# UNIVERSITETET I OSLO

## Det matematisk-naturvitenskapelige fakultet

| | |
|---|---|
| Examination in: | INF1100 — Introduction to programming with scientific applications |
| Day of examination: | Friday, December 16, 2011 |
| Examination hours: | 09.00 – 13.00. |
| This examination set consists of 11 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 80 points. There are 10 exercises, and the number of points for each exercise is given in the heading.

*(Continued on page 2.)*

## Exercise 1 (3 points)

What is printed by the following program?

```
for i in range(2, 4):
    print i
    for j in range(i-1, i+1):
        for k in range(j-1, j):
            if i != j:
                print j, k
```

Solution:

```
2
1 0
3
2 1
```

## Exercise 2 (2 points)

Let `elements` be some Python list. Write code that picks out a random element in the list and removes the element from the list.
Solution:

```
import random

# mylist is some list

# Shuffle list first and draw first element
mylist = random.shuffle(mylist)
item = mylist[0]
del mylist[0]

# Or use pop (equiv. to the last two statements)
mylist = random.shuffle(mylist)
item = mylist.pop(0)

# Or draw a random index
i = random.randint(0, len(mylist)-1)
item = mylist[i]
del mylist[i]
```

*(Continued on page 3.)*

```
# Or use random.choice and list.remove
item = random.choice(mylist)
mylist.remove(item)
```

## Exercise 3 (5 points)

Given a dictionary of the form

```
colors = {'red': 10, 'yellow': 100,
          'green': 41, 'blue': 88}
```

write a Python function `dict2list` that takes a dictionary `colors` as above as argument and returns a list of strings where the string `c` occurs exactly `colors[c]` times. Let `c` run over all keys in the `colors` dictionary. Here is a sample session using the function:

```
>>> colors = {'red': 3, 'yellow': 1, 'purple': 2}
>>> dict2list(colors)
['red', 'red', 'red', 'purple', 'purple', 'yellow']
```

Solution:

```
def dict2list(colors):
    colors_list = []
    for color in colors:
        for i in range(colors[color]):
            colors_list.append(color)
    return colors_list
```

## Exercise 4 (10 points)

Consider the following game. You flip a coin twice and win if you get one tail and one head. Write a program that applies Monte Carlo simulation for estimating the probability of winning. (Monte Carlo simulation means simulating the experiment on the computer a large number of times.)

Solution:

```
import random, sys
N = int(sys.argv[1])
M = 0

for e in range(N):
    c1 = random.randint(1, 2)
    c2 = random.randint(1, 2)
```

*(Continued on page 4.)*

```
    if c1 != c2:
        M += 1
print float(M)/N
```

## Exercise 5 (10 points)

Generalize the program in Exercise 4 to the case where you flip the coin $n$ times and win if you get at least twice as many heads as tails. Put the code in a function that takes $n$ and $N$ as arguments and returns the probability, where $N$ is the number of experiments in the Monte Carlo simulation. Read the $N$ value from the command line. Make a plot of the probability versus $n$ for $n = 3, 6, 9, 12, 15$. Mark each point in the plot by a symbol (for instance a circle). Write the value of $N$ in the title of the plot.

Solution:

```
import random

def simulate(n, N=10000):
    M = 0
    for e in range(N):
        c = [random.randint(1, 2) for i in range(n)]
        tails = c.count(1)
        heads = c.count(2)
        if heads >= 2*tails:
            M += 1
    return float(M)/N

import sys
from scitools.std import plot
try:
    N = int(sys.argv[1])
except IndexError:
    print 'Give N on the command line'
    sys.exit(1)

n_values = [3, 6, 9, 12, 15]
prob = [simulate(n, N) for n in n_values]
plot(n_values, prob, 'ro', title='N=%d' % N)
```

Typical values of `prob` are 0.50, 0.34, 0.26, 0.19, 0.15.

## Exercise 6 (10 points)

We have a hat of 8 red balls, 2 yellow balls, 6 green balls, and 9 black balls. What is the probability of getting (at least) a yellow and a red ball when

*(Continued on page 5.)*

drawing four balls (without replacement) from the hat? Write a program that applies Monte Carlo simulation to estimate the probability. (Hint: Use code and ideas from Exercises 2, 3, and 4.)
Solution:

```
import random

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random.randint(0, len(hat)-1)
    color = hat[index]
    del hat[index]
    return color, hat

def new_hat(colors):
    hat = []
    for color in colors:
        for i in range(colors[color]):
            hat.append(color)
    return hat

n = 4
N = 100000
colors = {'red': 8, 'yellow': 2, 'green': 6,
          'black': 9}

# run experiments:
M = 0  # no of successes
for e in range(N):
    hat = new_hat(colors)
    balls = []           # the n balls we draw
    for i in range(n):
        color, hat = draw_ball(hat)
        balls.append(color)
    #if balls.count('yellow') >= 1 and \
    #   balls.count('red') >= 1:
    if 'yellow' in balls and 'red' in balls:
        M += 1
print 'Probability:', float(M)/N
```

