsum
5 What is printed?	Skriveoppgave	Manuell poengsum
6 Python functions	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
7 Python classes	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
8 File read and write	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
9 Random numbers	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
10 Random numbers	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
11 Taylor series	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
12 ODESolver	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
13 ODESolver	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum
14 Modelling with ODEs	*CONTENT_QUESTION_QTI2_CODE_EDITOR*	Manuell poengsum

INF1100 1 Grunnkurs i programmering for naturvitenskapelige anvendelser

Starttidspunkt:	14.12.2016 15:00	PDF opprettet	16.12.2016 09:47
Sluttidspunkt:	14.12.2016 19:00	Opprettet av	Helge Engeseth Kleivane
		Antall sider	15

Section 1; what is printed?

Front page

UNIVERSITY OF OSLO
Faculty of mathematics and natural sciences

Examination in: INF1100 - Introduction to programming with scientific applications
Day of examination: December 14th 2016
Examination hours: 15.00-19.00
Attachments: 1 (ODESolver.pdf)
Permitted aids: None

- Read through the complete exercise set before you start solving the individual exercises. If you miss information in an exercise, you can provide your own reasonable assumptions as long as you explain them in detail.
- Most of the exercises result in short code where there is little need for comments, unless you do something complicated or non-standard. In that case, comments should convey the idea behind the program constructions such that it becomes easy to evaluate the solution.
- Many exercises ask you to “write a function”. A main program calling the function is then not required, unless it is explicitly stated. You may, in these types of exercises, also assume that necessary modules are already imported outside the function. On the other hand, if you are asked to write a complete program, explicit import of modules must be a part of the solution.
- The maximum possible score on this exam is 75 points. There are 14 questions in total, and the number of points is specified for each individual exercise.

UNIVERSITETET I OSLO
Det matematisk-naturvitenskapelige fakultet
Eksamen i:
INF1100 - Grunnkurs i programmering for naturvitenskapelige anvendelser
Eksamensdag: 14. desember 2016
Tid for eksamen: 15.00-19.00
Vedlegg: 1 (ODESolver.pdf)
Tillatte hjelpemidler: Ingen.

- Les gjennom hele oppgavesettet før du begynner å løse oppgavene. Dersom du savner opplysninger i en oppgave, kan du selv legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så fall rede for forutsetningene og antagelsene du gjør.
- De fleste oppgavene resulterer i ganske kort kode hvor det normalt er lite behov for kommentarer med mindre man gjør noe komplisert eller ikke-standard (i så fall skal kommentarene forklare ideene bak program- konstruksjonene slik at det blir lett å vurdere koden).
- En oppgave kan be deg skrive en funksjon. Et hovedprogram der man kaller funksjonen er da ikke påkrevd, med mindre det er eksplisitt angitt. I denne typen oppgaver kan du også anta at nødvendige moduler er importert utenfor funksjonen. I andre tilfeller der du blir spurt om å skrive et program, er eksplisitt import av moduler en viktig del av besvarelsen.
- Maksimal poengsum på eksamen er 75 poeng. Det er totalt 14 oppgaver, og poengsummen for hver oppgave er oppgitt.

1 OPPGAVE

What is printed?

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

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
for i in range(2,10,5):
    print 'Letter %d is %s' %(i,alphabet[i])
```

2 OPPGAVE

What is printed?

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

```
class Y:
    def __init__(self,v0):
        self.v0 = v0
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0

y = Y(5)
print y
```

3 OPPGAVE

What is printed?

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

```
import sys
try:
    x = float(sys.argv[1])
    coeff = [float(s) for s in sys.argv[2:]]
except IndexError:
    print 'You need to provide at least two command-line arguments.'
    sys.exit(1)
except ValueError:
    print 'Cannot convert argument to float.'
    sys.exit(1)

poly_val = 0
for i in range(len(coeff)):
    poly_val += coeff[i]*x**i
```

```
print 'The value of the polynomial is %g' % poly_val
```

The code is in a file "poly_list.py", which is run by:
Terminal> python poly_list.py 1.0 3.5 2

4 OPPGAVE

What is printed?

A text file "summer.txt" has the following content (no blank lines):

```
Average  19.3
June     20.0
July     10.5
August   27.5
```

What is printed when the following code is run?

