UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Tuesday, December 18, 2012
Examination hours:	09.00 – 13.00.
This examination set consists of 13 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 12 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 `f(x)` that returns the value of the mathematical function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(x-m)^2}$$

Let m be a global value in the program. Also write a main program that writes out the value of $f(0.5)$ in the case $m = 0$.

Solution:

```
from math import sqrt, pi, exp

def f(x):
    return (1/sqrt(2*pi))*exp(-0.5*(x-m)**2)

m = 0
print f(0.5)
```

Exercise 2 (5 points)

Write a Python class `Gaussian` that can evaluate the function $f(x)$ given in Exercise 1. The following code computes $f(2.5)$ when $m = 2$ and demonstrates how the class works:

```
f = Gaussian(m=2)
value = f(2.5)
```

Solution:

```
from math import sqrt, pi, exp

class Gaussian:
    def __init__(self, m=0):
        self.m = m
    def __call__(self, x):
        return (1/sqrt(2*pi))*exp(-0.5*(x-self.m)**2)

f = Gaussian(m=2)
value = f(2.5)
```

(Continued on page 3.)

Exercise 3 (5 points)

Write a program that reads m and a series of x values from the command line and then writes out $f(x)$ for each x value, where f is given in Exercise 1. Abort the program if the command line does not contain m and at least one x value.
Here is an example on using the program (whose name is `Gaussian.py`):

```
Terminal> python Gaussian.py 1 1.1 1.2 1.3 2
2.661E-01
2.897E-01
3.123E-01
3.989E-01
Terminal> python Gaussian.py 1
Usage: Gaussian.py m x1 x2 ...
```

Make sure you format the printing of numbers such that each $f(x)$ value appears as shown.
Solution:

```
import sys
if len(sys.argv) < 3:
    print 'Usage: %s m x1 x2 ...' % sys.argv[0]
    sys.exit(1)

m = float(sys.argv[1])
for x in sys.argv[2:]:
    x = float(x)
    print '%.3E' % (f(x))
```

Exercise 4 (5 points)

What is printed by the programs below?

(a)

```
method1 = "Newton"
method2 = method1
method1 = "Bisection"
print method2
```

Solution:

```
Newton
```

(Continued on page 4.)

(b)

```
import numpy
# The coefficients for the Taylor polynomial for exp(x)
Taylor_coefficients = numpy.array(
    [1.0, 0.5, 0.16666666666666666, 0.041666666666666664,
     0.008333333333333333, 0.001388888888888889])
coeff = Taylor_coefficients
coeff[1] = 0
print Taylor_coefficients[:2]
```

Solution:

```
[ 1.   0]
```

(c)

```
Lagrange_points = [4, 2, 1, 6, 9]
del Lagrange_points[2:-1]
print Lagrange_points
```

Solution:

```
[4, 2, 9]
```

(d)

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

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

Solution:

```
3
[1, 2, 3, 0, 1, 2]
ForwardEuler
```

(e)

```
m = 3
k = 0
for i in range(m):
    for j in range(i-1, m):
         if i != j:
             k += 1
print k
```

(Continued on page 5.)

Solution:

```
6
```

Exercise 5 (5 points)

What is printed by the program below?

```
class PowerFunction:
    def __init__(self, a=1, p=2):
        print 'in PowerFunction constructor'
        self.data = {'a': a, 'p': p}

    def __call__(self, x):
        print 'in __call__'
        return self.data['a']*(x-1)**self.data['p']

def reduce(x):
    print 'in reduce, x:', x
    return sqrt(x)

def composite_function(x, f1, f2):
    print 'in composite_function'
    y = f2(f1(x))
    return y

from numpy import *
x = linspace(1, 3, 3)
p = PowerFunction()
u = composite_function(x, p, reduce)
for x_, u_ in zip(x, u):
    print '%.1f %.1f' % (x_, u_)
```

Solution:

```
in PowerFunction constructor
in composite_function
in __call__
in reduce, x: [ 0.  1.  4.]
1.0 0.0
2.0 1.0
3.0 2.0
```

(Continued on page 6.)

Exercise 6 (5 points)

A file with name `data.txt` contains three columns of numbers. The first two correspond to x and y coordinates on a curve, while the third contains uncertainty estimates of the y values given in percent. There is no number in the third column if no uncertainty estimate of corresponding y value has been computed.

The start of the file looks as this:

```
-1.000000     -0.76E-2    0.1432
-0.959184     -0.74E-2
-0.918367     -0.72E-2
-0.877551     -0.70E-2   -0.9078
```

Make a program that can plot the y coordinates in the second column against the x coordinates in the first column using a red line. Assume no empty lines in the file.

Solution:

```
import scitools.std as plt
# or
#import matplotlib.pyplot as plt
infile = open('data.txt')
x = []; y = []
for line in infile:
    numbers = line.split()
    x.append(float(numbers[0]))
    y.append(float(numbers[1]))
infile.close()
plt.plot(x, y, 'r-')
plt.show()
```

Exercise 7 (5 points)

Extend the program from Exercise 6 with statements that read the file again and visualize how the uncertainty estimates in the third column varies with the corresponding x values. Use small blue circles to visualize the data points.

