HOMEWORK 6

Dingcheng Yue

Those are the general import I use for ipython notebook

```
import numpy as np
import scipy as sp
import numpy.linalg as la
import matplotlib.pyplot as plt
import seaborn
import scipy.optimize as op

%matplotlib inline
```

Problem 1 (with computer): (25 points)

Using

- Romberg Integration
- Gauss Quadrature

evaluate the double integral

$$\int\int e^{-xy} dxdy$$

over each of the following regions:

(a) The unit square, i.e. $0 \leq x \leq 1, 0 \leq y \leq 1$
(b) The quarter of the unit disk lying in the first quadrant, i.e., $x^2 + y^2 \leq 1, x \geq 0, y \geq 0$

For (a) the integration is

$$\int_0^1 \int_0^1 e^{-xy} dxdy = \int_0^1 e^{-x} dx \int_0^1 e^{y} dy$$

For (b), the integration is not in a rectangle, we then could change the coordinates to polar form (r, φ)

$$\int_0^1 \int_0^{\sqrt{1-y^2}} e^{-xy} dxdy$$
$$= \int_0^{\pi/2} \int_0^1 re^{-sin(\varphi)rcos(\varphi)r} drd\varphi$$

The code is following

```
def memorise(f):
    """ Memoization decorator for functions taking one or more
arguments. """
    class memodict(dict):
        def __init__(self, f):
            self.f = f
        def __call__(self, *args):
            return self[args]
        def __missing__(self, key):
            ret = self[key] = self.f(*key)
            return ret
    return memodict(f)

def trapozoid(f, a1, b1, a2, b2, n):
    steps = (1<<n)+1
    h1 = (b1-a1)/(steps-1)
    h2 = (b2-a2)/(steps-1)
    w = np.ones(steps)
    w[0] = w[-1] = 0.5
    w2 = np.outer(w, w)
    p1 = np.outer(np.linspace(a1, b1, steps), np.ones(steps))
    p2 = np.outer(np.ones(steps), np.linspace(a2, b2, steps))
    return np.sum(np.sum(f(p1, p2)*w2))*h1*h2

def romberg2(f, a1, b1, a2, b2, n=4):
    @memorise
    def T(i, k):
        if i==0:
```

```
                return trapozoid(f, a1, b1, a2, b2, k)
            else:
                return (4**i*T(i-1, k)-T(i-1, k-1))/(4**i-1)
        return T(n, n)

    from numpy.polynomial.legendre import leggauss

    def GuasssianQ2d(f, a1, b1, a2, b2, n=4):
        p,w = leggauss(n)
        w2 = np.outer(w,w)
        p1 = np.outer((p+1)*(b1-a1)*0.5+a1, np.ones(n))
        p2 = np.outer(np.ones(n), (p+1)*(b2-a2)*0.5+a2)
        return np.sum(np.sum(f(p1, p2)*w2))*(b1-a1)*(b2-a2)/4

    f1 = lambda x,y: np.exp(-x*y)
    f2 = lambda r,p: r*np.exp(-r**2*np.sin(p)*np.cos(p))

    print("Guassian Quadrature Rules")
    print(GuasssianQ2d(f1, 0, 1, 0, 1))
    print(GuasssianQ2d(f2, 0, 1, 0, np.pi/2))

    print("Romberg + composite trapozoid")
    print(romberg2(f1, 0, 1, 0, 1, n=2))
    print(romberg2(f2, 0, 1, 0, np.pi/2))
```

result:

> Guassian Quadrature Rules
> 0.796599599217
> 0.675165202482
> Romberg + composite trapozoid
> 0.796592085491
> 0.675167079729

If I plug equations to wolfram alpha, I get the following result
a: 0.7966
b: 0.6752

Problem 2 (by hand): (20 points)

Classify each of the following ODEs as having unstable, stable, or asymptotically stable solutions. Briefly explain your answer for each.

(a) $y' = y + t$

$$\frac{\partial y'}{\partial t} = 1 > 0$$

It is unstable.

(b) $y' = y - t$

$$\frac{\partial y'}{\partial t} = 1 > 0$$

It is unstable.

(c) $y' = t - y$

$$\frac{\partial y'}{\partial t} = -1 < 0$$

It is stable.

(d) $y' = 1$

$$\frac{\partial y'}{\partial t} = 0$$

It is partially stable.

Problem 3 (by hand): (5 points)

What does it mean for the accuracy of a numerical method for solving ODEs to be of order of p?

Since we are using iteration method for solving equations, we define each iteration step h. Then the order of a numerical method is the number of factors of h in the global truncation

error estimate for the method.

