Math 4610 – Top 5 Software Manual Pages

Jake Daniels A02307117

December 9, 2022

Module Name: fnIter.py
Author: Jake Daniels
Language: Python
Description/Purpose: This function will compute an approximation of a root for any given function using functional iteration. The main idea is that we will take a function f and create a new function g(x)=x-f(x) where we will look where g(x0)=x0 which will be where f(x0)=0 which will be a root.
Input: The user will need to input a function, f They will also need to give an initial point x0, as well as a max number of iterations and a tolerance. As well as choose whether or not they want a table.
Output: The function will return two approximations of a root for the given function, one from g1(x) and another from g2(x).
Usage/Example: We will answer the following prompts as follows for the example of f (x) = x*np.exp(-x):

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Input initial guess for root here: 2
Input tolerance here: .00000001
Input max number of iterations here: 25
Do you want a table?
  (1) yes
  (2) no
Enter here: 2
```

We need to input the functions as such using the numpy module and construction of python. The following input will yield the following output:

```
g1 approximates the root of f as x =  9.860761315262648e-32
g2 approximates the root of f as x =  4.639207134292721
```

Implementation/Code: The following is code for fnIter.py

```
# Import needed modules
import numpy as np

# Ask the user for the function they want to find the roots of
def fn():
    f = input("Enter function you want to find the roots for (in terms of x): ")
    return f

# Ask the user for an initial guess
def initialGuess():
    x0 = eval(input("Input initial guess for root here: "))
    return x0

# Ask the user for the wanted tolerance
def tolerance():
    tol = eval(input("Input tolerance here: "))
    return tol

# Ask the user for the max number of iterations
def maxIter():
    mI = eval(input("Input max number of iterations here: "))
    return mI

# Below are the two functions that we are using to find fixed points for
def g1(f, x):
    gval = x - eval(f)
    return gval

def g2(f, x):
```

```
    gval = x + eval(f)
    return gval

# Ask the user if they want a table
def wantTable():
    print("Do you want a table?")
    print("\t1) Yes")
    print("\t2) No")
    choice = eval(input("Enter here: "))
    return choice

# Create the table if the user wants one
def table(a,b,c):
    print("(1)" , a, " (2) ", b, " (3) ", c)

# Here is the actual function for calculating the roots
def functionalIteration():

    # Here we are calling the above funcitions to ask the user for values
    f = fn()
    x0 = initialGuess()

    # Saving our initial guess as a second variable to calculate the fixed point using
    # G1 and G2
    y0 = x0

    # Initialize variables
    tol = tolerance()
    mI = maxIter()
    choice = wantTable()

    # Here we set variables that will be used in our while loops so it won't run infinitely
    error = 10 * tol
    iter = 0

    # Case where a user wants a table
    if choice == 1:
        print("Table for g1:")

        # Here is the afformentioned while loop that will run for as many iterations as the user
        # wants or until our error is smaller than the given tolerance
        while error > tol and iter < mI:

            # Calculate our initial guess for the root
            x1 = g1(f, x0)

            # Calculate a new error based on guess above
            error = np.abs(x1 - x0)

            # Set guess as our new "initial guess" so the loop will return a new guess and not x1 again
            x0 = x1

            # Create a line of our table
            table(iter + 1, x0, error)

            # Increase number of iterations as to make sure we don't run forever
            iter = iter + 1
```

```
        # Reset variables for loop so we can rerun the loop using g2
        # All the reasoning above is the same for the below code
        iter = 0
        error = 10 * tol
        print("Table for g2:")
        while error > tol and iter < mI:
            y1 = g2(f, y0)
            error = np.abs(y1 - y0)
            y0 = y1
            table(iter + 1, y0, error)
            iter = iter + 1

    # Case where user doesn't want a table, all the same reasoning for things as above
    elif choice == 2:
        while error > tol and iter < mI:
            x1 = g1(f, x0)
            error = np.abs(x1 - x0)
            x0 = x1
            iter = iter + 1
        error = 10 * tol
        iter = 0
        while error > tol and iter < mI:
            y1 = g2(f, y0)
            error = np.abs(y1 - y0)
            y0 = y1
            iter = iter + 1

