Report Lab 2

Nguyen Tien Duc - ITITIU18029

December 20, 2019

Part I

Bisection vs False-Position

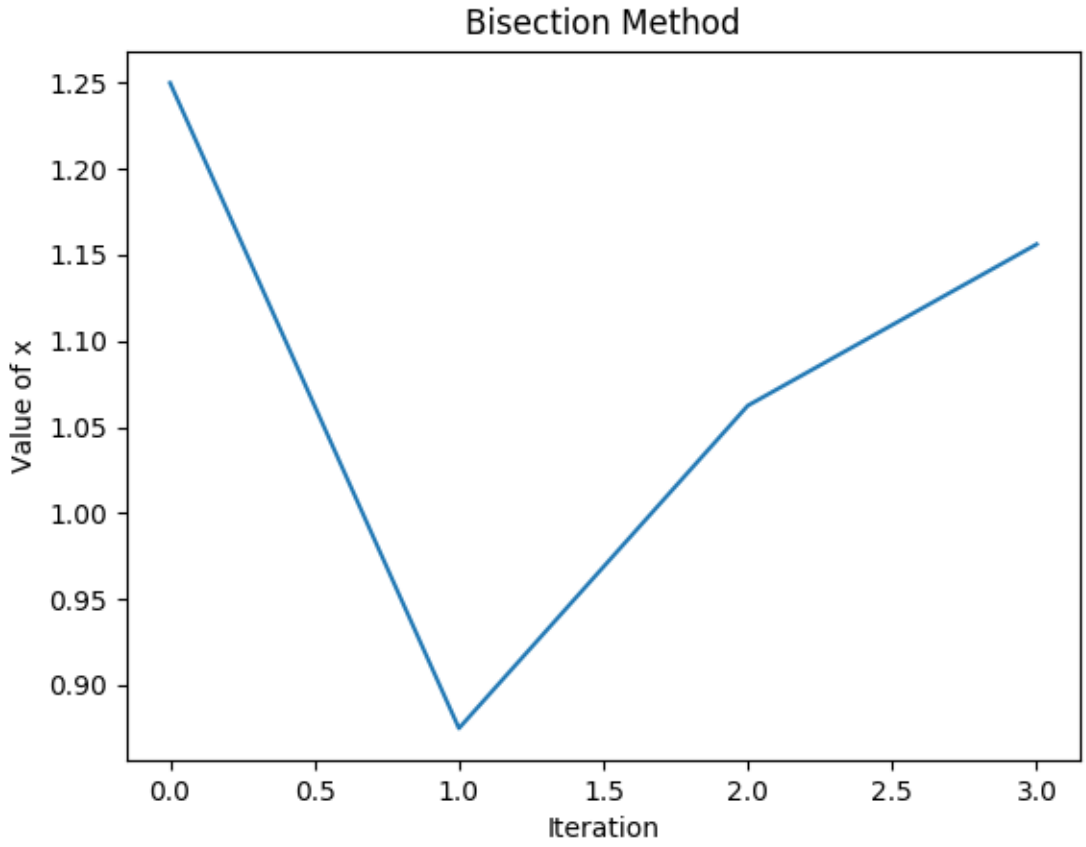
Bisection

Code

```
import numpy as np
import matplotlib.pyplot as plt

l_x = []
l_iter = []


def bisection(f,a,b,N):
    print("Iteration%20s%20s%20s%20s" % ("xM", "xI" , "xU", "f(xI)*f(xM)", "f(xU)
                                                    *f(xM)"))

    if f(a)*f(b) >= 0:
        print("Bisection method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(0,N+1):
        l_iter.append(n)
        m_n = (a_n + b_n)/2
        l_x.append(m_n)
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Bisection method fails.")
            return None
```

```
        print("%9d%20f%20f%20f%20f%20f" % (n, m_n, a, b, f(a_n)*f(m_n), f(b_n)*f(m_n)
                                                                    ))
    return m_n


f = lambda x: np.log(x**4) - 0.7
result = bisection(f, 0.5, 2, 3)
print("The result is %f" % result)


plt.title("Bisection Method")
plt.xlabel("Iteration")
plt.ylabel("Value of x")
plt.plot(l_iter, l_x)
plt.savefig("BisectionGraph.png")
plt.show()
```

