UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Thursday, December 17, 2015
Examination hours:	09.00 – 13.00.

This examination set consists of 9 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.)

Exercise 1 (5 points)

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

(a)

```
print '4' in '37.5 degrees'
```

(b)

```
q = -2
for k in range(2, 5, 2):
    q += 1
print q
```

(c)

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]

print q[1]
print q[-1][-1]
```

(d)

```
import sys
C = '20.0 degrees'

try:
    C = float(C)
except ValueError:
    print 'Cannot convert %s to float' %type(C)
    sys.exit(1)
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

(e)

```
def test_sum():
    expected = 1+2+3+4+5
    computed = sum(range(6))
    assert expected == computed

test_sum()
```

(Continued on page 3.)

Exercise 2 (5 points)

A piecewise linear function is defined as follows:

$$y = \begin{cases} -x & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Implement this mathematical function as a Python function. Make a test function for verifying the implementation (test for equal values with a tolerance).

Exercise 3 (10 points)

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

(a) The file `summer.txt` has the following content

```
Average  27.1
June     36.4
July     17.5
August   27.5
```

The program looks like

```
infile = open('summer.txt', 'r')
infile.readline()
for line in infile:
    month, rain = line.split()
    rain = float(rain)
    print 'In %s, total rainfall was %.2f' %(month,rain)
```

(b)

```
def add(a, b):
    return a + b

print add(1, 2)
print add([1,2,3], [0,1,2])
```

(c)

```
method1 = "ForwardEuler"
method2 = method1
method1 = "RK2"
print method2
```

(Continued on page 4.)

(d)

```
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
```

(e)

```
from random import randint
N = 1000
heads = 0
for i in range(N):
    result = randint(0,1)
    if result == 0:
        heads += 1
p = heads/N

print p
```

Exercise 4 (10 points)

(a) Write a Python function that takes a number n as input, and uses Monte Carlo simulation to estimate the probability of throwing at least one six when throwing n dice. The function shall return the estimated probability. Use a fixed value for the number of experiments in the Monte Carlo simulation.

(b) Write a vectorized version of the function in (a), 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` over the dimension `axis`.

(Continued on page 5.)

Exercise 5 (25 points)

A polynomial can be represented as a class, using a list to hold the coefficients of the polynomial. One implementation of such a class may look like this:

```
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):
        # Start with the longest list and add in the other
        if len(self.coeff) > len(other.coeff):
            result_coeff = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                result_coeff[i] += other.coeff[i]
        else:
            result_coeff = other.coeff[:] # copy!
            for i in range(len(self.coeff)):
                result_coeff[i] += self.coeff[i]
        return Polynomial(result_coeff)
```

(a) What is printed by the following interactive session?

```
>>> from Polynomial import Polynomial
>>> p1 = Polynomial([1,1,1])
>>> p2 = Polynomial([0,0,0,5])
>>> p3 = p1+p2
>>> print p3(1.0)
```

(b) The Taylor Polynomial of degree N for the exponential function e^x is given by

$$p(x) = \sum_{k=0}^{N} \frac{x^k}{k!}.$$

Write a python function `taylor_exp(N)`, where N is the number of terms in the Taylor polynomial. The function shall return a `Polynomial` instance (object) representing the Taylor polynomial $p(x)$. Recall that $k!$ is the factorial of k, and can be computed by the function `math.factorial(k)`.

(c) Write a test function `test_taylor_exp()` for the function in (b). Use a fixed (low) value of N in the test function, and compare the value of the polynomial returned by `taylor_exp` to a taylor polynomial derived by hand, for a single value of x.

(Continued on page 6.)

(d) Extend the `Polynomial` class with 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 from (a), we can do

```
>>> p4 = p3.diff()
>>> p4.__class__.__name__
'Polynomial'
>>> print p4.coeff
[1,2,15]
```

(e) Write a test function for the `diff` function in (d). As in (c) above, the test can be based on choosing a single value of x, and comparing the value of the polynomial returned by the `diff` function to the expected value. If you did not manage to write the `diff` function in (d), you can simply assume that it exists.