        # Lastly return the two approximations of the root for the function
        print("g1 approximates the root of f as x = ", x0)
        print("g2 approximates the root of f as x = ", y0)
```

Module Name: newtons.py
Author: Jake Daniels
Language: Python
Description/Purpose: This function will compute an approximation of a root for any given function using Newton's Method.
Input: The user will need to input a function, f, and the derivative of that function, f'. They will also need to give an initial guess for the root of f, as well as a max number of iterations and a tolerance.
Output: The function will return a single approximation of a root for the given function.
Usage/Example: We will answer the following prompts as follows for the example of f (x) = xex:

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Enter derivative here: np.exp(-x) - x*np.exp(-x)
Input initial guess for root here: .5
Input tolerance here: .00000001
Input max number of iterations here: 25
```

We need to input the functions as such using the numpy module and construction of python. The following input will yield the following output:

```
Root approximation:  -8.80999858950826e-27
```

Which is approximately zero, so using this approximation we can guess that the function above has a root at or near x=0.
Implementation/Code: The following is code for newton.py

```
# This is the module that allows us to input functions involving elementary functions like (e, sin, cos,...)
import numpy as np

# This is the function that will be used to ask the user for the function
def fn():
    f = input("Enter function you want to find the roots for (in terms of x): ")
    return f

# This is the function that will ask the user for the derivative
def fPrime():
    fp = input("Enter derivative here: ")
    return fp

# This is the function that will compute a function, at a value, x
def f(fn, x):
    fval = eval(fn)
    return fval

# This is the function that will ask the user for an initial guess
def initialGuess():
    x0 = eval(input("Input initial guess for root here: "))
    return x0

# This function asks the user for the tolerance wanted
def tolerance():
    tol = eval(input("Input tolerance here: "))
    return tol

# This function asks the user for the max number of iterations
def maxIter():
    mI = eval(input("Input max number of iterations here: "))
    return mI

# Here is the function that will ask the user if they want a table, and return their decision
```

```
def wantTable():
    print("Do you want a table?")
    print("\t1) Yes")
    print("\t2) No")
    choice = eval(input("Enter here: "))
    return choice

# Here is the function that will actually create the table, that will print lines with the wanted values
def table(a,b,c):
    print("(1)" , a, " (2) ", b, " (3) ", c)

# Below is the function that actually compiles all these functions to compute the approximation
def newtonsMethod():
    # Below we are asking the user for everything they want
    fun = fn()
    fp = fPrime()
    x0 = initialGuess()
    tol = tolerance()
    mI = maxIter()

    # This is where the user is asked if they want a table
    choice = wantTable()

    # We set an iteration counter and an initial error so that we can make the following while loop
    error = 10 * tol
    iter = 0

    # 1 corresponds to when the user selects, yes, they want a table
    if choice == 1:

      # This while loop will run until the error is less than the tolerance or we
      # hit the maximum number of iterations
      while error > tol and iter < mI:

          # Here we compute the values of the function and derivative and save them as variables
          f0 = f(fun, x0)
          fp0 = f(fp, x0)

          # We compute the approximation of the root
          x1 = x0 - f0 / fp0

          # Update error and iteration number to make sure the loop is updated to see if it needs to stop
          error = np.abs(x1 - x0)
          iter = iter + 1

          # By passing the iterations, approximation, and error into this table function
          # we create the wanted table
          table(iter, x1, error)

          # Reset initial variable as initial approximation to continue the loop and obtain
          # the next approximation
          x0 = x1

    # 2 corresponds to when the user does not want a table so we remove the table function
    # and add the last print statement
    elif choice == 2:
```

```
 while error > tol and iter < mI:
     f0 = f(fun, x0)
     fp0 = f(fp, x0)
     x1 = x0 - f0 / fp0
     error = np.abs(x1 - x0)
     iter = iter + 1
     x0 = x1

# Take the last approximation given once loop ends and print it for the user to see
print("Root approximation: ", x0)
```

