UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Examination in:	INF1100 — Introduction to programming with scientific applications
Day of examination:	Monday, December 3, 2007
Examination hours:	09.00 – 12.00.

This examination set consists of 13 pages.

Appendices:	None.
Permitted aids:	1 handwritten A5 sheet, both sides can be used.

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 that in detail.
- Your answer must be written on these exercise sheets, and not on separate sheets. In the exercises where you need to write program code we recommend that you first make some notes on a separate sheet of paper before you write down the final solution on these exercise sheets at the marked location. If you, contrary to expectation, cannot fit your code on these exercise sheets, additional sheets can be used. These must be clearly marked with exercise number and stapled to the exercise sheets.
- The maximum possible score on the exam is 75 points. There are 14 exercises, and the number of point for each exercise is given in the heading.
- For the first five exercises, make sure to limit your answer to a single line.

(Continued on page 2.)

Exercise 1 (3 points)

What will the value of the variable `lnx` be at the end of the following code?

```
x = 1024; lnx = 0
while x > 1:
    x/=2
    lnx += 1
```

Answer: 10

Exercise 2 (3 points)

What will the value of the variable `result` be at the end of the following code?

```
result = 1
for i in range(4):
    result = 2*(i + result)
```

Answer: 38

Exercise 3 (3 points)

What will the value of the variable `result` be at the end of the following code?

```
result = 0
for i in range(5):
    for j in range(i, 5):
        result += 1
```

Answer: 15

(Continued on page 3.)

Exercise 4 (3 points)

What will the value of the variable `result` be at the end of the following code?

```
result = 0
for i in range(1, 4):
    prod = 1
    for j in range(1, i+1):
        prod*=i
    result += prod
```

Answer: 32

Exercise 5 (3 points)

How many times is the text ``‘‘INF1100’’`` printed by the following code?

```
for i in range(10, 100, 10):
    for j in range(i-1, i+3):
        print "INF1100"
```

Answer: 36

(Continued on page 4.)

Exercise 6 (6 points)

Write a function `position(t, v0)` that returns the value of the function

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

and its derivative $y'(t)$. Also write a main program that calls `position` to compute $y(0.1)$ and $y'(0.1)$ for $v_0 = 2$. Write out the two values that were returned from the function. The parameter g is the acceleration of gravity, which can be set equal to 9.81.

```
def position(t, v0):
    g = 9.81
    y1 = v0*t - 0.5*g*t**2
    y2 = v0 - g*t
    return y1, y2

y, y_deriv = position(0.1, 2)
print 'y(t) =', y
print "y'(t) =", y_deriv
```

(Continued on page 5.)

Exercise 7 (6 points)

The trajectory of a ball (neglecting air resistance) is described by the parabola

$$y = f(x) = Y - \frac{1}{2U^2}\frac{gx^2}{\cos^2\alpha} + x\tan\alpha.$$

In this expression, x is a horizontal coordinate, g is the acceleration of gravity, U is the size of the initial velocity that makes an angle α with the x axis, and $(0, Y)$ is the initial position of the ball.

In a program, first read the input data Y, α, and U from the command line. Then let the program compute where the ball hits the ground, i.e., the value x_g for which $f(x_g) = 0$ (the formula and also the code from Exercise 12 can be used here). Plot the trajectory $y = f(x)$ for x in the interval $[0, x_g]$.

```
import sys
from scitools.all import *
Y = eval(sys.argv[1])
alpha = eval(sys.argv[2])
U = eval(sys.argv[3])
g = 9.81

def f(x, y, a, u):
    return y - g*x**2/(2.*u**2*cos(a)**2) + x*tan(a)

a = -g/(2.*U**2*cos(alpha)**2)
b = tan(alpha)
c = Y
x1 = (-b + sqrt(b**2 - 4*a*c))/(2.*a)
x2 = (-b - sqrt(b**2 - 4*a*c))/(2.*a)
xg = x1
if x2 > x1:
    xg = x2
print 'ball hits the ground after', xg, 'meter'
x = linspace(0, xg, 1001)
y = f(x, Y, alpha, U)
plot(x, y)
```

(Continued on page 6.)

Exercise 8 (6 points)

A difference equation for the daily growth of money in a bank reads

$$x_n = x_{n-1} + \frac{p}{100 \cdot 360} x_{n-1},$$

where p is the annual interest rate and n counts days. Write a function `solve(n, p, x0)` for computing x_n after `n` days with annual interest rate `p` and a start capital `x0`. The function should return x_n. Do not store the x_n values in a list; use instead as little storage as possible inside the `solve` function. Write a test program where you compute how much 100 NOK has grown to after 720 days with annual interest rate $p = 4$.

```
def solve(n, p, x0):
    for i in range(n):
        xn = x0 + (p*x0)/(100*360.)
        x0 = xn
    return xn

money = 100
days = 720
percentage = 4
fortune = solve(days, percentage, money)
print 'money has grown to %f after %d days' %(fortune, days)
```

(Continued on page 7.)

Exercise 9 (6 points)

Represent the mathematical function

$$f(t;p,q,r) = \sin(\pi t) + p\sin(q\pi t) + r\sin(4q\pi t) \qquad (1)$$

by a class `SineFunc`. The class must be written such that the following program works when the class is stored in the file `SineFunc.py`:

```
from SineFunc import SineFunc
f = SineFunc(p=0.1, q=4, r=0.05)
for t in [0.2, 0.4, 1, 2]:
    f_value = f(t)
    print 'f(%g)=%g' % (t, f_value)
```

```
class SineFunc:
    from scitools.all import sin, pi
    def __init__(self, p, q, r):
        self.p = p
        self.q = q
        self.r = r
    def __call__(self, t):
        p = self.p
        q = self.q
        r = self.r
        return sin(pi*t) + p*sin(q*pi*t) + r*sin(4*q*pi*t)
```

(Continued on page 8.)