Exercise 6 (10 points)

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

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

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 $y' = f(y,t)$ is Kutta's third order method:

$$\begin{aligned}
k_1 &= \Delta t\, f(y_k, t_k), \\
k_2 &= \Delta t\, f(y_k + \frac{1}{2}k_1, t_k + \frac{1}{2}\Delta t), \\
k_3 &= \Delta t\, f(y_k - k_1 + 2k_2, t_k + \Delta t), \\
y_{k+1} &= y_k + \frac{1}{6}(k_1 + 4k_2 + k_3),
\end{aligned}$$

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

(Continued on page 7.)

(a) Write a subclass of `ODESolver` to implement the 3rd-order Kutta method. The subclass code should be in a file `Kutta3.py`, separate from `ODESolver.py` (i.e., you need to import `ODESolver`).

(b) Write a test function for class `Kutta3`. Hint: the 3rd-order Kutta 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., $y'(t) = 2$, $y(0) = 1$, has solution $y = 2t+1$. Class `Kutta3` should reproduce this solution to machine precision.

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

```
import numpy as np

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

      y'(t) = f(y, t)

    Attributes:
    t: array of coordinates of the independent variable
    y: array of solution values (at points t)
    k: step number of the most recently computed solution
    f: callable object implementing f(y, t)
    """
    def __init__(self, f):
        self.f = lambda y, t: np.asarray(f(y, t), 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, t_points):
        """
        Compute solution y for t values in the list/array t_points.
        """
        self.t = np.asarray(t_points)
        n = self.t.size
        if self.neq == 1:  # scalar ODEs
            self.y = np.zeros(n)
```

(Continued on page 8.)

```
        else:              # systems of ODEs
            self.y = np.zeros((n,self.neq))

        # Assume that self.t[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.t

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

Exercise 7 (10 points)

This exercise presents a model for the spreading of a disease. The population is divided into three groups: susceptibles (S) who can get the disease, infected (I) who have developed the disease and who can infect susceptibles, and recovered (R) who have recovered and become immune. Let $S(t)$, $I(t)$, and $R(t)$ be the number of people in category S, I, and R, respectively. We also consider people moving in and out of the population of interest (for instance moving to a geographical region), with a rate of entry Σ and exit μ. The following differential equations describe how $S(t)$, $I(t)$ og $R(t)$ develop in a time interval $[0, T]$:

$$S'(t) = \Sigma(t) - b(t)S(t)I(t) + dR(t) - \mu S(t), \tag{1}$$

$$I'(t) = b(t)S(t)I(t) - qI(t) - \mu I(t), \tag{2}$$

$$R'(t) = qI(t) - dR(t) - \mu R(t). \tag{3}$$

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

(Continued on page 9.)

Write a Python function `SIR(S0, I0, sigma, mu, b, q, d, T)` that takes the initial values `S_0` and `I_0`, the functions `sigma(t), b(t)`, the parameters `mu, q, d`, and the end time `T` for the simulation as arguments. Use class `RungeKutta4` in the `ODESolver` hierarchy to solve the differential equations Let the time unit be days. Use ten time steps per day such that the total number of time points for a simulation in $[0, T]$ is $10T + 1$. Four arrays should be returned from the function `SIR`:

- `t` containing the time points $t_k = k\Delta t$, where the numerical solution is computed, $k = 0, 1, \ldots, n$,
- `S` containing $S(t_0), S(t_1), \ldots, S(t_n)$,
- `I` containing $I(t_0), I(t_1), \ldots, I(t_n)$,
- `R` containing $R(t_0), R(t_1), \ldots, R(t_n)$.

We look at the spreading of the disease in a small 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 $1/1000$. We set the entry rate to $\Sigma = 10$ and the exit rate to $\mu = 1/100$. The value $1/d$ is the average time before a recovered loses immunity, and we take $d = 1/100$.

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

END