# UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

| | |
|---|---|
| Examination in: | INF1100 — Introduction to programming with scientific applications |
| Day of examination: | Tuesday, December 16, 2008 |
| Examination hours: | 14.30 – 17.30. |

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.
- 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.

## Exercise 1 (5 points)

Write a function even(N) that returns a list of all even numbers $2, 4, 6, \ldots, N$.
Solution:

```
def even(N):
    return range(2, N+1, 2)

# or
def even(N):
    numbers = 2
    numbers_list = []
```

*(Continued on page 2.)*

```
    while numbers <= N:
        numbers_list.append(numbers)
        numbers += 2
    return numbers_list

print even(5)
```

*(Continued on page 3.)*

## Exercise 2 (10 points)

A file with name `mydat.txt` contains two columns of numbers, corresponding to $x$ and $y$ coordinates on a curve. The start of the file looks as this:

```
-1.000000          -0.761594
-0.959184          -0.743913
-0.918367          -0.725124
-0.877551          -0.705190
```

Make a program that can plot the $y$ coordinates in the second column against the $x$ coordinates in the first column. There are no empty lines in the file.
Solution:

```
infile = open('mydat.txt', 'r')
x = []; y = []
for line in infile:
    words = line.split()
    x.append(float(words[0]))
    y.append(float(words[1]))
infile.close()
from scitools.std import plot
plot(x, y)
```

## Exercise 3 (10 points)

The formula for a line going through the points $(x_0, y_0)$ and $(x_1, y_1)$ takes the form

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)\,. \tag{1}$$

The purpose of this exercise is to write a class `Line` that can be used as follows in an interactive Python session:

```
>>> from Line import Line
>>> point1 = (0,-1)
>>> point2 = (2,4)
>>> line = Line(point1, point2)
>>> line(0)   # compute y corresponding to x=0
-1
>>> line(1)
1.5
```

That is, the constructor takes two points on the line and a `__call__` method can compute the $y$ value corresponding to a given $x$. The file containing

*(Continued on page 4.)*

class `Line` must be written so that it can be used as a module. Write the complete module file. The module file should have a test block containing the test from the interactive session above.
Solution:

```
class Line:
    def __init__(self, p1, p2):
        x0, y0 = p1
        x1, y1 = p2
        # be careful with potential integer division:
        self.a = float(y1-y0)/(x1-x0)
        self.b = y0 - self.a*x0

    def __call__(self, x):
        return self.a*x + self.b

if __name__ == '__main__':
    point1 = (0,-1)
    point2 = (2,4)
    line = Line(point1, point2)
    print 'value at x=0;', line(0)
    print 'value at x=1:', line(1)
```

## Exercise 4 (10 points)

Consider the following game. You pay 1 unit of money to throw 16 dice. If the sum of the dice is larger than 70, you win 75 units of money. Make a computer program that can help you decide whether you will win or lose money in the long run by playing this game.
Solution:

```
N = 10000   # no of experiments
money = 0   # money won in total
ndice = 16  # no of dice to throw in each game
sum_crit = 70  # critical sum to win a game
award = 75  # money won in one game
cost = 1    # cost of one game

import random
for i in range(N):
    money -= cost
    eyes = [random.randint(1, 6) for i in range(ndice)]
    if sum(eyes) > sum_crit:
        money += award
print 'Money after %d games: %d' % (N, money)
```

*(Continued on page 5.)*

## Exercise 5 (10 points)

Integrals of the form

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

can be approximated by the Monte Carlo integration method:

$$\int_0^T G(t)dt \approx \frac{T}{n+1}\sum_{p=0}^{n} G(t_p), \qquad (2)$$

where $t_0, t_1, \ldots, t_n$ are random numbers uniformly distributed in the interval $[0, T]$.

Write a function `MonteCarlo(G, T, n)` that returns an approximation to the integral above using the method (2). The mathematical function $G(t)$ is assumed to be implemented in a Python function `G(t)`. Use the `uniform` function in the built-in Python module `random` to draw the random numbers.

Write a main program that calls the `MonteCarlo` function for computing

$$\int_0^{2\pi} \sin^3 t\, dt$$

with $n = 100000$.