```
infile = open('summer.txt', 'r')
infile.readline()
total_rain = 0
months = []
for line in infile:
    months.append(line.split[0])
    total_rain += float(line.split[-1])
print 'The total rainfall from %s to %s was %.2f' %(month[0],month[-1],total_rain)
```

5 OPPGAVE

What is printed?

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

```
import numpy as np

def fibonacci(N):
    x = np.zeros(N+1, int)
    x[0] = 1
    x[1] = 1
    for n in range(2, N+1):
        x[n] = x[n-1] + x[n-2]
    return x

def test_fibonacci():
    expected = [1,1,2,3,5]
    computed = fibonacci(4)
    for e,c in zip(expected,computed):
        assert e == c

test_fibonacci()
```

Seksjon 2; Python programming

6 OPPGAVE

Python functions

A mathematical function is defined as follows:

$$x = \begin{cases} 0 & \text{for} \quad x < 0 \\ x^2 & \text{for} \quad 0 \leq x < 2 \\ 4 & \text{for} \quad x \geq 2 \end{cases}$$

Implement this function as a Python-function. Write the code for calling the function with *x=2* and printing the result to the screen.

7 OPPGAVE

Python classes

A class for a parabola is defined by the following code*:*

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

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

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
```

```
            s += '%12g %12g\n' % (x, y)
        return s
```

Implement a class named Line, for a linear function $y = c_0 + c_1 x$, as a subclass of Parabola. Reuse as much code as possible from Parabola. The class Line must support the following use:

```
>>> from Line import Line
>>> l = Line(1.0,2.5) #create a line y=1.0+2.5*x
>>> print l(0.5)
>>> print l.table(0,1,3)
           0           1
         0.5        2.25
           1         3.5
```

8 OPPGAVE

File read and write

A file contains lines with numbers separated by blanks. Write a Python-function sum_file(inputname,outputname) that reads such a file (with filename given in inputname), calculates the sum of the numbers on each line, and writes a new file (filename given in outputname) where each line contains the numbers read (on one line) followed by the sum of the numbers. Format the numbers in the file so that they appear as straight columns. The number of numbers on each line may vary. There are no blank lines in the file.

The file to be read may for instance look like this:

```
1.2500    3.00  4.50
2.25      4       4.50
3.25    5.00    0.50   0.5
4.250    6.2     1
```

The file to be written may then look like this:

```
  1.25   3.00   4.50   8.75
  2.25   4.00   4.50  10.75
  3.25   5.00   0.50    0.5   9.25
  4.25   6.20   1.00  11.45
```

9 OPPGAVE

Random numbers

Write a Python-function that estimates the probability of drawing two or more hearts when you draw five cards from a card deck (without putting any cards back). The deck is of regular type, with 52 cards out of which 13 are hearts.
Hint: for a list a, the command random.shuffle(a) (from the random module) will shuffle the elements of a in random order (the argument is changed "in-place"). Furthermore, the function a.pop() will return the first element in the list a, and remove it from the list.

10 OPPGAVE

Random numbers

Write a Python-funksjon that takes an integer N as input, simulates flipping a coin N times, and returns the number of "heads". The code is to be fully vectorized, which means there should be no loops or list comprehensions in Python.
Hint: numpy.random(low,high,size), where size is a tuple (n,N), returns an array of dimensions (n,N) containing random numbers in the interval low to high, including the boundaries. Furthermore, numpy.sum(a,axis) will return the sum of elements in the array a along the dimension axis.

Seksjon 3: difference equations and ODEs

11 OPPGAVE

Taylor series

The Taylor series that approximates the exponential function can be written as

$$e^x \approx \sum_{n=0}^{N} \frac{x^n}{n!}$$

A Python function to compute this series can be written as:

```
from math import factorial

def taylor_exp(x,N):
    s = 0
    x = float(x)
    for n in range(N+1):
        s += x**n/factorial(n)
    return s
```

The imlementation above is inefficient because of repeated calls to the function factorial(n). The code can be made more efficient by implementing the series as a difference equation. We can write the Taylor series as a system of difference equations on the form:

$$e_n = e_{n-1} + a_{n-1},$$

$$a_n = \frac{x}{n} a_{n-1},$$

with e_0=0 and a_0=1. Write a function taylor_exp_diffeq, which solves this system to compute the Taylor series. The input parameters and return value shall be exactly as for the function given above.

12 OPPGAVE

ODESolver

A differential equation, or a system of differential equations, written on the general form

$y'(t) = f(y,t), \quad y(0) = Y_0, \qquad (*)$

can be solved with methods implemented in the class hierarchy ODESolver. The complete Python code for the base class and two sub classes can be found in the attached pdf file. An alternative numerical method for solving equations on the form (*) is the explicit midpoint method. The method can be written on the form

$k_1 = \Delta t\, f(y_k, t_k),$

$k_2 = \Delta t\, f(y_k + \frac{1}{2}k_1, t_k + \frac{1}{2}\Delta t),$

$y_{k+1} = y_k + k_2,$

where y_k is the numerical approximation to the exact solution *y(t)* at time $t = t_k = k\Delta t$.

Write a sub class to the class ODESolver that implements the explicit midpoint method. The sub class is to be written in a separate file "Midpoint.py", so you have to import the class ODESolver from the file "ODESolver.py". Reuse as much code as possible from the base class ODESolver.

