UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Friday, December 17, 2010
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.
- An exercise may ask you to "write a function". A main program calling the function is then not required, unless it is explicitly stated. In the function you write, you can 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 is an important part of the solution.
- The maximum possible score on the exam is 75 points. There are 8 exercises, and the number of points for each exercise is given in the heading.

(Continued on page 2.)

Exercise 1 (5 points)

What is printed by this program?

```
L = [(0,0), (1,0), (1,1), (0,2), (0,0)]
for p in L[2:-1]:
    print p[1]
```

Solution:

```
1
2
```

Exercise 2 (5 points)

What is printed by this program:

```
from math import sqrt

def f(x):
    return x**4

def g(x):
    return sqrt(sqrt(x))

q = 2
v = g(f(q))
print v
```

Solution:

```
2
```

Exercise 3 (10 points)

Implement the mathematical function

$$T(x,t) = e^{-qx}\cos(t - qx)$$

as a Python function `T(x, t)`. Also, write a main program that plots $T(x,t)$ as a function of x when $q = 1$, $x \in [0,3]$, and $t = 1$.
Solution:

(Continued on page 3.)

```
def T(x, t):
    return exp(-q*x)*cos(t - q*x)

from scitools.std import exp, cos, linspace, plot
q = 1
t = 1
x = linspace(0, 3)
y = T(x, t)
plot(x, y)
```

Exercise 4 (10 points)

Write a complete program for making a movie (animation) of the function $T(x,t)$ from Exercise 3. Each frame in the movie corresponds to a certain time t and shows a graph of T versus x. Let $q = 1$, $x \in [0, 3]$, and $t \in [0, 6\pi]$. Use a spacing between t values of $\pi/16$.

Hint: Let the frames of the movie be stored in files with names `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so forth, created by `'tmp_%04d.png' % c`, where `c` is an integer counter $(0, 1, 2, \ldots)$. The movie can then be created by

```
from scitools.std import movie
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='movie.gif')
```

To remove old frame files from a previous run of the program, you can use the code

```
import glob, os
for name in glob.glob('tmp_*.png'):
    os.remove(name)
```

Solution:

```
from scitools.std import *
# clean up old frames, i.e., tmp_*.png files:
import glob, os
for name in glob.glob('tmp_*.png'):
    os.remove(name)

def T(x, t):
    return exp(-q*x)*cos(t - q*x)

xmax = 3
q = 1
x = linspace(0, xmax, 1001)
```

(Continued on page 4.)

```
n = 6*16 + 1
t_values = linspace(0, 6*pi, n)
counter = 0
for t in t_values:
    y = T(x, t)
    plot(x, y, hardcopy='tmp_%04d.png' % counter)
    counter += 1
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='movie.gif')
```

Exercise 5 (10 points)

A file with name `density.dat` contains information on how the density of air varies with temperature. The file may look as follows:

```
# Density of air versus temperature (1 atm pressure)
# Column 1: temperature in Celsius degrees
# Column 2: density in kg/m^3
-10     1.341
 -5     1.316
  0     1.293
  5     1.269
 10     1.247
 15     1.225
 20     1.204
 25     1.184
 30     1.164
# Source: Wikipedia
```

Lines starting with # are comment lines. Blank lines are not allowed in the file. The first column contains temperatures in Celsius degrees while the second column contains the corresponding densities. Your task is to read this file in a Python program and write out a similar file where the temperature is given in Fahrenheit instead of Celsius degrees. The relation between Fahrenheit (F) and Celsius (C) degrees reads $F = 1.8C + 32$.

Solution:

```
infile = open('density.dat', 'r')
outfile = open('density2.dat', 'w')
for line in infile:
    if line.startswith('#'):
        line = line.replace('Celsius', 'Fahrenheit')
        outfile.write(line)
    else:
        C, density = [float(w) for w in line.split()]
        F = 1.8*C + 32
```

(Continued on page 5.)

```
        outfile.write('%5.1f  %10.4f' % (F, density)
infile.close()
outfile.close()
```

Exercise 6 (15 points)

A flip-coin game costs 1 NOK to play. You flip a coin five times. If heads come up three times or more, you get paid 3 NOK. Make a program that determines if you, in the long run, will earn money by playing this game.

