Computational Physics HW1

Yue (Fred) Shi

September 2019

10/10

1 Answers

a. The following fig1 is the code for the three differential methods. The astype('float32') command is used for changing the precision. The default precision in python and numpy are all double. Unfortunately it seems impossible to change its default setting. Thus in order to deal with the precision, I have to make all the variables in the method to float 32 (single precision float) manually.

ok good.

```
def forward_difference(func, x, h):
    f_prime = (func(x+h).astype('float32') - func(x).astype('float32'))/h
    return f_prime

def central_difference(func, x, h):
    f_prime = ((func(x+h).astype('float32') - func(x-h).astype('float32'))/(2*h))
    return f_prime

def extrapolation_method(func, x, h):
    f_prime = ((-func(x+2*h).astype('float32')+8*func(x+h).astype('float32')-8*func(x-h).astype('float32')+func(x-2*h).astype('float32'))
    return f_prime
```

Figure 1: Code for differential methods

b. Fig2 is the plot for ϵ vs h for the derivative of cos function at x=0.1.

We see that the for the blue curve (forward difference method), ϵ grows with h and can reach up to 10^{-4} precision(10^{-3} in the cos case). And for the red curve (central difference method), ϵ grows with h^2 and can reach up to 10^{-5} precision. Lastly for the green curve (extrapolation method), ϵ grows with h^4 and can reach up to 10^{-6} precision. And clearly when h becomes small enough, they are all limited by the roundoff error and e starts to grow.

✓

This is generally the same for the derivative for the exponential function as well, as can be shown by the Fig3 below.

The figs below with the black dotted line are $\epsilon_0 * h^2$ and $\epsilon_0 * h^4$ respectively, which indicates my previous claim.

c. For both of the function, truncation error dominates the total e in the range when the step size is bigger than 10^{-2} and when h is smaller than 10^{-2}, the roundoff error dominates the ϵ.