Module Name: secondDerivative.py
Author: Jake Daniels
Language: Python
Description/Purpose: This function will compute an approximation of a the second derivative for a given function, the user will also need to provide an inital value, a value for h, and an exact value (which we use to compute the error not actually necessary for computing the approximation itself)
Input: The user will need to input a function, f, an initial value, x0, an h value, and an exact value of the second derivative.
Output: The function will return a three vectors, one for the h values used, one for the f" values, and one for the difference between our approximation and the exact value.
Usage/Example: I created the following test function that puts our values into a table:

```
def test():
    exact = 16 * (-np.pi ** 3 + 3 * np.pi ** 2 - 1040 * np.pi + 1040) / (np.pi ** 2 + 1040) ** 2
    hVals, fVals, diff = secondDerivative(lambda x: (x - np.pi / 2) * np.tan(x) *
        np.tan(x) / (x * x + 65), np.pi / 4, 1.0, exact)
    print("h values | approximations | exact | difference")
    for i in range (0, 19):
        print(hVals[i], " | ", fVals[i], " | ", exact, " | ",  diff[i])

test()
```

The ouput will look as follows which I have shortened to save space:

```
h values | approximations | exact | difference
1.0  |  0.08888433309385912  |  -0.03235131011796782  |  0.12123564321182695
...
3.814697265625e-06  |  -0.06767797470092773  |  -0.03235131011796782  |  -0.03532666458295992
```

Implementation/Code: The following is code for secondDerivative.py
import numpy as np
def secondDerivative(f, x0, h, exact):

```
  # Here I compute the value of f at the given intial value
  f0 = f(x0)

  # Here I create the three empty vectors that we will be appending values to
  diff = []
  fV = []
  hV = []

  # Here I create a for loop to approximate the second derivative 20 times at smaller h values
  for i in range (1, 20):

      # Append initial h value to out hV vector
      hV.append(h)

      # Save values for f just to the right and left of our initial point
      f1 = f(x0 + h)
      f2 = f(x0 - h)

      # Calculate our approximation
      fval = (f1 - 2 * f0 + f2) / (h * h)

      # Append the approximation and error to their respected vectors
      fV.append(fval)
      diff.append(fval - exact)

      # Decrease our h by a factor of 2 so we get closer to the intitial x0 value
```

```
    # and hence a better approximation
    h = h / 2

# return 3 vectors
return hV, fV, diff
```

Module Name: trapezoid.py
Author: Jake Daniels
Language: Python
Description/Purpose: Approximate the solution to an integral using trapezoid rule
Input: A function f, the values for our definite integral, a and b, lastly n, the amount of subintervals we want to split our interval a,b into
Output: The approximation
Usage/Example: I used the following code to approximate the integral of exp(-xˆ2):

```
trapezoid(lambda x: np.exp(- x * x), 0, np.pi / 4, 16)
print("sum = ", sum)
```

Which outputted the following:

```
sum = 0.6497100964398593
```

Implementation/Code: The following is code for trapezoid.py

```
import numpy as np

def trapezoid(f, a, b, n):
    # calculate the step size
    h = (b - a) / n

    # We know that from the trapezoid rule we will only need to add f(a) and f(b) once so we add them
    # at the beginning divided by 2 again due to the rule
    sum = 0.5 * (f(a) + f(b))

    # Now we loop over our whole interval
    for i in range(1, n):

        # Move our x i*h to the right until we eventually span the whole interval
        x = a + i * h

        # Add the function value at the above x for each x value to the sum
        fx = f(x)
        sum = sum + fx

    # Multiply our sum by the step size h and return it
    sum = sum * h
    return sum
```

Module Name: L1Norm.py
Author: Jake Daniels
Language: Python
Description/Purpose: This function will take a vector and return the L1 Norm
Input: a vector
Output: L1 Norm
Usage/Example: I wrote the following code to test it:

```
a = [1, 2, 3]
v4 = L1Norm(a)
print("v4 = ", v4)
```

Which ouputs the following value:

```
v4 = 6
```

Implementation/Code: The following is code for L1Norm.py

```
def L1Norm(a):
    # Initialize sum
    sum = 0

    # Loop over full vector
    for i in range(len(a)):

        # Add absolute value of each component to sum
        sum += abs(a[i])

    # Return norm
    return sum
```