Denne oppgaven inneholder en PDF. Se neste side.

```
import numpy as np

class ODESolver(object):
    """
    Superclass for numerical methods solving scalar
    and vector ODEs

      du/dt = f(u, t)

    Attributes:
    t: array of time values
    u: array of solution values (at time points t)
    k: step number of the most recent solution
    f: callable object implementing f(u, t)
    """
    def __init__(self, f):
        # ensure self.f returns an array:
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # scalar ODE
            self.neq = 1
            U0 = float(U0)
        else:                            # ODE system
            U0 = np.asarray(U0)
            self.neq = U0.size
        self.U0 = U0

    def solve(self, time_points):
        """
        Compute solution u for t values in
        the list/array time_points.
        """
        self.t = np.asarray(time_points)
```

```
        n = self.t.size
        if self.neq == 1:  # scalar ODE
            self.u = np.zeros(n)
        else:              # ODE system
            self.u = np.zeros((n,self.neq))

        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
            self.u[k+1] = self.advance()
        return self.u[:k+2], self.t[:k+2]

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

class RungeKutta4(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t
        dt = t[k+1] - t[k]
        dt2 = dt/2.0
        K1 = dt*f(u[k], t[k])
        K2 = dt*f(u[k] + 0.5*K1, t[k] + dt2)
        K3 = dt*f(u[k] + 0.5*K2, t[k] + dt2)
        K4 = dt*f(u[k] + K3, t[k] + dt)
        u_new = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return u_new
```

13 OPPGAVE

ODESolver

Write a test function for the sub-class Midpoint from the previous question. You can assume the test function is in the same file as the Midpoint class, so no import is needed.
Hint: The explicit midpoint method, as well as most numerical methods for ordinary differential equations, can reproduce a linear solution on the form
$y(t) = at + b$
exactly (for arbitrary constants *a,b*). One can construct a differential equation with such a linear solution , for instance
$y'(t) = 5, \quad y(0) = 1,$
which has the solution
$y = 5t + 1$.
The class Midpoint shall reproduce this solution to machine precision.

14 OPPGAVE

Modelling with ODEs

This question presents a model for an outbreak of Ebola in Sierra Leone in 2014.
The population is divided into four groups; those that can be infected (S), those that are infected but have not yet developed the diseases, and can not yet infect others (E), those that are sick and can infect others (I), and those that have died from the disease (D). Let *S(t), E(t), I(t)* and *D(t)* be the number of people in each category. The following system of differential equations describes the dynamics of *S(t), E(t), I(t)* and *D(t)* in a time interval *[0,T]:*

$$S'(t) = -p(t)S(t)I(t),$$
$$E'(t) = p(t)S(t)I(t) - qE(t),$$
$$I'(t) = qE(t) - rI(t),$$
$$D'(t) = rI(t).$$

At time *t=0* we have the initial conditions $S(0) = S_0$, $E(0) = E_0$, $I(0) = 0$, $D(0) = 0$. The function p*(t)* and the constants *q* and *r* are assumed known. All constants and functions are >0.

Write a Python function SEID(S0,E0,p,q,r,T), which takes initial values S0, E0, the function p(t), constants q, r, and the end time T as input arguments. Use the class RungeKutta4 in the ODESolver hierarchy to solve

the differential equations. Let the time be given in days. Use ten time steps per day, so that the total number of time points for a simulation over $[0,T]$ is 10T+1. The function SEID shall return 5 arrays:

- t, which holds the time points t_k where the numerical solution is calculated,
- S, which contains $S(0), S(t_1), \ldots S(t_n)$,
- E, which contains $E(0), E(t_1), \ldots E(t_n)$,
- I, which contains $I(0), I(t_1), \ldots I(t_n)$,
- D, which contains $D(0), D(t_1), \ldots D(t_n)$.

We reason as follows to set the necessary parameters in the model. At the outbreak of the disease ($t=0$) the population in Sierra Leone was 5.48 million, and we want to study an extreme case where 10% of the population is infected and about to develop the disease. This gives S0=4.93 and E0=0.55. On average it took 10 days from the time of infection to the start of the disease, and 10.38 days from the start of the disease until the person died. This gives $q = 1/10$ and $r = 1/10.38$.
The function $p(t)$ describes the likelihood of an infected person getting interacting with and infecting a healthy, susceptible person. If nothing is done to slow spreading of the disease we can assume that this is constant, and for the disease outbreak we study it was estimated to $p=0.0233$.

Write the code for calling the function SEID with the given parameters and T=100. Also add code to plot $S(t)$, $E(t)$, $I(t)$ and $D(t)$ in the same window, with a legend for each curve.