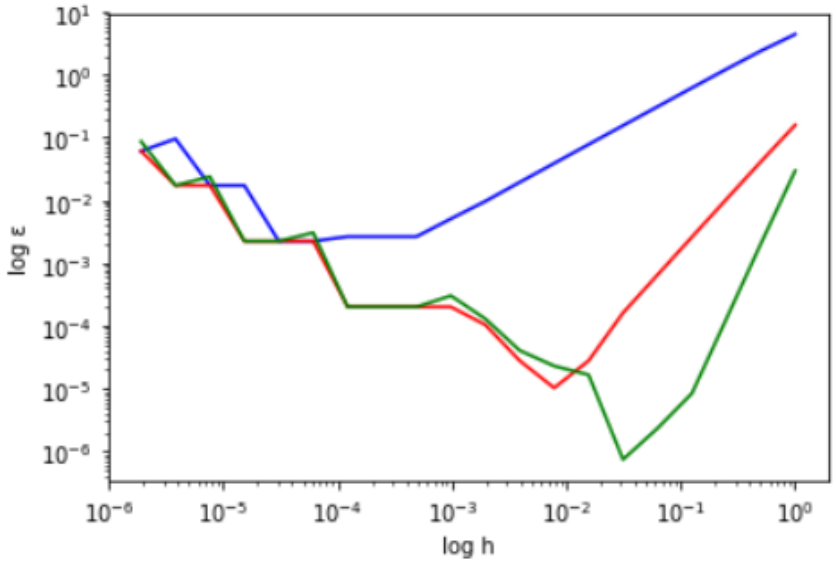

Figure 2: ϵ vs h for (cos(x))' at x=0.1

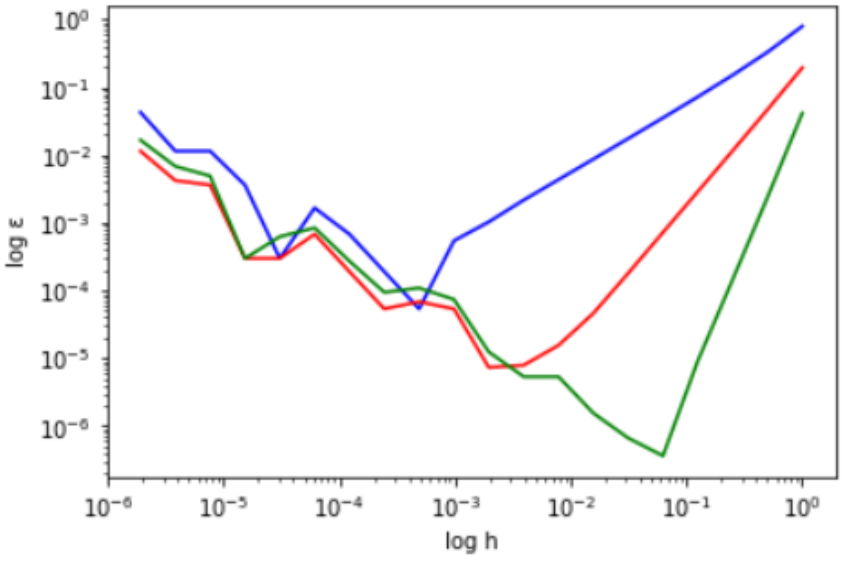

Figure 3: ϵ vs h for (exp(x))' at x=0.1

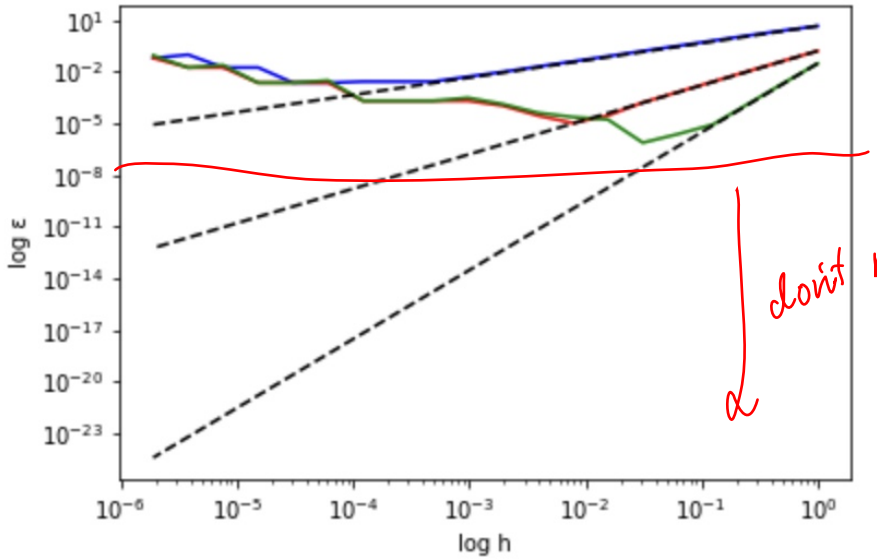

Figure 4: Graph with compared slopes

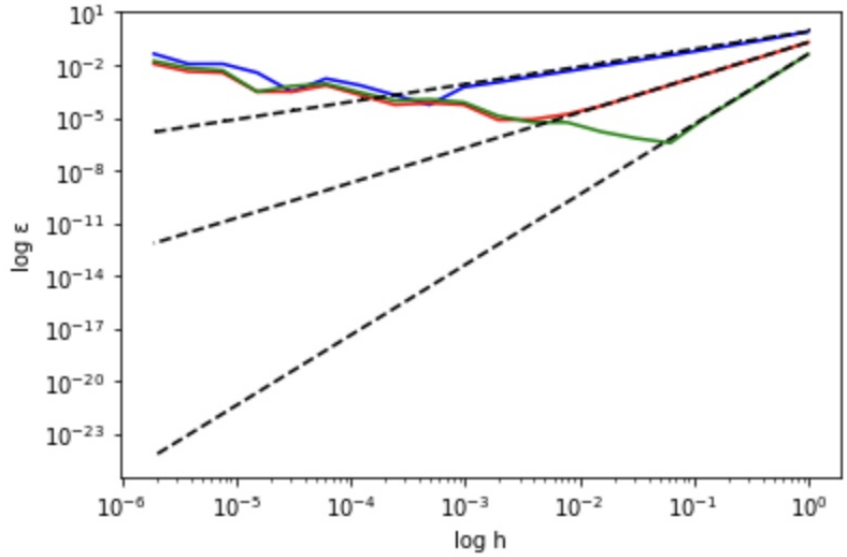

Figure 5: Graph with compared slopes

2 Answers

a. Here is the code for the integrating methods. Again, the astype('float') are

used to change precision.

```
def midpoint_int(func, steps, x1=1, x0=0):
    h = np.float32((x1-x0)/steps)
    intgrl = 0
    for i in range(0, steps):
        intgrl = intgrl + func(x0+h*(i+1/2)).astype('float32')*h
    return intgrl

def trapzoid_int(func, steps, x1=1, x0=0):
    h = np.float32((x1-x0)/steps)
    intgrl = 0
    for i in range(0, steps):
        intgrl = intgrl +  (func(x0+i*h).astype('float32')+func(x0+(i+1)*h).astype('float32'))*h/2
    return intgrl

def simpson_int(func, steps, x1=1, x0=0):
    h = np.float32((x1-x0)/steps)
    intgrl = 0
    for i in range(0, steps):
        intgrl = intgrl +  1/6*h*(func(x0+i*h).astype('float32')+4*func(x0+(i+1/2)*h).astype('float32')+func(x0+(i+1)*h).astype('float32'))
    return intgrl
```

