Becs-114.1100 Computational Science – exercise round 3

Kunal Ghosh, 546247

October 6, 2015

1 Solution to Question 3 (a)

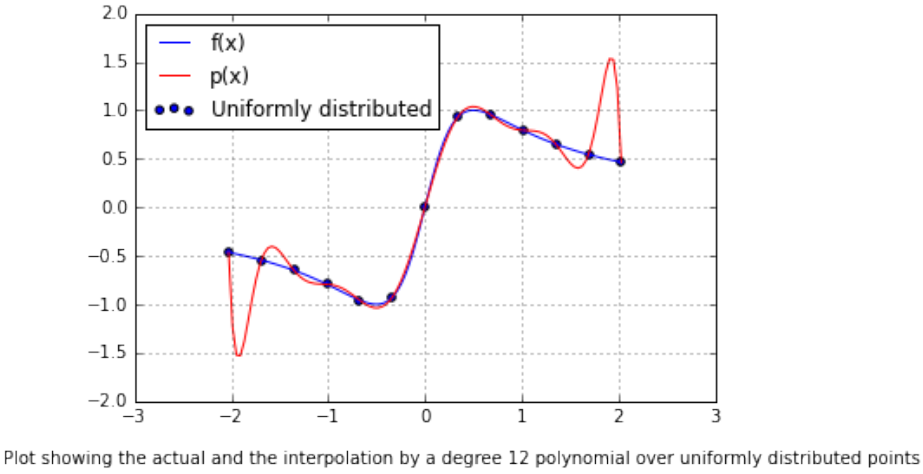

Figure 1: Plot showing the actual and the interpolation by a degree 12 polynomial over uniformly distributed points

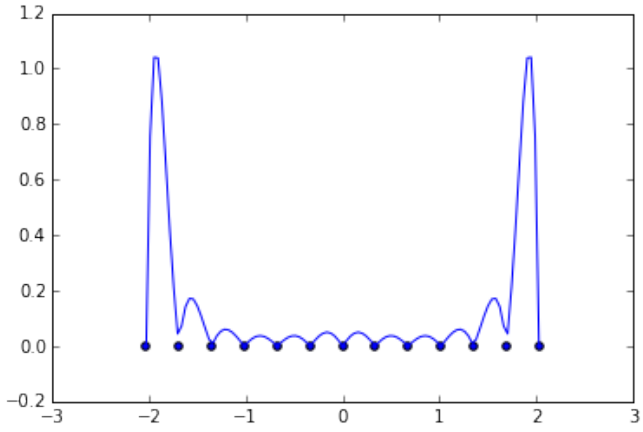

Figure 2: The absolute error |f(x) - p(x)| in polynomial interpolation of $x/(0.25 + x^2) using 13 equally spaced points.$

It can be observed from the table of x_n and $f(x_n)$ that initially the root found by newton's method goes to really large values very far away from the actual root. This is because we hit the "Flat Spot" at that point where the slope $f'(x)$ is almost zero which means $\frac{1}{f'(x)}$ becomes really large and when

we subtract that from x_n we get really large approximation which might throw us far away from the actual root.

The corresponding python code can be found at 4

2 Solution to Question 2 (b)

In the solution to this problem I arbitrarily chose bounds of -5 and +5 for the bisection method. Then everytime, the x_n from newton method went out of bounds, I used bisection method and updated the bounds of the bisection method, as can be seen in the table below. The number of steps taken to converge to the root reduced drastically !

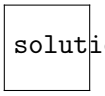

Figure 3: Plot of the function $x^3 - x - 5 = 0$ and our approximated root x_n plotted by the point.

The corresponding python code can be found at 5

3 Solution to Question 3

This problem is essentially the same as the second question. The only difference is that the new functions must handle compelx numbers instead of real values.

Note : Early stopping of iteration is really important and I noticed it in this problem since my first draft of the code did not have the stopping criterion and the plot for 1000 points took a really really long time (maybe around 7 hours, I ran it overnight).