Solution:

```
import random
def MonteCarlo(G, T, n):
    s = 0
    for i in range(n+2):
        t = random.uniform(0, T)
        s += G(t)
    return T/float(n+1)*s

def f(t):
    return sin(t)**3

from math import sin, pi
print MonteCarlo(f, 2*pi, n=10000)
```

## Exercise 6 (10 points)

The Midpoint method for solving an ordinary differential equation

$$\frac{dy}{dt} = f(y, t), \quad y(0) = Y$$

consists in first taking a Forward Euler step:

$$y_1 = y_0 + \Delta t f(y_0, 0),$$

*(Continued on page 6.)*

where $y_0 = Y$ is the initial condition at $t = 0$ and $\Delta t$ is the time step length. The next steps in the Midpoint method are computed by the formula

$$y_{k+1} = y_{k-1} + 2\Delta t f(y_k, t_k), \quad k = 1, 2, 3, \ldots$$

$y_k$ is a short notation for $y(t_k)$ and $t_k = k\Delta t$.
The purpose of the exercise is to write and test a function `midpoint(f, Y, N, dt)` that returns the solution of an ordinary differential equation using the Midpoint method. The right-hand side function $f(y, t)$ of the differential equation is represented by a Python function `f(y, t)`; `Y` is the initial value ($Y$); `N` is the number of steps to be calculated; and `dt` is the time step length ($\Delta t$). The `midpoint` function should return two arrays: the solution $(y_0, y_1, \ldots, y_N)$ and the corresponding time values $(t_0, t_1, \ldots, t_N)$.
Exemplify the use of the `midpoint` function by solving the differential equation

$$\frac{dy}{dt} = -Ay, \quad y(0) = 1,$$

where $A$ is a positive constant to be specified on the command line. Read also $N$ and $\Delta t$ from the command line. Visualize the numerical solution and the exact solution ($y(t) = e^{-At}$) in the same plot.
Solution:

```
def Midpoint(f, Y, N, dt):
    """Integrate y'=f(y,t), y(0)=Y in N steps of size dt."""
    y = []; t = []   # y[k] is the solution at time t[k]
    y.append(Y)
    t.append(0)
    # the first step is a Forward Euler step:
    ynew = y[0] + dt*f(y[0], t[0])
    y.append(ynew)
    t.append(dt)

    # Midpoint formula for the rest of the time levels:
    for k in range(2, N+1):
        ynew = y[k-1] + 2*dt*f(y[k], t[k])

        y.append(ynew)
        tnew = t[-1] + dt
        t.append(tnew)
    return numpy.array(y), numpy.array(t)

def f(y, t):
    return -A*y

A = float(sys.argv[1])
dt = float(sys.argv[2])
N = int(sys.argv[3])
Y = 1
```

*(Continued on page 7.)*

```
y, t = Midpoint(f, Y, N, dt)
from scitools.std import *
y_exact = exp(-A*t)
plot(t, y, 'r-',
     t, y_exact, 'b-',
     legend=('numerical', 'exact'),
     title="Solution of y'=-Ay, y(0)=1 by the Midpoint method",
     hardcopy='tmp.eps')
```

## Exercise 7 (10 points)

The purpose of this exercise is to implement the Midpoint method defined in the previous exercise in a class hierarchy representing numerical methods for ordinary differential equations. The superclass in this hierarchy, called `ODESolver`, stores $f(y,t)$ and $\Delta t$ in the constructor, and provides a method `initcond` for setting the initial condition and a method `integrate` for carrying out $N$ steps in the solution method. Here is the complete code of class `ODESolver`:

```
import numpy

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

      dy/dt = f(y, t)

    Attributes:
    t: array of time values
    y: array of solution values (at time points t)
    k: step number of the most recently computed solution
    f: callable object implementing f(y, t)
    dt: time step (assumed constant)
    """
    def __init__(self, f, dt):
        self.f = f
        self.dt = dt

    def method(self):
        """Advance solution one time step."""
        raise NotImplementedError

    def initcond(self, Y):
        self.y = []    # y[k] is solution at time t[k]
```

*(Continued on page 8.)*

```
        self.t = []    # time levels in the solution process

        self.y.append(Y)
        self.t.append(0)
        self.k = 0  # time level counter

    def integrate(self, N):
        """Advance solution N steps forward in time."""
        for i in range(N):
            ynew = self.method()

            self.y.append(ynew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.y), numpy.array(self.t)
```

Subclasses implement various numerical methods by providing their specific version of the method called `method`, which advances the solution one step forward in time. For example, the Forward Euler scheme may be implemented in a separate file as

```
from ODESolver import ODESolver

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

        ynew = y[k] + dt*f(y[k], t)
        return ynew
```

Implement the Midpoint scheme from the previous exercise in a subclass `Midpoint`. Exemplify the use of the `Midpoint` class by solving the same problem as in the previous exercise, i.e., $dy/dt = -Ay$, $y(0) = 1$. Set $A = 1$, $\Delta t = 0.1$ and $N = 50$. Print the computed $y_N$ value and the exact value $e^{-AN\Delta t}$.