Solution:

```
import random

def play():
    heads = 0
    for i in range(5):
        r = random.randint(1,2)
        if r == 1:
            heads += 1
    if heads >= 3:
        return True  # win

N = 100000
netincome = 0
for i in range(N):
    netincome -= 1   # pay cost in game no. i
    if play():
        netincome += 3  # get award for winning
netincome_per_game = netincome/float(N)
if netincome > 0:
    print 'Yes, you win in the long run,',
else:
    print 'No, you lose in the long run,',
print '%.1f NOK per game' % netincome_per_game
```

Exercise 7 (10 points)

Systems of ordinary differential equations of the form

$$u' = f(u,t)$$

can be solved by classes in the `ODESolver` class hierarchy shown below (this is a slightly simplified version of `ODESolver.py` from the course material).

(Continued on page 6.)

```
class ODESolver:
    """
    Superclass for numerical methods solving 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 recently computed solution
    f: callable object implementing f(u, t)
    dt: time step (assumed constant)
    """
    def __init__(self, f, dt):
        self.f = lambda u, t: numpy.asarray(f(u, t), float)
        self.dt = dt

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process

        self.u.append(numpy.asarray(u0, float))
        self.t.append(float(t0))
        self.k = 0  # time level counter

    def solve(self, T):
        """
        Advance solution from t = t0 to t = T, in steps of dt.
        """
        self.k = 0
        tnew = 0
        while tnew < T:
            unew = self.advance()

            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)


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

        unew = u[k] + dt*f(u[k], t)
```

(Continued on page 7.)

```
        return unew

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

The task now is to extend the `ODESolver` hierarchy with a new subclass implementing the following numerical method for $u' = f(u,t)$:

$$\begin{aligned} u_* &= u_k + \frac{1}{2}\Delta t\left(f(u_k, t_{k+1}) + f(u_k, t_k)\right), \\ u_{k+1} &= u_k + \frac{1}{2}\Delta t\left(f(u_*, t_{k+1}) + f(u_k, t_k)\right) \end{aligned} \qquad (1)$$

Here, u_k denotes $u(t)$ at the k-th time level, where $t = t_k = k\Delta t$. u_* is a help variable. Implement this numerical method in a subclass `RK2` of class `ODESolver`. The following minimalistic demonstration code for solving $u' = -u$, $u(0) = 1$, should work:

```
from ODESolver import RK2
def f(u, t):
    return -u

dt = 0.1
method = RK2(f, dt)
method.set_initial_condition(1)
u, t = method.solve(T=3)
```

Solution:

```
from ODESolver import ODESolver
class RK2(ODESolver):
    def advance(self):
        """Advance the solution one time step."""
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]
        u_star = u[k] + 0.5*dt*(f(u[k], t+dt) + f(u[k], t))
        unew = u[k] + 0.5*dt*(f(u_star, t+dt) + f(u[k], t))
        return unew

# test:
```

(Continued on page 8.)

```
def f(u, t):
    return -u

dt = 0.1
method = RK2(f, dt)
method.set_initial_condition(1)
u, t = method.solve(T=3)
from scitools.std import plot
plot(t, u, 'r-',
     t, exp(-t), 'b-')
```

Exercise 8 (10 points)

We have the following system of ordinary differential equations for two functions $x(t)$ and $y(t)$:

$$x'(t) = y(t)/Y - x(t)/X, \quad (2)$$
$$y'(t) = x(t)/X - y(t)/Y, \quad (3)$$

with initial conditions $x(0) = y(0) = 1$. Here, X and Y are two known parameters. Use a class in the `ODESolver` hierarchy from Exercise 7 to compute approximations to $x(t)$ and $y(t)$ for $t \in [0, 3000]$ when $X = 480$ and $Y = 2400$. Use a time step $\Delta t = 10$. Store the discrete values of x and y in arrays `x` and `y`. Describe how you can verify that the program works.
Solution:

```
class RHS:
    def __init__(self, X, Y):
        self.X, self.Y = float(X), float(Y)

    def __call__(self, u, t):
        # u is a 2-array [x, y] at time t
        x, y = u
        X, Y = self.X, self.Y
        return [y/Y - x/X, x/X - y/Y]

u0 = [1, 1]
f = RHS(X=480, Y=2400)
method = RK2(f, dt)
method.set_initial_condition(u0)
dt = 10
u, t = method.solve(T=3000)
x = u[:,0]
y = u[:,1]
```

(Continued on page 9.)

```
# can plot too (although the exercise does not ask for that):
from scitools.std import plot
plot(t, x, 'r-',
     t, y, 'b-',
     legend=('x', 'y'))
```

To verify the implementation, one can calculate by hand x_1, x_2, y_1, y_2, write out these values in the code and compare with the hand-calculated values. One can also for this special ODE system see that the solutions x and y become constant as $t \to \infty$, and the constants can be inserted back in the ODE system to check that they fit.

END