

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in: INF1100 — Introduction to programming with scientific applications

Day of examination: Wednesday, December 17, 2014

Examination hours: 09.00 – 13.00.

This examination set consists of 8 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 9 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.)

Exercise 1 (5 points)

What is printed in the terminal window when the programs below are run?

(a)

```
s = -2
for k in range(2, 5, 2):
    s += 2
print s
```

(b) The file `mydata.txt` contains the numbers

```
1.0  2.3  4.5
2.0  2.7 -2.0
3.0  2.9  4.8
4.0  3.2 -8.5
5.0  4.3  4.5
```

The program looks like

```
infile = open('mydata.txt', 'r')
infile.readline()
infile.readline()
for line in infile:
    a, b, c = [float(w) for w in line.split()]
    print 'a=%.2f b=%.2f c=%.2f' % (a, b, c)
```

(c)

```
def myfunc(a, b):
    s = 0
    for k in a:
        s += a[k]*b**k
    return s

print myfunc({1:-1, 3:1}, 3)
```

(d)

(Continued on page 3.)

```

def myfunc(a, b):
    c = a.copy()
    for k in b:
        if k in c:
            c[k] += b[k]
        else:
            c[k] = b[k]
    return c

print myfunc({1:-1, 3:1}, {1:3, 2:2})

```

(e)

```

from math import factorial
# factorial(N) is N! = N*(N-1)*(N-2)*...*2*1

def test_factorial():
    expected = 5*4*3*2*1
    computed = factorial(5)
    assert expected == computed

test_factorial()

```

Exercise 2 (5 points)

A piecewise constant function has the value 0 when its argument t is less than 5, and the value 0.4 otherwise. Implement such a mathematical function in a Python function. Make a test function for verifying the implementation (test for equal values with a tolerance).

Exercise 3 (20 points)

We want to solve the difference equation

$$y_i = ny_{i-1}, \quad y_0 = 1$$

for $i = 1, 2, 3, 4$, by the program

```

N = 3
from numpy import zeros
y = zeros(N+1, int)

```

(Continued on page 4.)

```

for i in range(1, N+1):
    y[i] = i*y[i-1]
    print i, y[i]

```

- What is printed by this program?
- How must the program be changed in order to solve the stated mathematical problem?
- Put the program in a Python function that returns the array y . Call the function to compute y_{14} . The function should not print anything.
- Write a test function for the Python function in c).

Exercise 4 (10 points)

The information about a polynomial

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

can be stored in a dictionary with i as key and c_i as value. For example, the polynomial $p(x) = -1 + 4x^3 + 20x^8$ is represented by the dictionary $\{0: -1, 3: 4, 8: 20\}$.

- Write a Python function `polyeval(x, p)` that takes a dictionary representation p of a polynomial and returns the value of the polynomial for the given x value. For example, `polyeval(2, {1:-1, 3:1})` should return the value $-1 \cdot 2^1 + 1 \cdot 2^3 = 6$.
- Write a Python function `polyadd(p, q)` that returns the dictionary representation of the sum of two polynomials whose dictionary representations are p and q . For example, `polyadd({1:-1, 3:1}, {1:3, 2:2})` should return $\{1:2, 2:2, 3:1\}$ since $-x + x^3 + 3x + 2x^2 = 2x + 2x^2 + x^3$.

Exercise 5 (5 points)

This is a continuation of Exercise 4. Now we want to represent a polynomial $p(x) = \sum_{i=0}^N c_i x^i$ by a class `Poly`. The class has one attribute: a dictionary representation of the polynomial (i.e., the collection of the coefficients and associated powers), as explained in Exercise 4. Equip the class with two special methods: `__call__` for evaluating the polynomial at a point x and `__add__` for adding two polynomials. The following interactive Python session should work (imagine that the class resides in a file `Poly.py`):

(Continued on page 5.)

```

>>> from Poly import Poly
>>> p1 = Poly({1:-1, 3:1})
>>> p1(3)
24
>>> p2 = Poly({1:3, 2:2})
>>> p3 = p1 + p2
>>> p3.coef
{1: 2, 2: 2, 3: 1}

```

It is possible to do this exercise even if you have not managed to write the functions in Exercise 4 (just assume that the functions from Exercise 4 exist).