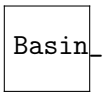

Figure 4: Basin of attraction with Real values on X axis and Imaginary values on the Y scale for $z^3 - 1 = 0$ Green is $(-1, 0j)$, Red is $(-5, -0.866j)$ and Blue is $(-5, 0.866j)$

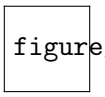

Figure 5: Basin of attraction plotted for 100 values each, on X and Y axes. Real values on X axis and Imaginary values on the Y scale for $z^3 - 1 = 0$ Green is $(-1, 0j)$, Red is $(-5, -0.866j)$ and Blue is $(-5, 0.866j)$

The corresponding python code can be found at 6

4 Appendix A

Python source code for **??**.

```
from __future__ import division
import doctest
import pylab as pl
import numpy as np
# use newton's method to calculate roots
# equation = x^3 - x - 5
# x_initial = 0.57735
# max_iter = 50
# print the results and explain them

## TODO : Write tests
# x^3 - x - 5 , re-written to get rid of powers
def f(x):
    '''
    >>> f(1)
    -5
    >>> f(2)
    1
    >>> f(3)
    19
    '''
    return (x * (x+1) * (x-1)) - 5

# 3 x^2 - 1 , re-written to get rid of powers
def f_dash(x):
    '''
    >>> f_dash(1)
    2
    >>> f_dash(2)
    11
    '''
    return (3 * (x + 1) * (x - 1)) + 2


def get_next_appox(x_n_minus_1):
    x = x_n_minus_1
    return x - (f(x) / f_dash(x))

if __name__ == '__main__':
    doctest.testmod()
    x_n_minus_1 = 0.57735 # x_initial
    # iterate for a maximum of 50 iterations
    max_iter = 50
    output = []
    for _ in xrange(max_iter):
        x_n = get_next_appox(x_n_minus_1)
        output.append((x_n_minus_1, f(x_n_minus_1)))
        x_n_minus_1 = x_n

    for val in output:
        print val

    # plot the curve
    pl.figure(1)
    x = np.linspace(-5,5,1000)
    y = [f(_) for _ in x]
```

```
pl.grid(True)
pl.plot(x,y)
pl.scatter(output[-1][0],0)

pl.savefig("solution2a_fig.pdf")
pl.show()
```

5 Appendix B

Python source code for 2.

```
from __future__ import division
import doctest
import pylab as pl
import numpy as np
# a)   use newton's method to calculate roots
#      equation = x^3 - x - 5
#      x_initial = 0.57735
#      max_iter = 50
#      print the results and explain them


# x^3 - x - 5 , re-written to get rid of powers
def f(x):
    '''
    >>> f(1)
    -5
    >>> f(2)
    1
    >>> f(3)
    19
    '''
    return (x * (x+1) * (x-1)) - 5

# 3 x^2 - 1 , re-written to get rid of powers
def f_dash(x):
    '''
    >>> f_dash(1)
    2
    >>> f_dash(2)
    11
    '''
    return (3 * (x + 1) * (x - 1)) + 2

def is_within_bound(val, lower, upper):
    returnVal = False
    # Checks for strict bounds
    if val >= lower and val <= upper:
        returnVal = True
    return returnVal

def get_next_approx_newton(x_n_minus_1):
    x = x_n_minus_1
    return x - (f(x) / f_dash(x))

def get_next_approx_bisection(func, b_min, b_max):
    # returns the new approx root of func
    # via bisection method, new lower and upper bounds
    mid = 0.5 * (b_min + b_max)
    new_b_min = b_min
    new_b_max = b_max
    if (func(new_b_min) * func(mid) < 0):
        new_b_max = mid
    elif (func(mid) * func(new_b_max) < 0):
        new_b_min = mid
    return mid, new_b_min, new_b_max
```

```
if __name__ == '__main__':
    doctest.testmod()
    x_n_minus_1 = 0.57735 # x_initial
    # iterate for a maximum of 50 iterations
    max_iter = 50
    # Bounds for bisection method.
    # Assuming [-5, 5] arbitrarily and it also satisfies
    # f(-5) * f(5) < 0
    bisection_min, bisection_max = -5, 5

    # run till new root approximation and old root approximation
    # are not same upto the given precision
    tolerance = 0.0000000001

    # first approximation from newton's method to setup the while loop
    x_n = get_next_approx_newton(x_n_minus_1)
    print ["method", "New x", "f(x)"]
    print ["newton", x_n, f(x_n)]

    while abs(x_n - x_n_minus_1) > tolerance:
        x_n_minus_1 = x_n
        if is_within_bound(x_n, bisection_min, bisection_max):
            # if the root is within bounds then use newton's method
            x_n = get_next_approx_newton(x_n_minus_1)
            print ["newton", x_n, f(x_n)]
        else:
            # else use bisection method
            x_n, bisection_min, bisection_max = get_next_approx_bisection(f, bisection_min,
                bisection_max)
            print ["bisection", x_n, f(x_n), "new bisec min = ", bisection_min, "new bisec
                max = ", bisection_max]

    pl.figure(1)
    # plot the curve
    x = np.linspace(-5,5,1000)
    y = [f(_) for _ in x]
    pl.grid(True)
    pl.plot(x,y)
    pl.scatter(output[-1][0],0)
    pl.savefig("solution2b_fig.pdf")
    pl.show()
```