Running



Iteration	xM	xI	xU	f(xI)*f(xM)	f(xU)*f(xM)
0	1.250000	0.500000	2.000000	-0.668731	0.037085
1	0.875000	0.500000	2.000000	1.523066	-0.237661
2	1.062500	0.500000	2.000000	0.209308	-0.088103
3	1.156250	0.500000	2.000000	0.014226	-0.022969

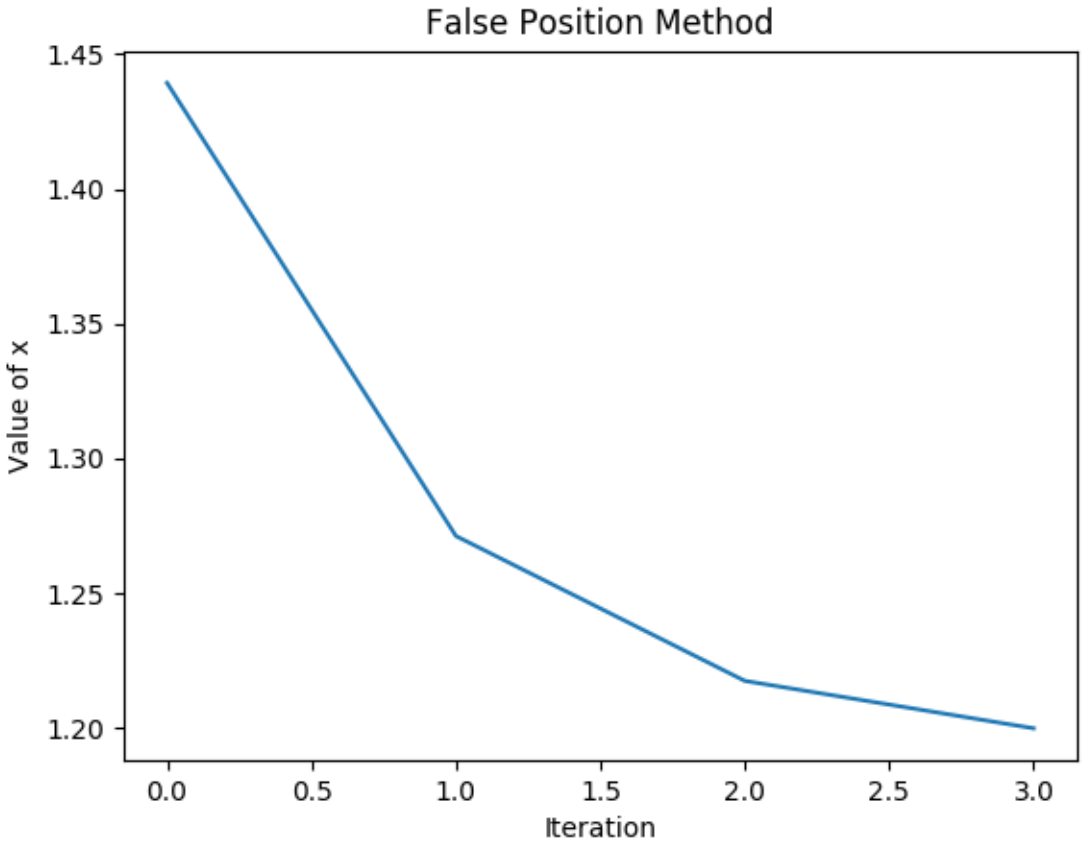
The result is 1.156250

False-Position

Code

```
import numpy as np
import matplotlib.pyplot as plt

list_xH = []
list_iter = []


def regula_fasi(f, xI, xU, iter):
    if f(xI)*f(xU) >= 0:
        print("Wrong initial guesses")
        return -1

    print("Iteration%10s%10s%10s" % ("xH", "xI" , "xU"))

    for i in range(iter):
        # Find the touch point
        xH = (xI*f(xU)-xU*f(xI))/(f(xU)-f(xI))

        list_xH.append(xH)
        list_iter.append(i)

        if f(xH) == 0:
            break
        elif f(xH)*f(xI) < 0:
            xU = xH
        else:
            xI = xH
        print("%9d%10f%10f%10f" % (i, xH, xI, xU))
    return xH


f = lambda x: np.log(x**4) - 0.7
result = regula_fasi(f, 0.5, 2, 4)
print("The positive real root is %f" % result)


plt.title("False Position Method")
plt.xlabel("Iteration")
plt.ylabel("Value of x")
plt.plot(list_iter, list_xH)
plt.savefig("RegulaFalsiGraph.png")
plt.show()
```

Running



Iteration	xH	xI	xU
0	1.439354	0.500000	1.439354
1	1.271272	0.500000	1.271272
2	1.217534	0.500000	1.217534
3	1.199936	0.500000	1.199936

The positive real root is 1.199936

Comment

This exercise showed right away that false-position method is better than bisection method.

- The first thing to notice is that the true value is 1.1912, which is much closer to the result of the false-position method after 3 iterations. On the other hand, the same number of iterations of bisection method only return a result that is accurate to the first decimal place.
- Secondly, the algorithm for false-position method is shorter, simpler, requires less computations and comparisons as showed.
- False Position likes to stick to one end for many iterations.
- This implementation is just a raw one, there are even faster ways to implement false-position method.
- Though false-position method is usually faster than bisection method. With certain complex functions, it can be still as slow as bisection method.

In conclusion, both are very old methods to find roots.
However, due to the advantages of false-position method, some versions of it is still in use.
Nevertheless, in reality, most people will switch to Newton's method or even Secant Method as those methods can usually do a better job than both false-position and bisection methods.