Exercise 6 (5 points)

Extend class `Poly` from the previous exercise by a method `diff` that returns the derivative of the polynomial. The derivative of $p(x) = \sum_{i=0}^N c_i x^i$ becomes

$$p'(x) = \sum_{i=1}^N i c_i x^{i-1}$$

If we continue the interactive session in Exercise 5, we can do

```

>>> p4 = p3.diff()
>>> p4.__class__.__name__ # show that diff returns a Poly object
'Poly'
>>> p4.coef
{0: 2, 1: 4, 2: 3}

```

Exercise 7 (10 points)

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

$$y'(x) = f(y, x), \quad y(0) = Y_0,$$

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. One numerical solution technique for $y' = f(y, x)$ is the 2nd-order Runge-Kutta method:

$$\begin{aligned}
 k_1 &= \Delta x f(y_k, x_k), \\
 k_2 &= \Delta x f\left(y_k + \frac{1}{2}k_1, x_k + \frac{1}{2}\Delta x\right), \\
 y_{k+1} &= y_k + k_2,
 \end{aligned}$$

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

(Continued on page 6.)

a) Write a subclass of ODESolver to implement the 2nd-order Runge-Kutta method. The subclass code should be in a file RK2.py, separate from ODESolver.py (i.e., you need to import ODESolver).

b) Write a test function for class RK2. (Hint: the 2nd-order Runge-Kutta method, as well as most methods for ordinary differential equations, can reproduce a linear solution $y(x) = ax + b$ exactly (for arbitrary constants a and b). One can construct a differential equation with such a linear solution, e.g., $y'(x) = 2 + (y - (2x + 3))^2$, $y(0) = 3$, has solution $y = 2x + 3$. Class RK2 should reproduce this solution to machine precision.)

Code for class ODESolver and a subclass ForwardEuler:

```
import numpy as np

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

     $y'(x) = f(x, y)$ 

    Attributes:
    x: array of coordinates of the independent variable
    y: array of solution values (at points x)
    k: step number of the most recently computed solution
    f: callable object implementing f(y, x)
    """
    def __init__(self, f):
        self.f = lambda y, x: np.asarray(f(y, x), float)

    def set_initial_condition(self, Y0):
        if isinstance(Y0, (float,int)): # scalar ODE
            self.neq = 1
            Y0 = float(Y0)
        else: # system of ODEs
            Y0 = np.asarray(Y0) # (assume Y0 is sequence)
            self.neq = Y0.size
        self.Y0 = Y0

    def solve(self, x_points):
        """
        Compute solution y for x values in the list/array x_points.
        """
        self.x = np.asarray(x_points)
        n = self.x.size
        if self.neq == 1: # scalar ODEs
            self.y = np.zeros(n)
        else: # systems of ODEs
```

(Continued on page 7.)

```

        self.y = np.zeros((n,self.neq))

        # Assume that self.x[0] corresponds to self.Y0
        self.y[0] = self.Y0

        for k in range(n-1):
            self.k = k
            self.y[k+1] = self.advance()
        return self.y, self.x

class ForwardEuler(ODESolver):
    def advance(self):
        y, f, k, x = self.y, self.f, self.k, self.x
        dx = x[k+1] - x[k]
        return y[k] + dx*f(y[k], x[k])

```

Exercise 8 (10 points)