Solution:

```
class Midpoint(ODESolver):
    def method(self):
        y, dt, f, k, t = \
           self.y, self.dt, self.f, self.k, self.t[-1]

        if k == 0:
            # the first step is a Forward Euler step:
            ynew = y[k] + dt*f(y[k], t)
        else:
```

*(Continued on page 9.)*

```
            # the Midpoint formula:
            ynew = y[k-1] + 2*dt*f(y[k], t)
        return ynew

def f(y, t):
    return -A*y

A = 1
dt = 0.1
N = 50
Y = 1
method = Midpoint(f, dt)
method.initcond(Y)
y, t = method.integrate(N)
from math import exp
print 'Numerical vs exact:', y[-1], exp(-A*N*dt)
```

## Exercise 8 (10 points)

The classes `ODESolver`, `ForwardEuler`, and `Midpoint` from the previous exercise were developed with scalar ordinary differential equations in mind (not several (an array of) functions). However, they may also work for systems of differential equations provided that the initial condition `Y` is an array and that `f(y, t)` returns an array. We can either demand the user to ensure this, or we can modify class `ODESolver` to convert `Y` and `f` to arrays automatically. Following the latter idea, we replace the statement

```
        self.y.append(Y)
```

in the `initcond` method by

```
        self.y.append(numpy.asarray(Y))
```

The `numpy.asarray` function converts its argument to an array if it is not already an array. We also replace the simple assignment

```
        self.f = f
```

in the constructor of class `ODESolver` by

```
        self.user_f = f
        self.f = self.f_array_return
```

where `f_array_return` is a new method in class `ODESolver` that calls the user's `f` function, stored in `self.f`, and then converts the return value to an array:

*(Continued on page 10.)*

```
    def f_array_return(self, y, t):
        rhs = self.user_f(y, t)
        return numpy.asarray(rhs)
```

Assume that the above modifications are made in class `ODESolver`, and hence that `ODESolver`, `ForwardEuler`, and `Midpoint` work for systems of ordinary differential equations even if the user specifies `Y` as a list and lets `f` return a list. (Note that class `ODESolver` from the course software already includes the modifications above, but the modification of `self.f` is done in a more compact way: `self.f = lambda y, t:  numpy.asarray(f(y, t))`.)

The task in this exercise is to apply the `Midpoint` or `ForwardEuler` classes to solve a second-order differential equation,

$$u'' + P(u') + Q(u) = 0, \quad u(0) = a,\ u'(0) = 0,$$

written as a system of two first-order equations

$$\begin{aligned}\frac{d}{dt}y^0 &= y^1,\\ \frac{d}{dt}y^1 &= -P(y^1) - Q(y^0)\end{aligned}$$

The initial conditions are $y^0(0) = a$ and $y^1(0) = 0$. $P$ and $Q$ are functions specified as

$$P(u') = b|u'|u'$$

$$Q(u) = c\sin(u)$$

These choices correspond to an oscillating pendulum driven by gravity and slowed down by air resistance. The symbols $a$, $b$, and $c$ are given positive constants.

Implement a right-hand side function `f(y, t)` for the system of the two first-order equations listed above. You may use a plain Python function for `f` or (better) a class with a `__call__` method and $b$ and $c$ as attributes.

Create a `ForwardEuler` or `Midpoint` instance and use it to solve the second-order differential equation when $a = c = 1$ and $b = 0.2$. Use $\Delta t = \pi/40$ and take $N = 320$ steps. Plot the solution $u(t)$ versus $t$ (recall that the solution returned from `ForwardEuler.integrate` is a two-dimensional array containing both $u(t)$ and $u'(t)$ values).

Solution:

```
class RHS:
    def __init__(self, b, c):
        self.b,self.c = b, c
    def __call__(self, y, t):
        # y is a list of two elements, introduce
        # names according to math here:
        y0, y1 = y
        b, c = self.b, self.c  # short names
```

*(Continued on page 11.)*

```
        P = b*abs(y1)*y1
        Q = c*sin(y0)
        return [y1, -P - Q]

from scitools.std import *
a = c = 1
b = 0.2
dt = pi/40
N = 320
f = RHS(b, c)
method = Midpoint(f, dt)
method.initcod(a)
y, t = method.integrate(N)
u = y[:,0]
plot(t, y)
```

END