UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Tuesday, December 15, 2009
Examination hours:	14.30 – 17.30.
This examination set consists of 7 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 the exam is 100 points. There are 10 exercises, and the number of points for each exercise is given in the heading.

(Continued on page 2.)

Exercise 1 (5 points)

Write a Python function `h(y)` for evaluating the mathematical function

$$\frac{2}{\sqrt{\pi}}e^{-y^2}.$$

Also write a main program where you call the Python function.

Exercise 2 (10 points)

Write a Python function for solving the following system of two difference equations:

$$\begin{aligned} v_i &= v_{i-1} + dw_{i-1}, \\ w_i &= w_{i-1} + d(A\sin(c(i-1)d) - p|w_{i-1}|w_{i-1} - q\sin(v_{i-1})), \end{aligned}$$

for $i = 1, \ldots, N$. The initial conditions read $v_0 = s$ and $w_0 = 0$. The parameters d, A, c, p, and q in the equations are prescribed constants. The Python function should return the sequences $v_0, v_1, \ldots, v_N$ and $w_0, w_1, \ldots, w_N$.

Exercise 3 (10 points)

The function in Exercise 2 stores all the values $v_0, v_1, \ldots, v_N$ and $w_0, w_1, \ldots, w_N$. If the aim is to compute just v_N and w_N, only four values of the sequences are strictly necessary to store during the calculations. Make a new version of the function where you minimize the storage. Return the final values v_N and w_N.

Exercise 4 (10 points)

An integral

$$\int_a^b g(t)dt$$

can be approximated by the formula

$$\frac{b-a}{n+1}\sum_{i=0}^{n} g(t_i), \tag{1}$$

which arises from the Monte Carlo integration method. In this method, t_i are random variables uniformly distributed in the interval $[a, b]$. Write a Python function `MC(g, a, b, n=10000)` for computing an integral by the formula

(Continued on page 3.)

(1) (where the arguments `g`, `a`, `b`, and `n` correspond to the quantities $g(t)$, a, b, and n in the mathematical formula). Call the function to compute

$$\frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} dx \,.$$

Exercise 5 (10 points)

Vectorize the `MC` function from the previous exercise. That is, make sure that there are no explicit Python loops in the code. Assume that the `g(t)` function can accept an array `t` as argument and (in that case) return an array. (Hint: use `numpy.random.uniform(a, b, n)` and `numpy.sum`.)

Exercise 6 (10 points)

Modify the function `MC` from Exercise 4 such that it also writes a file with information on how the approximation evolves as we increase the number of function evaluations. To be specific, define

$$I_k = \frac{b-a}{k+1} \sum_{i=0}^{k} g(x_i) \tag{2}$$

as the approximation using $k+1$ function evaluations, and write to file the quantities $I(0)$, $I(1)$, ..., $I(n)$. (This can easily be done inside a loop in the `MC` function.) The resulting file, called `approx.dat`, looks as follows (only the first nine lines are shown here):

```
k:       0,   approximation=0.509454
k:       1,   approximation=0.806124
k:       2,   approximation=0.905143
k:       3,   approximation=0.915171
k:       4,   approximation=0.837735
k:       5,   approximation=0.867419
k:       6,   approximation=0.834705
k:       7,   approximation=0.849747
k:       8,   approximation=0.822398
```

Here, `approximation` corresponds to the value of I_k.

Exercise 7 (10 points)

Consider the following class and an associated main program:

(Continued on page 4.)

```
class Diffme:
    def __init__(self, g, dx=1E-7):
        self.g, self.dx = g, dx

    def __call__(self, x):
        g, dx = self.g, self.dx
        return (g(x+dx) - g(x-dx))/(2.*dx)

def h(t):
    return 3*t + 2

dhdt = Diffme(h)
print dhdt(1)
```

Explain the program flow. (You do not need to calculate a numerical value for `dhdt(1)`.)

Exercise 8 (10 points)

A cylindrical tank of radius R is filled with water to a height h_0. By opening a valve of radius r at the bottom of the tank, water flows out, and the height of water at time t, denoted by $h(t)$, decreases with time. The function $h(t)$ is governed by the differential equation

$$\frac{dh}{dt} = -\left(\frac{R}{r}\right)^{-2}\left(1+\left(\frac{r}{R}\right)^4\right)^{-1/2}\sqrt{2gh}\,. \tag{3}$$

Write a program for computing and plotting $h(t)$, using the class `RungeKutta4` from the `ODESolver` hierarchy of methods for ordinary differential equations (see code below). Let $r = 1$ cm, $R = 30$ cm, $g = 9.81$ m/s^2, $h_0 = 0.5$ m in the program example. Use a time step of $\Delta t = 10$ s and simulate for six minutes.