*(Continued on page 6.)*

## Exercise 7 (10 points)

We consider approximation of an integral $\int_0^T g(t)dt$ by some numerical integration rule of the form

$$\int_0^T g(t)dt \approx \sum_{i=0}^{n} w_i f(t_i),$$

where $w_0, \ldots, w_n$ and $t_0, \ldots, t_n$ are weights and points of the rule. The following class implements the computations:

```
from numpy import dot

class IntegralApproximation:
    def __init__(self, T, n):
        self.T, self.n = T, n
        self.t, self.w = self.set_weights_points()

    def __call__(self, g, vectorized=True):
        return self.vectorized_code(g) if vectorized else \
               self.scalar_code(g)

    def scalar_code(self, g):
        s = 0
        for i in range(len(self.w)):
            s += self.w[i]*g(self.t[i])
        return s

    def vectorized_code(self, g):
        return dot(self.w, g(self.t))

    def set_weights_points(self):
        raise NotImplementedError(
           'no set_weights_points method in class %s' \
           % self.__class__.__name__)
```

This class cannot be used for any real integration computation since it does not set the points and weights of the rule to be used. Subclasses are meant to define points and weights through the `set_weights_points` method.

We want to use a Monte Carlo integration rule where the points are random coordinates in the integration interval and where all the weights are equal to the length of the integration interval divided by the number of integration points. Write such a subclass and demonstrate how to use it to integrate

$$\int_0^{10} e^{-t/5} \sin^2(2\pi t)dt$$

in a vectorized fashion.

Solution:

*(Continued on page 7.)*

```
from numpy import zeros, random, sin, exp, pi

class MonteCarloInt(IntegralApproximation):
    def set_weights_points(self):
        # must use vectorized drawing of numbers
        # since the exercise wants a vectorized version
        weight = float(self.T)/(self.n+1)
        w = zeros(self.n+1) + weight
        t = random.uniform(0, self.T, self.n+1)
        return t, w

def g(t):
    return exp(-t/5.)*sin(2*pi*t)

integrator = MonteCarloInt(10, n=2000000)
I = integrator(g, vectorized=True)
print I
```

## Exercise 8 (10 points)

The result of some computation is a set of points $(x, y)$ on a curve. This set of points is stored in a file with the the $x$ coordinates in the first column and the $y$ coordinates in the second column. More precisely, the file format looks like this:

```
# File with (x, y) data
#
x=0.102871   y=8.12134
x=0.113526   y=7.98211
x=0.132912   y=2.67152
```

The file may contain some comment lines in the beginning, starting with `#` at the very beginning of the line. Make a function that takes the filename as argument and returns the $x$ and $y$ data as two arrays.
Solution:

```
def read(filename):
    x = []
    y = []
    infile = open(filename, 'r')
    for line in infile:
        if line.startswith('#'):
            continue
        words = line.split()

        xi = words[0].split('=')[1]
```

*(Continued on page 8.)*

```
        yi = words[1].split('=')[1]
        # Alternative:
        xi = words[0][2:]
        yi = words[1][2:]

        x.append(float(xi))
        y.append(float(yi))
    from numpy import array
    return array(x), array(y)

x, y = read('tmp.dat')
print x
print y
```

## Exercise 9 (10 points)

Suppose the curve data in Exercise 8 represent some quantity $y$ that oscillates with $x$. We are interested in locating all the local maxima of the curve and the $x$ distances between the maxima (these distances reflect the period of oscillations, while the maxima reflect the amplitude of the oscillations). Write a function taking the $x$ and $y$ coordinates as array arguments and returning the maxima points and the $x$ distances between them. Note that a local maximum takes place at `x[k]` if `y[k-1] < y[k] > y[k+1]`.

Show in a Python program how you can generate points on some function curve and call the function to compute local maxima and the distances between them. Write out the largest and smallest $y$ value of the maxima points.

Solution:

```
def findmax(x, y):
    maxima = []
    for k in range(1, len(y)-1, 1):
        if y[k-1] < y[k] > y[k+1]:
            maxima.append((x[k], y[k]))
    # Distances between maxima:
    dist = []
    for k in range(len(maxima)-1):
        dist.append(maxima[k+1][0] - maxima[k][0])
    return maxima, dist

from numpy import cos, linspace, pi
# Use a simple curve where we know the maxima
# (here all integer x) and distances (here 1)
def f(x):
    return cos(2*pi*x)
```

*(Continued on page 9.)*

```
L = 10
n = 101
x = linspace(0, L, n)
y = f(x)

maxima, dist = findmax(x, y)
# Turn maxima to array for easy analysis
from numpy import asarray
maxima = asarray(maxima)
ymaxima_max = maxima[:,1].max()
ymaxima_min = maxima[:,1].min()

# Alternative method, working with maxima as list of list
ymaxima = [y_ for x_, y_ in maxima]
ymaxima_max = max(ymaxima)
ymaxima_min = min(ymaxima)
print ymaxima_min, ymaxima_max
```

## Exercise 10 (10 points)

The differential equation for a pendulum subject to gravity forces and air resistance, and with an initial angle $\theta \in (0, \pi)$, is given by

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

Here, $m > 0$, $L > 0$, $c > 0$, $g > 0$, and $\theta$ are given constants. We want to solve this problem by the `ODESolver` software known from the course and listed below.

First we must rewrite the equation as a system of two first-order equations:

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

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

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

(b) 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 $\Delta t = P/40$.

*(Continued on page 10.)*

(c) Plot $v$ versus $t$. 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
            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):
```

*(Continued on page 11.)*

```
            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]))]

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, xlabel='t', ylabel='v')
```

END