Figure 6: Code for integrating methods

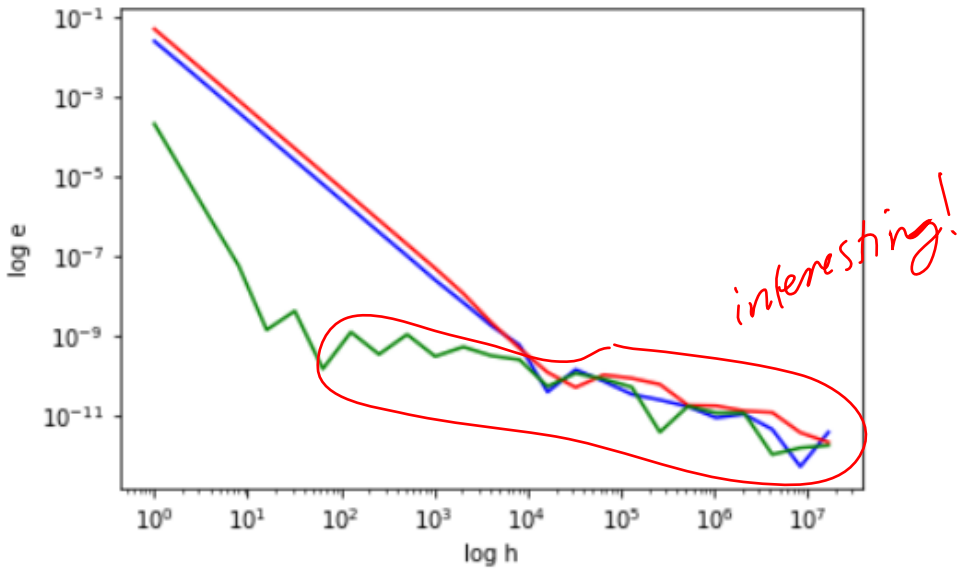

Figure 7: e vs h for the function in Q2

b. Fig8 is the plot for e vs h for the integral of the function in the question.You can see the roundoff error become more and more obvious when N for the blue and red curve(midpoint and trapezoid rule) is around 10^5 and N for the green curve(Simpson's Rule) is around 10. But it seems that python has some way to prevent the accumulation of the roundoff error, so we did not see roundoff errors going much higher as my N increases.

c. We see that the for the blue curve (midpoint method) and the red curve

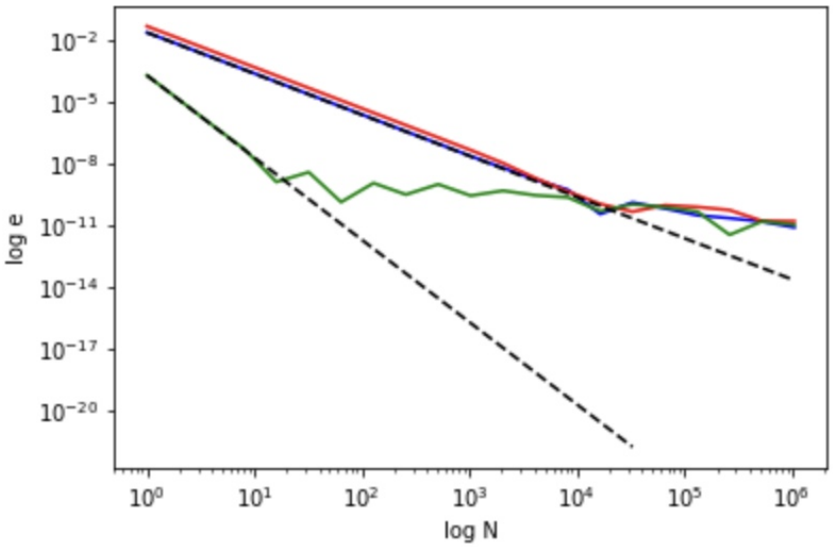

Figure 8: Graph with compared slopes

(trapezoid rule), ϵ grows with N^{-2} and can reach up to 10^{-9} total error (but can go up to 10^{-11} slowly when N become sufficiently large). Midpoint rule performs slightly better then trapezoid rule as the curvature of this function will let trapezoid rule lose more area for each step while midpoint rule can make up a bit of the area loss on each step. And for the green curve (Simpon's method), ϵgrows with $N{-}4$ and can also reach up to 10^{-9} precision, but much more quickly compares to the other two methods. The figs below with the black dotted line are $e_0 * N^{-2}$ and $e_0 * N^{-4}$ respectively, which indicates my previous claim.