For example, Euler's method is therefore a first-order method. $p = 1$

Problem 4 (by hand): (20 points)

Write each of the following ODEs as an equivalent first-order system of ODEs:

(a) Van der Pol equation:

$$y'' = y'(1 - y^2) - y$$

Solution:
y_i is the ith derivative of y

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_1(1 - y_0^2) - y_0 \end{aligned}$$

(b) Blasius equation:

$$y''' = -yy''$$

Solution:
y_i is the ith derivative of y

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= -y_0 y_2 \end{aligned}$$

(c) Newton's Second Law of Motion for two-body problem:

$$\begin{aligned} y_1'' &= -GMy_1/(y_1^2 + y_2^2)^{3/2} \\ y_2'' &= -GMy_2/(y_1^2 + y_2^2)^{3/2} \end{aligned}$$

Solution:
$s \leftarrow y_1$ and $t \leftarrow y_2$

$$
\begin{aligned}
s_0' &= s_1 \\
t_0' &= t_1 \\
s_1' &= -GMs_0/(s_0^2 + t_0^2)^{3/2} \\
t_1' &= -GMt_0/(s_0^2 + t_0^2)^{3/2}
\end{aligned}
$$

Problem 5: (30 points)

Consider the ODE $y' = -5y$ with initial condition $y(0) = 1$. We will solve this ODE numerically using a step size of $h = 0.5$. Parts (a) - (e) should be completed entirely by hand. Parts (f) and (g) should be done using a computer.

(a) Are the solutions to this ODE stable?

Yes, since $\frac{\partial y'}{\partial t} = -5 < 0$

(b) Is Euler's method stable for this ODE using this step size?

$\lambda = -5, h = 0.5, |1 + h\lambda| = 1.5 > 1$
No, it is not stable.

(c) Compute the numerical value for the approximate solution at $t = 0.5$ given by Euler's method.

$y_1 = y_0 + h\lambda y_0 = -1.5$

(d) Is the backward Euler method stable for this ODE using this step size?

Yes, since the backward Euler method is unconditionally stable.

(e) Compute the numerical value for the approximate solution at $t = 0.5$ given by the backward Euler method.

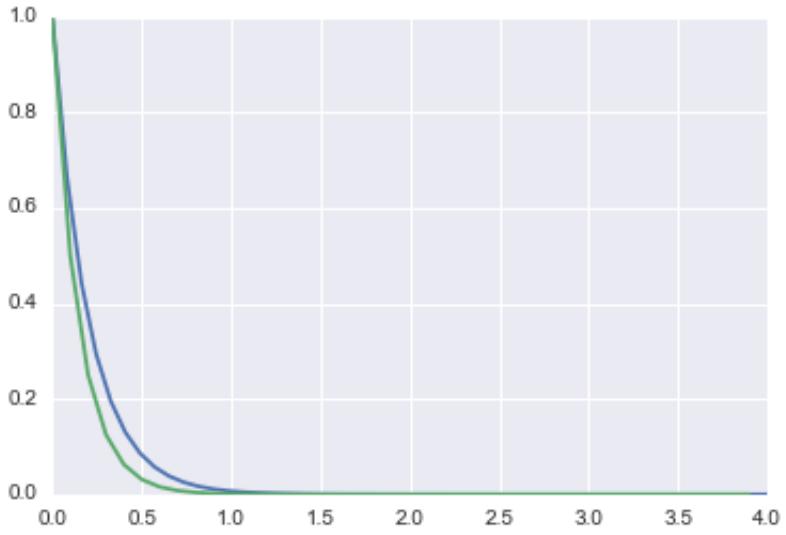
$$
\begin{aligned}
y_1 &= y_0 + hf(t_1, y_1) \\
y_1 &= 1 + 0.5 \times (-5y_1) \\
y_1 &= \frac{2}{7}
\end{aligned}
$$

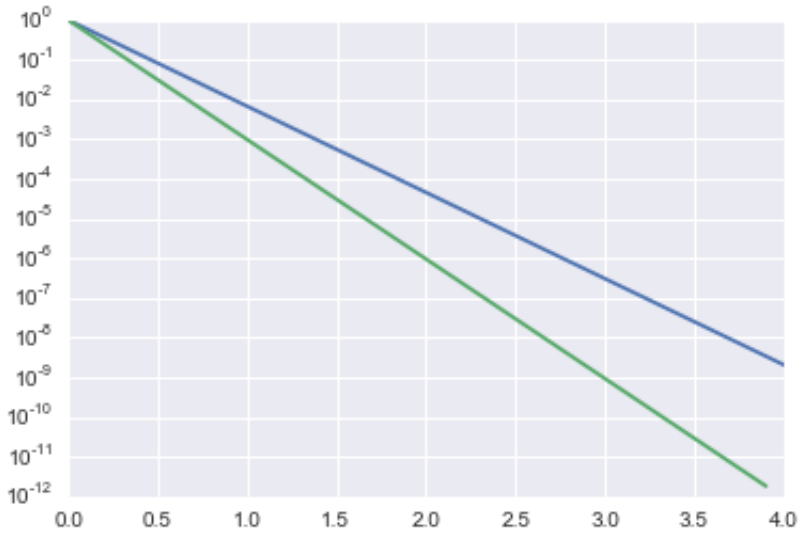
(f) Write a program that implements forward Euler. Use you implementation to compute the numerical value for the approximate solution at $t = 4.0$ using a step size of $t = 0.1$. What is the value obtained? Plot the sequence of solutions (i.e. plot y_k vs t_k.)

