UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Wednesday, December 14, 2016
Examination hours:	15.00 – 19.00.
This examination set consists of 15 pages.	
Appendices:	None.
Permitted aids:	None.

Make sure that your copy of the examination set is complete before you start solving the problems.

- 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 7 exercises, and the number of points for each exercise is given in the heading. Subexercises a), b), etc. count the same number of points.

(Continued on page 2.)

Section 1: What is printed

1 What is printed? (2pts)

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

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

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

2 What is printed? (2pts)

What is printed in the terminal window when this program is run? 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 What is printed? (2pts)

What is printed in the terminal window when the following program is run? 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)
```

(Continued on page 3.)

```
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 4
```

4 What is printed? (2pts)

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' %(months[0],months[-1],total_
```

5 What is printed? (2pts)

What is printed when the following program is run?

```
import numpy as np

def fibonacci(N):
    x = np.zeros(N+1, int)
    x[0] = 1
```

(Continued on page 4.)

```
    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()
```

Solutions questions 1-5:

```
Q1:
Letter 2 is c
Letter 7 is h
Q2:
v0*t - 0.5*g*t**2; v0=5
Q3:
The value of the polynomial is 5.5
Q4:
The total rainfall from June to August was 58.00
Q5:
```

On question 4, the original exam had a couple of bugs, which would cause an error message rather than the output listed above. Both answers were accepted on the exam. On Question 5, nothing is printed since the test passes.

Section 2: Python programming

6 Python functions (4 pts)

A mathematical function is defined as follows:

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

Implement this mathematical function as a Python function. Call the function for $x = 2$ and print the value to the screen.
Solution:

(Continued on page 5.)

```
def y(x):
    if x < 0:
        return 0
    elif x < 2:
        return x**2
    else:
        return 4

print y(2)
```

Alternative solution:

```
def y(x):
    return 0 if x < 0 else x**2 if x< 2 else 4

print y(2)
```

This function can be implemented in multiple ways. The two solutions above are just examples. Since the mathematical function is continuous, the use of $<$ *versus* $<=$ *is not important.*

7 Python classes (4 pts)

A class for a parabola $y = c_0 + c_1x + c_2x^2$ 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 `Line` for a linear function $y = c_0 + c_1x$ as a sub-class of the `Parabola` class. Reuse as much code from `Parabola` as possible. The class Line must support the following use:

(Continued on page 6.)

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

Solution:

```
from Parabola import *

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

This question also included a bug in the original exam. The line 2.25 above was missing, indicating there would be no output from print l(0.5) This exact behavior is quite difficult to achieve, but one can get close by altering the call method to return an empty string for instance. All answers including attempts at this will also get full score on the exam, as long as the rest of the subclass is correct.

8 File read and write (10 pts)

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
```

(Continued on page 7.)

Solution:

```
def sum_file(inputname,outputname):
    numbers = []
    with open(inputname,'r') as infile:
        for line in infile:
            row = [float(word) for word in line.split()]
            row.append(sum(row))
            numbers.append(row)

    with open(outputname,'w') as outfile:
        for row in numbers:
            line = ''
            for number in row:
                line += '%7.2f' %number
            line += '\n'
            outfile.write(line)


sum_file('innfil.txt','utfil.txt')
```

9 Random numbers (7 pts)

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.
Solution:

```
from random import shuffle

def two_hearts():
    N = 10000
    M = 0
    for i in range(N):
        deck = ['H','C','S','D']*13
        shuffle(deck)
        hearts = 0
        for j in range(5):
            if deck.pop() == 'H':
                hearts += 1
        if hearts >= 2:
```

(Continued on page 8.)

```
            M += 1
    return float(M)/N
```

Many alternative solutions exist. Here is one alternative using random integers instead of shuffling a list:

```
from random import randint

def two_hearts():
    N = 100000
    M = 0
    for i in range(N):
        hearts = 0
        cards_in_deck = 52
        for j in range(5):
            card = randint(1,cards_in_deck)
            cards_in_deck -= 1
            if card <= 13-hearts:
                hearts += 1
        if hearts >= 2:
            M += 1
    return float(M)/N
```

10 Random numbers (5 pts)

Write a Python function which takes a number N as input, simulates flipping a coin N times, and returns the number of heads. The code should be completely vectorized, i.e. there should be no explicit loops in Python. Hint: `numpy.random.random_integers(low,high,size)`, where `size` is a tuple `(n,N)`, returns an array of size `n,N` containing random integers between `low` and `high`. Furthermore, `numpy.sum(a,axis)` returns the sum of elements in array `a` along the dimension `axis`.
Solution:

```
import numpy as np

def flip_coin(N):
    flips = np.random.random_integers(0,1,(1,N))
    heads = np.sum(flips)
    return heads

print flip_coin(10)
```

In the exam this question also had a bug, as the function name random_integers had been left out of the hint. This made the hint suggest an incorrect use of

(Continued on page 9.)

numpy, but it could also easily be interpreted as suggesting the numpy.random.random function. This function only takes an array size as input, and returns an array of random floats between 0 and 1. Using this function, the solution may look like this:

```
import numpy as np

def flip_coin(N):
    flips = np.random.random((1,N))
    heads = np.sum(flips < 0.5)
    return heads

print flip_coin(10)
```

Because of the error in the hint, a number of different interpretations will be accepted and given full score for this question.