This exercise presents a model for the spreading of a flu. The population is divided into four groups: susceptibles (S) who can get the flu, infected (I) who have developed the flu and who can infect susceptibles, recovered (R) who have recovered from the flu and become immune, and the vaccinated (V). Let $S(t)$, $I(t)$, $R(t)$, and $V(t)$ be the number of people in category S, I, R, and V, respectively. The following differential equations describe how $S(t)$, $I(t)$, $R(t)$, and $V(t)$ develop in a time interval $[0, T]$:

$$S'(t) = -b(t)S(t)I(t) - p(t)S(t) + dR(t), \quad (1)$$

$$I'(t) = b(t)S(t)I(t) - qI(t), \quad (2)$$

$$R'(t) = qI(t) - dR(t), \quad (3)$$

$$V'(t) = p(t)S(t). \quad (4)$$

At $t = 0$ we have the initial conditions $S(0) = S_0$, $I(0) = I_0$, $R(0) = V(0) = 0$. The functions $b(t)$ and $p(t)$ as well as the constants $d > 0$ and $q > 0$ must be known.

Write a Python function `flu(S0, I0, b, q, d, p, T)` that takes the initial values S_0 and I_0 , the function $b(t)$, the parameter q , the parameter d , the function $p(t)$, and the end time T for the simulation as arguments. Use class `RK2` in the `ODESolver` hierarchy to solve the differential equations (if you did not manage to write class `RK2`, just assume that it exists). Let the time unit be days. Use five time steps per day such that the total number of time points for a simulation in $[0, T]$ is $5T + 1$. Five arrays should be returned from the function `flu`:

- `t` containing the time points $t_k = k\Delta t$, where the numerical solution is computed, $k = 0, 1, \dots, n$,

(Continued on page 8.)

- S containing $S(t_0), S(t_1), \dots, S(t_n)$,
- I containing $I(t_0), I(t_1), \dots, I(t_n)$,
- R containing $R(t_0), R(t_1), \dots, R(t_n)$.
- V containing $V(t_0), V(t_1), \dots, V(t_n)$.

We look at the spreading of the flu in small, closed population and reason as follows to set appropriate values of the parameters needed in the model. At $t = 0$ there are 1000 susceptibles and 2 infected. The value of $1/q$ reflects the average length of the disease, here taken as 7 days, so $q = 1/7$ (time t is measured in days). The function $b(t)$ measures how easily an infected person can infect a susceptible. This function is taken to be constant, equal to 0.001. $1/d$ is the average time before a recovered loses her/his immunity, and we take $d = 1/100$. The $p(t)$ function measures the effectiveness of vaccination. Suppose that vaccination takes place after k days such that $p = 0$ for $t < k$ and $p = 0.4$ for $t \geq k$.

Make a call to the function `flu` with the mentioned parameters, $T = 40$, and $k = 5$. Also add code for plotting $S(t)$, $I(t)$, $R(t)$, and $V(t)$ in the same figure with a legend for each curve.

Exercise 9 (5 points)

Write a Python function `dump(filename, t, S, I, R, V)` that can write five arrays of equal length, `t`, `S`, `I`, `R`, and `V`, to a file with name `filename`. Here is an example of how the numbers should be nicely formatted in a tabular fashion in the file:

0.0000	1000.0000	2.0000	0.0000	0.0000
0.2000	999.5659	2.3722	0.0620	0.0000
0.4000	999.0513	2.8133	0.1354	0.0000
0.6000	998.4416	3.3361	0.2223	0.0000
0.8000	997.7192	3.9555	0.3252	0.0000
1.0000	996.8636	4.6893	0.4471	0.0000
1.2000	995.8504	5.5581	0.5915	0.0000
1.4000	994.6510	6.5865	0.7625	0.0000
1.6000	993.2319	7.8031	0.9650	0.0000
1.8000	991.5536	9.2416	1.2048	0.0000
2.0000	989.5700	10.9415	1.4886	0.0000
2.2000	987.2272	12.9484	1.8244	0.0000
2.4000	984.4626	15.3159	2.2215	0.0000
2.6000	981.2035	18.1054	2.6911	0.0000
2.8000	977.3660	21.3881	3.2459	0.0000
3.0000	972.8538	25.2453	3.9009	0.0000

END