Solution:

```
infile = open('data.txt')
x = []; uncertainty = []
for line in infile:
    numbers = line.split()
    if len(numbers) == 3:   # do we have a third column?
        x.append(float(numbers[0]))
        uncertainty.append(float(numbers[2]))
```

(Continued on page 7.)

```
infile.close()
plt.figure()
plt.plot(x, uncertainty, 'bo')
plt.show()
```

Exercise 8 (5 points)

The purpose of this exercise is to write a file like `data.txt` in Exercise 6. We have two Numerical Python arrays, `x` and `y`, and a list `uncertainty`, all of equal length. The values of the two arrays and the list are to make up the three columns in the file. Some of the elements in the list `uncertainty` have `None` as value, which indicates there is no uncertainty estimate and hence no value should be written to the file. Write a function `dump_data(filename, x, y, uncertainty)` that creates a file with name `filename` as described. Use the same format for real numbers as exemplified in the snippet from `data.txt` in Exercise 6.

Solution:

```
def dump_data(filename, x, y, uncertainty):
    outfile = open(filename, 'w')
    for x_, y_, u_ in zip(x, y, uncertainty):
        outfile.write('%13.6f %12.2E' % (x_, y_))
        print x_, y_, u_
        if u_ is not None:
            outfile.write(' %9.4f' % u_)
        outfile.write('\n')
    outfile.close()
```

Exercise 9 (10 points)

Somebody proposes the following game: You flip a coin 20 times, and if 15 or more heads show up, you receive 400 NOK, otherwise you have to pay 10 NOK. Will you earn money in the long run if you play the game? Write a program that applies Monte Carlo simulation to answer the question.

Solution:

```
import random
num_throws = 20
min_heads = 15
cost_per_game = 10
award_per_game = 400
N = 1000000      # no of experiments
M = 0            # no of successes
```

(Continued on page 8.)

```
money = 0
for e in range(N):
    num_heads = 0
    for i in range(num_throws):
        coin = random.random() > 0.5
        if coin:
            num_heads += 1
    if num_heads >= min_heads:
        M += 1
        money += award_per_game
    else:
        money -= cost_per_game
print money/float(N), float(M)/N

# Vectorized version (optional)
import numpy
heads = numpy.random.random(size=(N,num_throws)) > 0.5
num_heads = numpy.sum(heads, axis=1)
M = numpy.sum(num_heads >= min_heads)
money = M*award_per_game - (N-M)*cost_per_game
print money/float(N), float(M)/N
```

Exercise 10 (10 points)

Various numerical integration methods for time integrals

$$\int_0^T G(t)dt$$

can be implemented in a class hierarchy. All the numerical integration methods are written as

$$\int_0^T G(t)dt \approx \sum_{p=0}^{n} w_p G(t_p),$$

where $t_0, \ldots, t_n$ are given coordinates and $w_0, \ldots, w_n$ are given weights. Each method has its own choice of $t_0, \ldots, t_n$ and $w_0, \ldots, w_n$. In a superclass `TimeIntegral` we store the function to be integrated, $G(t)$, the limit T, and the parameter n. The method `compute` computes and returns the sum $\sum_{p=0}^{n} w_p G(t_p)$. Another method, `initialize`, computes $t_0, \ldots, t_n$ and $w_0, \ldots, w_n$ as two arrays, but this method must be implemented in various subclasses corresponding to various integration rules.

```
class TimeIntegral:
    """
    Compute an approximation to the integral of G(t)
```

(Continued on page 9.)

```
    from 0 to T using a numerical integration rule
    with n+1 function evaluations.
    """
    def __init__(self, G, T, n):
        self.G = G
        self.T = T
        self.n = n
        self.initialize()  # compute weights and points

    def initialize(self):
        """
        Compute weights self.w and points self.t
        as two arrays of length self.n+1.
        """
        raise NotImplementedError

    def compute(self):
        """Return the approximation of the integral."""
        s = 0
        for p in range(self.n+1):
            s += self.w[p]*self.G(self.t[p])
        return s
```

All the code above appears in a file `TimeIntegral.py`. The module `TimeIntegral` can therefore be imported in other programs.
The Trapezoidal rule,

$$\int_0^T G(t)dt \approx h\left(\frac{1}{2}G(0) + \frac{1}{2}G(T) + \sum_{i=1}^{n-1} G(ih)\right),$$

where $h = T/n$, can be implemented as the following subclass of class `TimeIntegral` in a separate file `methods.py`:

```
from TimeIntegral import TimeIntegral
from numpy import linspace, zeros

class Trapezoidal(TimeIntegral):
    def initialize(self):
        """
        Compute weights self.w and points self.t
        as two arrays of length self.n+1.
        """
        self.t = linspace(0, self.T, self.n+1)
        h = self.T/float(self.n)
        self.w = zeros(len(self.t)) + h
        self.w[0] = self.w[0]/2
        self.w[-1] = self.w[-1]/2
```

(Continued on page 10.)