Exercise 10 (6 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. Assume no empty lines in the file.

```
from scitools.all import plot
ifile = open('myfile.dat')
x = []; y = []
for line in ifile:
    numbers = line.split()
    x.append(float(numbers[0]))
    y.append(float(numbers[1]))
plot(x, y)
```

(Continued on page 9.)

Exercise 11 (6 points)

The purpose of this exercise is to write a file like `mydat.txt` in the previous exercise. Given a function $f(x)$, an interval $[a, b]$, and a number of points n, make a function `table(f, a, b, n, filename)` that writes out $x_i = a + ih$ and $y_i = f(x_i)$ to a file `filename` for $i = 0, \ldots, n-1$ and $h = (b-a)/(n-1)$. Write a main program that calls `table` for the function defined in (1) in exercise 9. Choose the interval $[a, b]$ as $[0, \pi]$, set $n = 31$, and let the parameters in f be $p = 0.5$, $q = 3$, and $r = 0.05$.

```
from scitools.all import *
def table(f, a, b, n, filename):
    ofile = open(filename, 'w')
    x = linspace(a, b, n)
    y = f(x)
    for i in range(len(x)):
        ofile.write('%8f \t %8f\n' %(x[i], y[i]))

from SineFunc import SineFunc
f = SineFunc(p=0.5, q=3, r = 0.05)
a = 0
b = pi
n = 31
filename = 'mydat.txt'
table(f, a, b, n, filename)
```

(Continued on page 10.)

Exercise 12 (6 points)

In this exercise we shall write and test a class for representing a parabola $f(x; a, b, c) = ax^2 + bx + c$. Make a class `Parabola` which stores a, b, and c as attributes and which has three methods:

- `__init__` for storing the coefficients in the parabola: a, b, and c,
- `__call__` for computing a value of f at a point x,
- `roots` for computing the two roots of the parabola:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

You can assume that only real (not complex) roots are of interest. Write a short test program for evaluating the parabola $-1 + x^2$ and computing its roots.

```
from math import sqrt
class Parabola:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
    def __call__(self, x):
        a = self.a
        b = self.b
        c = self.c
        return a*x**2 + b*x + c
    def roots(self):
        a = self.a
        b = self.b
        c = self.c
        x1 = (-b + sqrt(b**2 - 4*a*c))/(2.*a)
        x2 = (-b - sqrt(b**2 - 4*a*c))/(2.*a)
        return x1, x2

if __name__ == '__main__':
    p = Parabola(1, 0, -1)
    print p(2)
    print p.roots()
```

(Continued on page 11.)

Exercise 13 (6 points)

Somebody suggests the following game: You pay 1 NOK of money and are allowed to throw three dice. If you find (at least) one die with 2 eyes and (at least) one die with 6 eyes among the three dice, you win 7 NOK, otherwise you lose your 1 NOK investment. Make a program for deciding whether you should play this game or not.

```
from numpy import random
n = 10000
money = 0
for i in range(n):
    r = random.random_integers(1, 6, size=3)
    if 2 in r and 6 in r:
        money += 7
    money -= 1
print 'money left:', money
# When running this program, we end up with about
# -250 NOK in average, so we should not play this
# game, unless we want to go bankrupt!
```

(Continued on page 12.)

Exercise 14 (12 points)

A class hierarchy `Integrator` implements different methods for numerical integration of $\int_a^b f(x)dx$. All methods can be written as a sum

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i). \tag{2}$$

Different integration methods differ in the way the weights w_i and the evaluation points x_i are defined. For example, the Trapezoidal rule has $x_i = a + ih$ for $h = (b-a)/(n-1)$ for $i = 0, \ldots, n-1$, and $w_0 = w_{n-1} = h/2$ while $w_1 = w_2 = \cdots = w_{n-2} = h$.

The superclass `Integrator` stores a, b, and the number of function evaluation points n in its constructor, while the method `__call__` computes the numerical approximation to the integral:

```
class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.set_up_points_and_weights()

    def set_up_points_and_weights(self):
        raise NotImplementedError, 'no rule in class %s' % \
              self.__class__.__name__

    def __call__(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

Subclasses implement different integration rules in the Python method `set_up_points_and_weights`. For example, we can implement the Trapezoidal rule as

```
class Trapezoidal(Integrator):
    def set_up_points_and_weights(self):
        h = (self.b - self.a)/float(self.n - 1)
        x = numpy.zeros(self.n)
        w = numpy.zeros(self.n)
        for i in range(self.n):
            x[i] = a + i*h
        w[0] = h/2
        for i in range(1, self.n-1):
            w[i] = h
        w[-1] = h/2
        return x, w
```

(Continued on page 13.)

The task of this exercise is to implement Monte Carlo integration as a subclass in the `Integrator` hierarchy. The Monte Carlo rule is defined as (2) with $w_i = \frac{b-a}{n}$ (for all i) and x_i as uniformly distributed random points in the interval $[a, b]$ (use the `uniform(a,b)` function in one of the `random` modules to draw the points). Write the complete code for a subclass implementing Monte Carlo integration. Also write a short main program for computing $\int_1^{10} 2\sin^{10}(3x)dx$ with 10000 evaluation points.

```
import numpy
from Integrator import *

class MonteCarlo(Integrator):
    def set_up_points_and_weights(self):
        w = numpy.ones(self.n)*(self.b - self.a)/float(self.n)
        x = numpy.random.uniform(self.a, self.b, size=self.n)
        return x, w

def f(x):
    return 2*numpy.sin(3*x)**10
a = 1
b = 10
n = 10000
m = MonteCarlo(a, b, n)
print m(f)
```

END