HW_1

March 18, 2015

1 Problem 1

Write a program to compute the solutions to the quadratic equation. The program should specify the coefficients a, b, c of $ax^2 + bx + c = 0$. The quadratic solution should be performed in a single function. Verify your solution for $a = c = 1$, and $b = 4$. Document the function for clarity.

```
In [6]: import cmath
        def solveQuad(a=1,b=1,c=1):
            """This function returns the solution of the
            quadratic equation, returning both roots of
            quadratic equation as real numbers when a real
            solution is possible, or both roots as complex
            numbers otherwise"""
            if b**2 - 4*a*c > 0:
                sol1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
                sol2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
                return [sol1,sol2]
            else:
                sol1 = (-b + cmath.sqrt(b**2 - 4*a*c))/(2*a)
                sol2 = (-b - cmath.sqrt(b**2 - 4*a*c))/(2*a)
                return sol1,sol2
        x1,x2=solveQuad(1,4,1)
        print("error one is {:g}".format(1*x1**2+4*x1+1))
        print("error two is {:g}".format(1*x2**2+4*x2+1))
        print("The solutions are {:g} and {:g}".format(x1,x2))
```

```
error one is -4.44089e-16
error two is 0
The solutions are -0.267949 and -3.73205
```

2 Problem 2

2.1 Part a

Recursion is a powerful programming technique in which a function calls itself as part of the solution process. This often simplifies algorithms and may improve computational speed. Write a recursive function that computes the factorial of an integer (positive or zero). Document the program appropriately. The program should be able to handle negative input.

```
In [7]: def customFactorial(x):
            """This function returns the factorial of x
            by calling itself recursively. It returns a value of
            False if the input is not defined by the traditional
```

```
    definition of Factorial.
    """
    if type(x) is not int:
        return False
    elif x > 1:
        return x * customFactorial(x-1)
    elif x == 1 or x == 0:
        return x
    else:
        return False
```

2.2 Part b

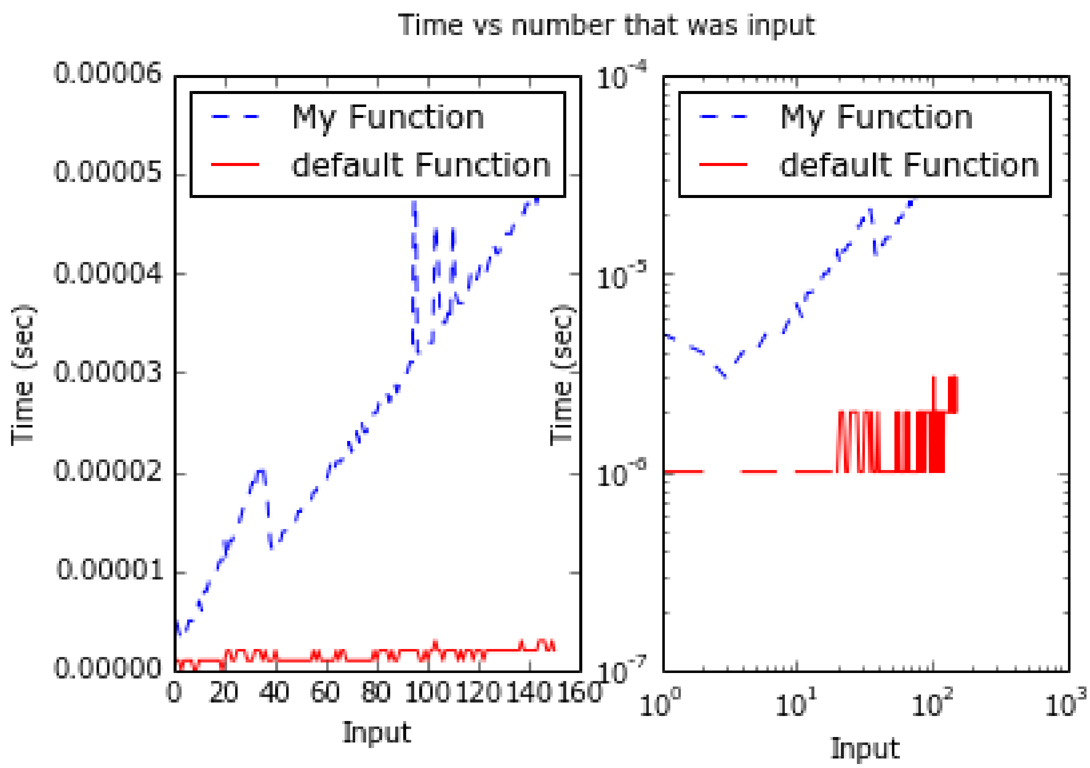
Recursive functions have a so-called recusion depth (how how many times the function calls itself). What happens to your program if you enter too high a number (like 1000)? Based on this, would you recommend a recursive routine for this calculation? Eventually we would run out of memory, and recursive function calls would make this run slower. I would recommend a for loop instead

2.3 Part c

Report the run time for your algorithm compared to the built in function (called factorial). Write a script to compute the run time for computing the factorial of integers between 1 and 150 for your function and built in factorial function. Have the script plot the results on linear and log-log scales using the subplot command. You should include axis labels and legends. Comment on the results.

```
In [12]: def runAgainstDefault():
             myTime = np.zeros(150)
             otherTime = np.zeros(150)
             for i in range(1,151):
                 myStart = time.clock()
                 customFactorial(i)
                 myTimer = time.clock()
                 myTime[i-1]=myTimer-myStart
                 myStart = time.clock()
                 factorial(i)
                 myTimer = time.clock()
                 otherTime[i-1]=myTimer-myStart
             xScale = np.arange(1,151,1)
             x = plt.figure()
             plt.subplot(1,2,1)
             plt.plot(xScale,myTime,'b--',xScale,otherTime,'r-')
             plt.legend(['My Function',"default Function"],'upper left')
             plt.xlabel('Input')
             plt.ylabel('Time (sec)')
             plt.subplot(1,2,2)
             plt.loglog(xScale,myTime,'b--',xScale,otherTime,'r-')
             plt.legend(['My Function',"default Function"],'upper left')
             plt.xlabel('Input')
             plt.ylabel('Time (sec)')
             x.suptitle('Time vs number that was input')
```

```
In [13]: runAgainstDefault()
```



The recursive version of this function definitely is much slower than the original, especially for large values.

3 Problem 3

List and describe five problems that require the use of numerical methods in your research or coursework. Each problem should require a different class of numerical method (such as a linear system of equations, or solution of ODEs, etc.).

1. Getting conductivity measurements from voltage readings requires: Solving a system of linear equations Numerical Integration Numerical differentiation to aid in numerical inversion
2. Image processing requires: Solving a system of linear equations Interpolation Differentiation
3. Using steam tables (at any point) requires Interpolation
4. Using MathCad's solver to determine reaction rate as a function of time and temperature requires: Solving a system of ODE's
5. Generating trends from data requires Interpolation