3 Answers

I use np.interp to interpolate the pk function and then I use the data comes out with trapezoid rule to calculate the integration with the required e(r) function. And we can see clearly the peak for the pk function around $k = 0.1$ and the peaks for the $e(r)r^2$ when r is roughly 20 and the second highest peak when r is roughly around 105.

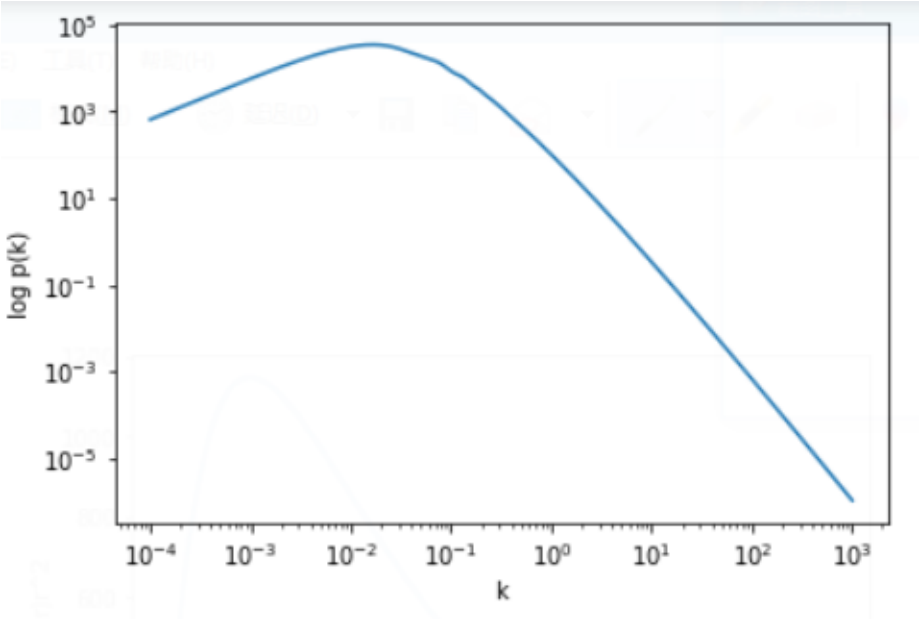

Figure 9: log k vs log p(k)

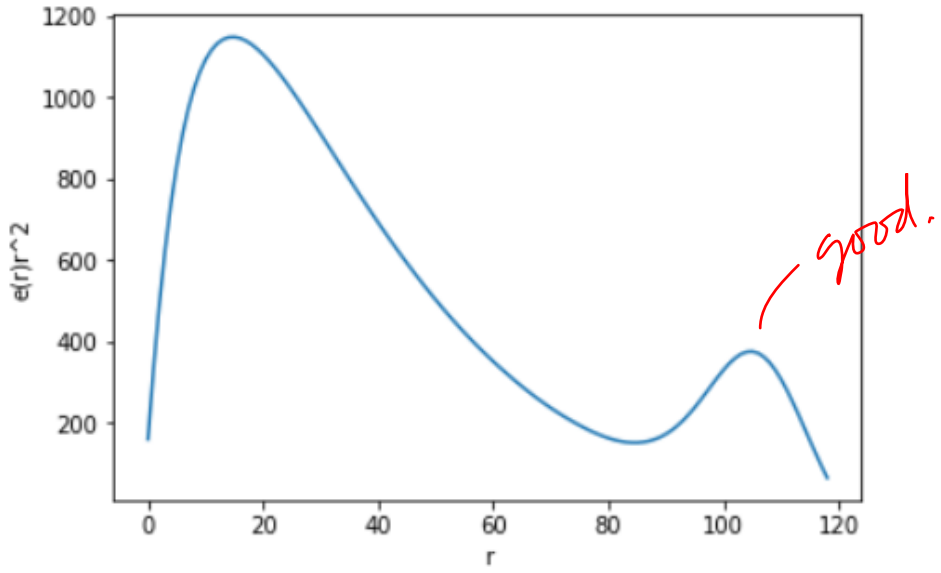

Figure 10: r vs $e(r)r^2$