The purpose of this exercise is to implement Monte Carlo integration as another subclass of `TimeIntegral`. The Monte Carlo integration method is defined through

$$\int_0^T G(t)dt \approx \frac{T}{n+1}\sum_{p=0}^{n} G(t_p),$$

where t_p are uniformly distributed random numbers in $[0,T]$. The weights are here constant: $w_p = T/(n+1)$. Add code for the Monte Carlo integration class in the `methods.py` file. Also add a function for testing that Monte Carlo integration gives exact result for a constant function, say $G(t) = 2.5$ for all $t \in [0,T]$.

Solution:

```
class MonteCarlo(TimeIntegral):
    def initialize(self):
        """
        Compute weights self.w and points self.t
        as two arrays of length self.n+1.
        """
        self.t = numpy.random.uniform(0, self.T, self.n+1)
        w = self.T/float(self.n+1)
        self.w = zeros(len(self.t)) + w

def _test(constant=2.5):
    def G(t):
        return constant

    T = 10
    n = 4
    integrator = MonteCarlo(G, T, n)
    print integrator.compute(), 'exact:', T*constant

_test()
```

Exercise 11 (10 points)

We have a pendulum of length L with a mass m at the end of a massless wire. At $t = 0$, the pendulum is at rest, making an angle $\theta \in (0,\pi)$ with the vertical (θ and all angles are measured in radians). We then release the pendulum and it moves back and forth driven by gravity. Air resistance will damp the motion and eventually bring the pendulum to rest.

At time t, the pendulum makes an angle $v(t)$ with the vertical. This angle can be computed by Newton's second law of motion, which takes the form of a differential equation:

$$mLv'' + c|v'|v' + mg\sin(v) = 0, \quad v(0) = \theta,\ v'(0) = 0.$$

(Continued on page 11.)

The constant c reflects the size of the air resistance. We want to solve this differential equation problem by the `ODESolver` software known from the course and listed below.

First we must rewrite the second-order differential equation for $v(t)$ as a system of two first-order equations:

$$\begin{aligned}\frac{d}{dt}u^{(0)} &= u^{(1)},\\ \frac{d}{dt}u^{(1)} &= -\frac{1}{mL}(c|u^{(1)}|u^{(1)} + mg\sin(u^{(0)}))\,.\end{aligned}$$

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

Make a class to represent the right-hand side of the differential equation system (known as the `f` object to constructors of classes in the `ODESolver` hierarchy). The physical parameters m, L, c, g, and θ should be attributes in the class.

Use the `RungeKutta4` method to solve the system. For simplicity, set all physical parameters to 1, except for g, which equals 9.81. A suitable time interval for simulation is $[0, T]$ with $T = 10P$, P being the time period of one oscillation, approximately given by $P = 2\pi/\sqrt{g}$. Choose 40 numerical time intervals during one oscillation: $\Delta t = P/40$.

Finally, plot v versus t as a blue curve. Mark the axis with t and v.

Here are the `ODESolver` and `RungeKutta4` classes:

```
import numpy as np

class ODESolver:
    """
    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 recently computed solution
    f: callable object implementing f(u, t)
    """
    def __init__(self, f):
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # scalar ODE
            self.neq = 1
            U0 = float(U0)
        else:                            # system of ODEs
```

(Continued on page 12.)

```
            U0 = np.asarray(U0)          # (assume U0 is sequence)
            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)
        n = self.t.size
        if self.neq == 1:  # scalar ODEs
            self.u = np.zeros(n)
        else:              # systems of ODEs
            self.u = np.zeros((n,self.neq))

        # Assume that self.t[0] corresponds to self.U0
        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, self.t

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)
        unew = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return unew
```

Solution:

```
class RHS:
    def __init__(self, m, L, g, c, theta):
        self.m, self.L, self.g, self.c, self.theta = \
                m, L, g, c, theta

    def __call__(self, u, t):
        m, L, g, c, theta = self.m, self.L, self.g, self.c, self.theta
        return [u[1],
                -1./(m*L)*(c*abs(u[1])*u[1] + m*g*sin(u[0]))]
```

(Continued on page 13.)

```
from ODESolver import RungeKutta4
from math import pi, sqrt, sin
import numpy as np
g = 9.81
f = RHS(m=1, L=1, g=g, c=1, theta=1)
P = 2*pi/sqrt(g)
dt = P/40
T = 10*P
n = int(T/dt)
time_points = np.linspace(0, T, n+1)
method = RungeKutta4(f)
method.set_initial_condition([f.theta, 0])
u, t = method.solve(time_points)

v = u[:,0]
from scitools.std import plot
plot(t, v, 'b', xlabel='t', ylabel='v')
```

Exercise 12 (5 points)

Implement the Forward Euler method in the class `ODESolver` hierarchy from the previous exercise. The Forward Euler method for an ODE or ODE system of the form $u' = f(u, t)$ can be written as

$$u_{k+1} = u_k + (t_{k+1} - t_k)f(u_k, t_k),$$

where t_k is the time at step number k, and u_k is an approximation to u at t_k.
Solution:

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t
        dt = t[k+1] - t[k]
        unew = u[k] + dt*f(u[k], t[k])
        return unew
```

END