The code is as follow

```
def forwardEurler(f, y0, h, t0):
    yield(t0, y0)
    while True:
        y0 = y0+h*f(t0,y0)
        t0 += h
        yield t0,y0

from itertools import islice
i = 0;
t, y = zip(*list(islice(forwardEurler(lambda t,y: -5*y, 1, 0.1,
0), 40)))

x = np.linspace(0.0,4.0)
plt.plot(x, np.exp(-5*x))
plt.plot(t,y)
plt.show()

plt.semilogy(x, np.exp(-5*x))
plt.semilogy(t,y)
plt.show()
print(y[-1])
```

And the result as follow





> 9.094947017729282e-13

(g) Write a program that implements backward Euler. Use you implementation to compute the numerical value for the approximate solution at $t = 4.0$ using a step size of $t = 0.1$. What is the value obtained? Plot the sequence of solutions (i.e. plot y_k vs t_k.) How do your results compare with your results from part (f) for this problem?

The exact solution should be

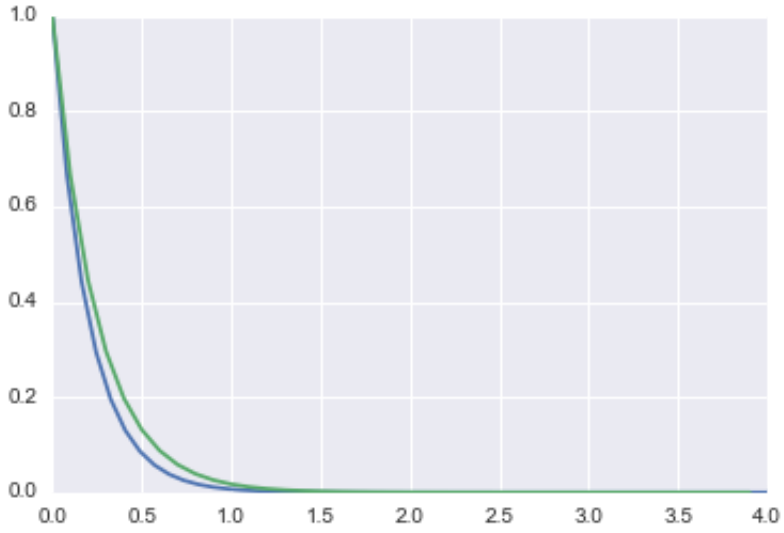
$$y = e^{-5t}$$

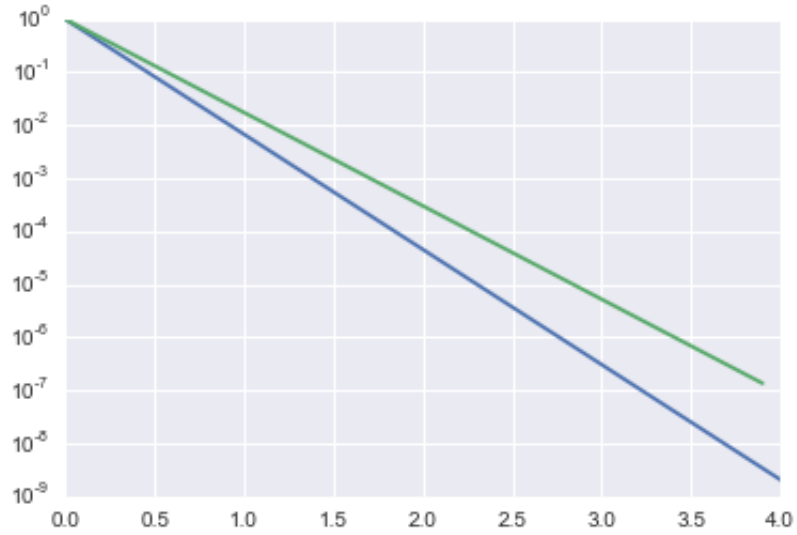
So, I did a plot and also a semilogy for comparison

```
def backwardEurler(f, y0, h, t0):
    yield(t0, y0)
    while True:
        t0 += h
        eq = lambda y1: y0+h*f(t0,y1)-y1
        y0 = op.fsolve(eq, y0)[0]
        yield t0,y0

from itertools import islice
i = 0;
t, y = zip(*list(islice(backwardEurler(lambda t,y: -5*y, 1, 0.1,
0), 40)))
x = np.linspace(0.0,4.0)
plt.plot(x, np.exp(-5*x))
plt.plot(t,y)
```

```
    plt.show()

    plt.semilogy(x, np.exp(-5*x))
    plt.semilogy(t,y)
    plt.show()
    print(y[-1])
```

And the result as follow






```
9.04377268382e-08
```

As we can see, both methods are stable but forward one has smaller value than actual and the backward one has larger one than actual.