Difference equations and ODEs

11 Taylor series (10 pts)

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

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

A Python function for evaluating 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 implementation above is inefficient because of the repeated evaluations of the factorial. We can make the code more efficient by implementing it as a difference equation. The Taylor series can be written as a system of difference equations on the form:

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

(Continued on page 10.)

with $e_0 = 0$ and $a_0 = 1$. Write a python function `exp_diffeq(x,N)`, which solves this system of difference equations to evaluate the Taylor series. The input parameters and return value shall be exactly as for the function above.

Solution:

```
import numpy as np
from math import factorial

def taylor_exp_diffeq(x,N):
    x = float(x)
    e = 0
    a = 1
    for n in range(1,N+2):
        e = e + a
        a = x/n*a
    return e

#alternative solution with arrays
def taylor_exp_diffeq2(x,N):
    x = float(x)
    e = np.zeros(N+2)
    a = np.zeros(N+2)
    a[0] = 1
    for n in range(1,N+2):
        e[n] = e[n-1] + a[n-1]
        a[n] = x/n*a[n-1]
    return e[-1]
```

12 ODESolver (5 points)

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

$$y'(t) = f(y,t), \quad y(0) = Y_0, \tag{1}$$

can be solved by tools in a class hierarchy `ODESolver`. The complete Python code of the superclass and a subclass in this hierarchy is listed below. An alternative numerical method for solving equations on the form (1) is the explicit midpoint method. The method can be written on the form:

$$\begin{aligned} k_1 &= \Delta t\, f(y_k, t_k), \\ k_2 &= \Delta t\, f(y_k + \frac{1}{2}k_1, t_k + \frac{1}{2}\Delta t), \\ y_{k+1} &= y_k + k_2, \end{aligned}$$

(Continued on page 11.)

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

Write a subclass of `ODESolver` to implement the explicit midpoint method. The subclass code should be in a file `Midpoint.py`, so you have to import the class `ODESolver` from `ODESolver.py`. Reuse as much code as possible from the base class `ODESolver`.

Code for class `ODESolver` and a subclass `RungeKutta4`:

```
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)
```

(Continued on page 12.)

```
        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
```

Solution:

```
from ODESolver import ODESolver

class Midpoint(ODESolver):
    def advance(self):
        u,f,k,t = self.u,self.f,self.k,self.t

        dt = t[k+1] - t[k]
        k1 = dt*f(u[k],t[k])
        k2 = dt*f(u[k]+0.5*k1,t[k]+0.5*dt)
```

(Continued on page 13.)

```
        return u[k] + k2
```

13 ODESolver (8 pts)

Write a test function for the sub-class `Midpoint` from the previous question. Hint: the explicit midpoint method, as well as most methods for ordinary differential equations, can reproduce a linear solution $y(t) = at + b$ exactly (for arbitrary constants a and b). One can construct a differential equation with such a linear solution, e.g., $y'(t) = 5$, $y(0) = 1$, has solution $y = 5t + 1$. Class `Midpoint` should reproduce this solution to machine precision.

Solution (assuming the test function is in the same file as the sub-class:

```
def test_Midpoint():
    f = lambda u, t: 5
    u_ref = lambda t: 5.0*t+1.0
    solver = Midpoint(f)
    solver.set_initial_condition(1.0)

    u,t = solver.solve([0,1,2,3,4])
    tol = 1e-10
    for u_n,t_n in zip(u,t):
        assert abs(u_n-u_ref(t_n)) < tol
```

14 Modeling with ODEs (12 pts)

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 $[0, T]$:

$$S'(t) = -p(t)S(t)I(t), \tag{2}$$
$$E'(t) = p(t)S(t)I(t) - qE(t), \tag{3}$$
$$I'(t) = qE(t) - rI(t), \tag{4}$$
$$D'(t) = rI(t). \tag{5}$$

(Continued on page 14.)

At $t = 0$ 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 d, p, q must be known. The constants and functions are all > 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 $t_k = k\Delta t$, where the numerical solution is calculated,
- `S` containing $S(t_0), S(t_1), \ldots, S(t_n)$,
- `E` containing $E(t_0), E(t_1), \ldots, E(t_n)$,
- `I` containing $I(t_0), I(t_1), \ldots, I(t_n)$,
- `D` containing $D(t_0), D(t_1), \ldots, D(t_n)$.

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.
Solution:

```
from ODESolver import *
import matplotlib.pyplot as plt
import numpy as np


def SEID(S0,E0,p,q,r,T):
    def f(u,t):
        S,E,I,F = u
        return [-p(t)*S*I,p(t)*S*I-q*E, q*E-r*I,r*I]

    solver = RungeKutta4(f)
    solver.set_initial_condition([S0,E0,0,0])

    time = np.linspace(0,T,10*int(T)+1)
```

(Continued on page 15.)

```
    u,t = solver.solve(time)
    S,E,I,D = u[:,0],u[:,1],u[:,2],u[:,3]
    return time, S, E, I, D


p = lambda t: 0.0233
q = 1.0/10
r = 1.0/10.38
T = 100

t,S,E,I,D = SEID(4.93,0.55,p,q,r,T)
plt.plot(t,S,t,E,t,I,t,D)
plt.legend(['S','E','I','D'])
plt.show()
```

END