Part II
Fixed-Point Iteration vs Newton-Raphson

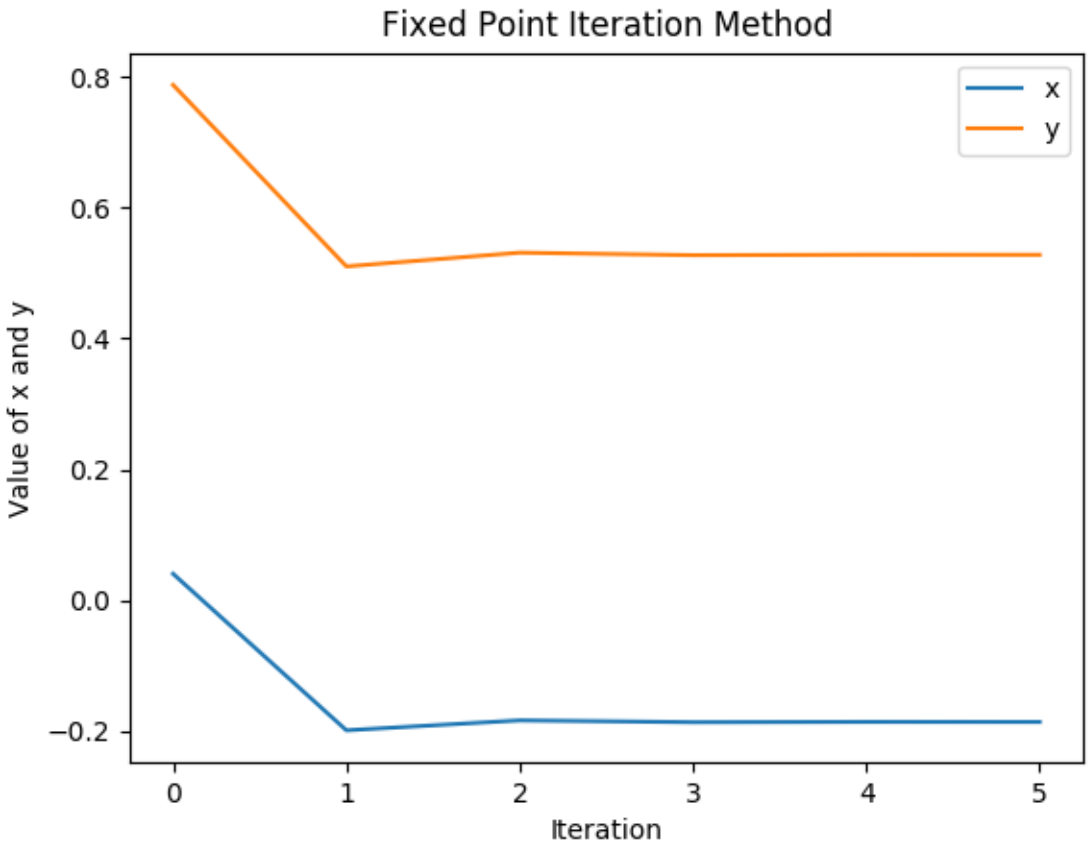
Fixed-Point Iteration

Code

```
from timeit import timeit as running_time
from matplotlib import pyplot as plt


def y_next(x, y):
  return -x ** 2 + x + 0.75


def x_next(x, y):
  return (x ** 2 - y) / (5 * y)


def err(value_new, value_old):
  return abs((value_new - value_old) / value_new) * 100


x_list = []
y_list = []


def fixed_point_iteration(x_i, y_i, e_s):
  x, y = x_i, y_i
  x_list.clear()
  y_list.clear()
  while err(x_next(x, y), x) > e_s or err(y_next(x, y), y) > e_s:
    x = x_next(x, y)
    y = y_next(x, y)
    x_list.append(x)
    y_list.append(y)
  return x, y


def result():
  return fixed_point_iteration(x_i=1.2, y_i=1.2, e_s=0.01)


time_elapsed = running_time(result, number=10)
print("Running time is:", end=" ")
print(time_elapsed)
print("Result using Fixed-Point Iteration Method:")
print(result())
```

```
plt.title("Fixed Point Iteration Method")
plt.xlabel("Iteration")
plt.ylabel("Value of x and y")
plt.plot(x_list, label='x')
plt.plot(y_list, label='y')
plt.legend()
plt.savefig("FixedPointIteration.png")
plt.show()
```

Running




```
Running time is 0.001283780999983719
Result using Fixed-Point Iteration's Method:
(-0.18680449435919252, 0.5282995865280139)
```