A (slightly simplified) version of class `ODESolver` and two subclasses are listed here for reference:

```
class ODESolver:
    """
    Superclass for numerical methods solving ODEs

      du/dt = f(u, t)

    Attributes:
    t: array of time values
```

(Continued on page 5.)

```
    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
        t = 0
        while t < T:
            unew = self.advance()

            self.u.append(unew)
            t = self.t[-1] + self.dt
            self.t.append(t)
            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)
        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)
```

(Continued on page 6.)

```
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
```

Exercise 9 (15 points)

The task in this exercise is to compute the solution $v(t)$ of the following second-order differential equation:

$$v'' + p|v'|v' + q\sin(v) = A\sin(ct), \quad v(0) = s,\ v'(0) = 0,$$

where $p \geq 0$, $q > 0$, $A \geq 0$, $c > 0$, and $s \in [0, \pi]$ are given constants. First we rewrite the equation as a system of two first-order equations

$$\begin{aligned} \frac{d}{dt}u^0 &= u^1, \\ \frac{d}{dt}u^1 &= A\sin(ct) - p|u^1|u^1 - q\sin(u^0)\,. \end{aligned}$$

The initial conditions for this system are $u^0(0) = s$ and $u^1(0) = 0$.

To solve the above first-order system, you shall apply a subclass, `ForwardEuler` or `RungeKutta4`, in the `ODESolver` hierarchy, listed in the previous exercise. These subclasses demand a right-hand side function `f(u, t)` defining the system of differential equations. Write a class for the relevant `f(u, t)` in this exercise. The class must have a `__call__` method and store p, q, A, c, and s as attributes. The equations are to be solved for $t \in [0, T]$. Show how to plot $v(t)$. You may set the following values of the parameters involved: $s = \pi/2$, $p = 0.1$, $q = 1$, $A = 1$, $c = 2$, time step $\Delta t = 2\pi/30$, and $T = 30\pi$. Figure 1 shows the corresponding solution $v(t)$ for these choices of parameters.

Exercise 10 (10 points)

One numerical method for solving an ordinary differential equation

$$u'(t) = f(u(t), t), \quad u(0) = U_0,$$

is the *midpoint* method:

$$u_{k+1} = u_{k-1} + 2\Delta t f(u_k, t_k), \tag{4}$$

where k is a time level, Δt the time step, and u_k is the approximation to u at time level k, i.e., when $t = t_k$. Equation (4) applies for $k = 1, 2, 3, \ldots$, while for $k = 0$ we use a simple Forward Euler approximation:

$$u_1 = u_0 + \Delta t f(u_0, t_0)\,. \tag{5}$$

(Continued on page 7.)

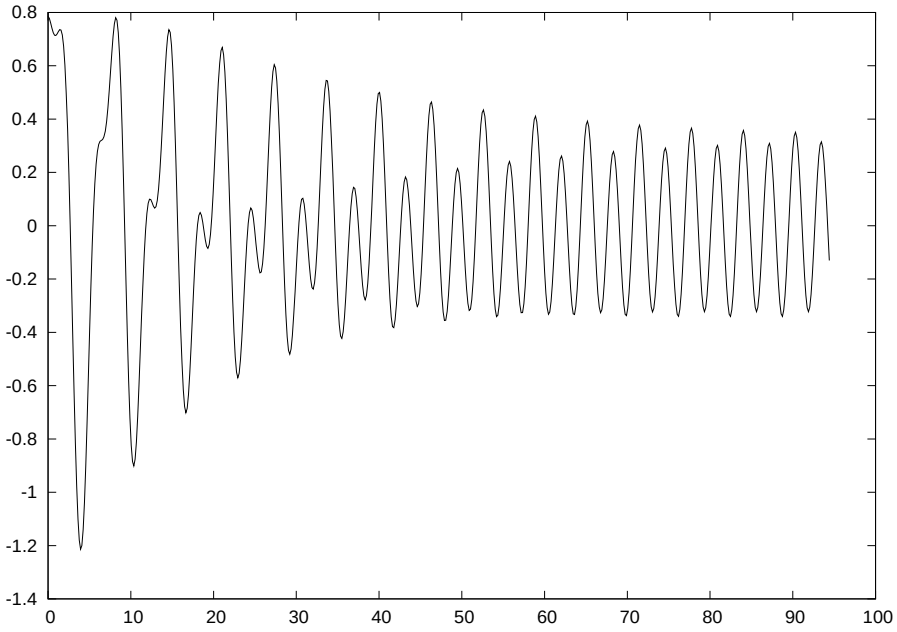

Figure 1: Plot of the solution of a 2nd-order differential equation.

The midpoint method defined by (4) and (5) is also valid for a system of ordinary differential equations when u and f are vectors.

Implement the midpoint method as a subclass of `ODESolver` (see Exercise 8 for relevant code).

END