6 Appendix C

Python source code for 3.

```
from __future__ import division
import doctest
import pylab as pl
import numpy as np
# a)   use newton's method to calculate roots
#      equation = x^3 - x - 5
#      x_initial = 0.57735
#      max_iter = 50
#      print the results and explain them

## TODO : Write tests
# x^3 - 1 , re-written to get rid of powers
def f(x_r,x_i):
    '''
    >>> f(1,1)
    (-3.0, 2.0)
    >>> f(2,1)
    (1.0, 11.0)
    >>> f(1,0)
    (0.0, 0.0)
    '''
    z = complex(x_r, x_i)
    retVal = (z * z * z) - 1
    return retVal.real, retVal.imag

# 3 x^2 , re-written to get rid of powers
def f_dash(x_r, x_i):
    '''
    >>> f_dash(1,1)
    (0.0, 6.0)
    >>> f_dash(2,1)
    (9.0, 12.0)
    '''
    z = complex(x_r, x_i)
    retVal = 3 * z * z
    return retVal.real, retVal.imag


def get_next_appox(x_r, x_i):
    '''
    x_r : previous real value of x
    x_i : previous imag value of x
    '''
    xr = x_r
    xi = x_i
    x = complex(xr, xi)

    fx_tuple = f(xr, xi)
    fx = complex(fx_tuple[0], fx_tuple[1])

    fdashx_tuple = f_dash(xr, xi)
    fdashx = complex(fdashx_tuple[0], fdashx_tuple[1])

    retVal = x - (fx / fdashx)
    return retVal.real, retVal.imag
```

```
def get_magnitude(z):
    xr, xi = z
    return xr * xr + xi * xi

def run_newton_method(xr_init, xi_init, epsilon=10 ** (-12), max_iter=100):
    '''
    xr_init = initial real value of x
    xi_init = initial imag value of x
    '''
    xr_old = xr_init
    xi_old = xi_init
    xr_new, xi_new = get_next_appox(xr_init, xi_init)

    while(max_iter != 0):
        max_iter -= 1
        xr_old, xi_old = xr_new, xi_new
        xr_new, xi_new = get_next_appox(xr_old, xi_old)
        if abs(xr_old - xr_new) < epsilon and abs(xi_old - xi_new) < epsilon:
            break
        if f(xr_new, xi_new) < epsilon:
            break
    return xr_new, xi_new

def whichRoot(xr, xi):
    retVal = -1
    if xr < -0.4 and xr > -0.6 and xi > -0.9 and xi < -0.7:
        retVal = "r."
    elif (xr + -1 < .01 and xi < .01):
        retVal = "g."
    elif (xr > -0.6 and xr < -0.4) and (xi > 0.8 and xi < 0.9):
        retVal = "b."
    return retVal

if __name__ == '__main__':
    doctest.testmod()

    x_real_nums = np.linspace(-1, 1, 1000)
    x_imag_nums = np.linspace(-1, 1, 1000)

    outputs = []

    for xr in x_real_nums:
        outputs.append([])
        for xi in x_imag_nums:
            fx_r, fx_i = run_newton_method(xr, xi)
            outputs[-1].append(whichRoot(fx_r,fx_i))
    pl.figure(1)
    pl.ion()
    pl.show()
    pl.pcolor(np.array(outputs, np.float))
    pl.draw()
    pl.savefig("figure_q3_test.pdf")
```