Newton-Raphson

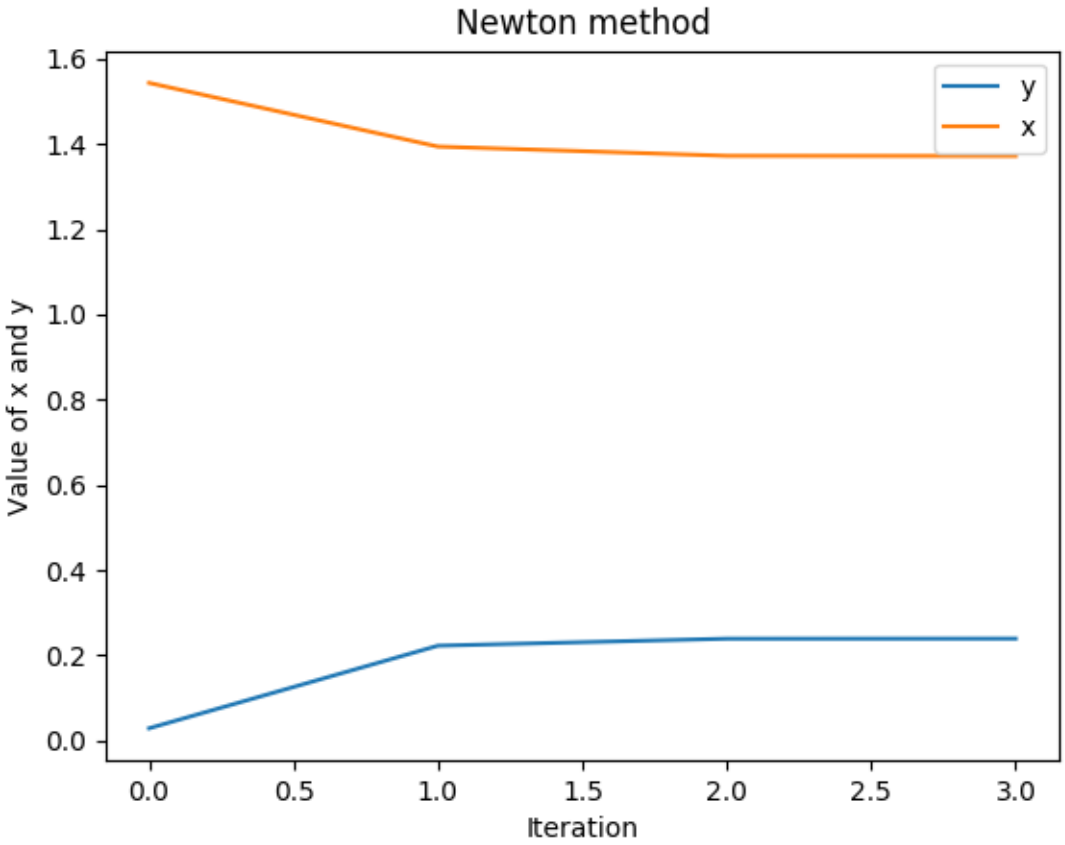
Code

```
from timeit import timeit as running_time
from matplotlib import pyplot as plt
import sympy as sp
```

```
X, Y = sp.symbols('X, Y', real=True)
U = -X ** 2 + X + 0.75 - Y
V = X ** 2 - 5 * X * Y - Y


def u(x, y):
  return U.subs([(X, x), (Y, y)])


def v(x, y):
  return V.subs([(X, x), (Y, y)])


def dudx(x, y):
  return sp.diff(U, X).subs([(X, x), (Y, y)])


def dudy(x, y):
  return sp.diff(U, Y).subs([(X, x), (Y, y)])


def dvdx(x, y):
  return sp.diff(V, X).subs([(X, x), (Y, y)])


def dvdy(x, y):
  return sp.diff(V, Y).subs([(X, x), (Y, y)])


def determinant(x, y):
  return dudx(x, y) * dvdy(x, y) - dudy(x, y) * dvdx(x, y)


def x_next(x, y):
  numerator = u(x, y) * dvdy(x, y) - v(x, y) * dudy(x, y)
  return x - numerator / determinant(x, y)


def y_next(x, y):
  numerator = v(x, y) * dudx(x, y) - u(x, y) * dvdx(x, y)
  return y - numerator / determinant(x, y)


def err(value_new, value_old):
  return abs((value_new - value_old) / value_new) * 100


x_list = []
y_list = []


def newton(x_i, y_i, e_s):
  x_list.clear()
  y_list.clear()
  x, y = x_i, y_i
  while err(x_next(x, y), x) > e_s or err(y_next(x, y), y) > e_s:
    x_tmp = x_next(x, y)
    y_tmp = y_next(x, y)
```

```
    x, y = x_tmp, y_tmp
    x_list.append(x)
    y_list.append(y)
  return x, y


def result():
  return newton(x_i=1.2, y_i=1.2, e_s=0.01)


time_elapsed = running_time(result, number=10)
print("Running time is:", end=" ")
print(time_elapsed)
print("Result using Newton's Method:")
print(result())


plt.title("Newton method")
plt.xlabel("Iteration")
plt.ylabel("Value of x and y")
plt.plot(y_list, label='y')
plt.plot(x_list, label='x')
plt.legend()
plt.savefig("NewtonRaphson.png")
plt.show()
```

Running




```
Running time is: 0.3388859950000551
Result using Newton's Method:
(1.37206552043655, 0.239501879437314)
```

Comment

As the running time showed, Newton-Raphson method for solving system of non-linear equations is much slower compared to fixed-point iteration. However, the main advantage is the ability to solve bigger, more complex systems which is hard to show due to the simplicity of the system of equations given in the lab session.

In conclusion, each method has their own advantages, thus they should be chosen wisely based on each particular function.