# HW1

Jinjie Zhang

September 27, 2019

9.5/10

## Problem 1

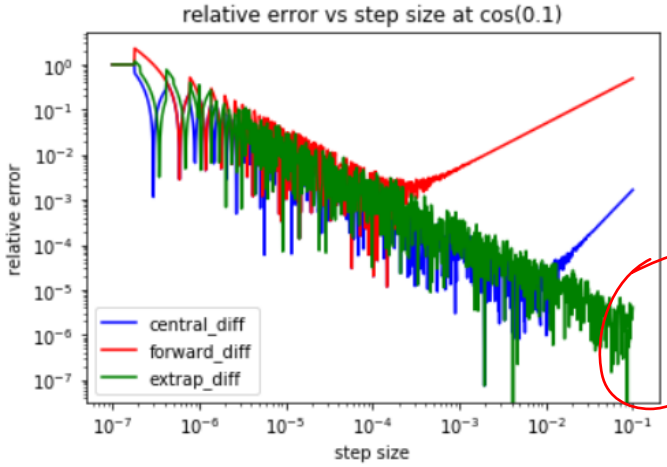
a) Code in Python 3.

```
def forward_diff(f,x,h):
    y1=np.float32(f(x+h))
    y2=np.float32(f(x))
    y3=np.float32(h)
    return (y1-y2)/y3

def central_diff(f,x,h):
    y1=np.float32(f(x+h))
    y2=np.float32(f(x-h))
    return (y1-y2)/np.float32(2*h)

def extrap_diff(f,x,h):
    y1=np.float32(f(x+2*h))
    y2=np.float32(f(x+h))
    y3=np.float32(f(x-h))
    y4=np.float32(f(x-2*h))
    return (-y1+8*y2-8*y3+y4)/np.float32(12*h)
```

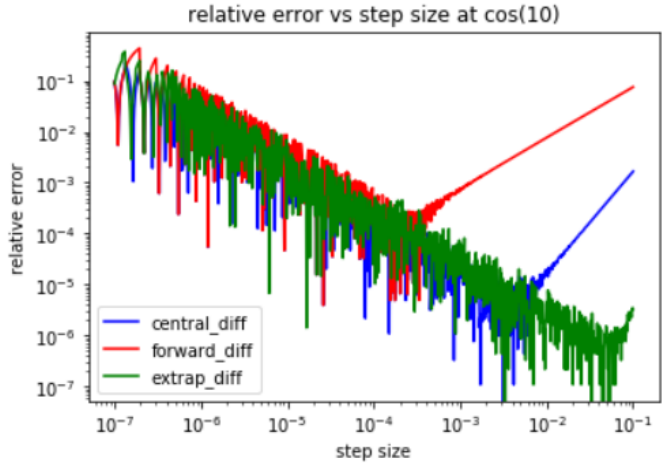
Setting $h$ to be $\boxed{0.01}$(no specific reason), we have

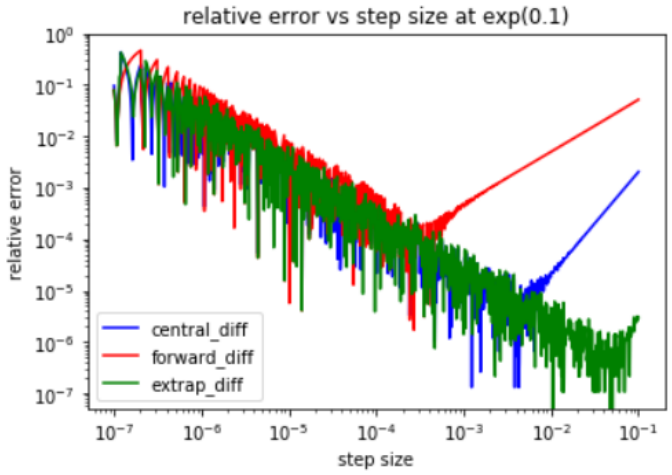
| | forward difference | central difference | extrapolated difference |
|---|---|---|---|
| $\cos'(0.1)$ | -0.10480881 | -0.09983182 | -0.09983331 |
| $\cos'(10)$ | 0.54820776 | 0.5440146 | 0.5440245 |
| $\exp'(0.1)$ | 1.1107087 | 1.1051893 | 1.1051714 |
| $\exp'(10)$ | 22137.11 | 22026.855 | 22026.498 |

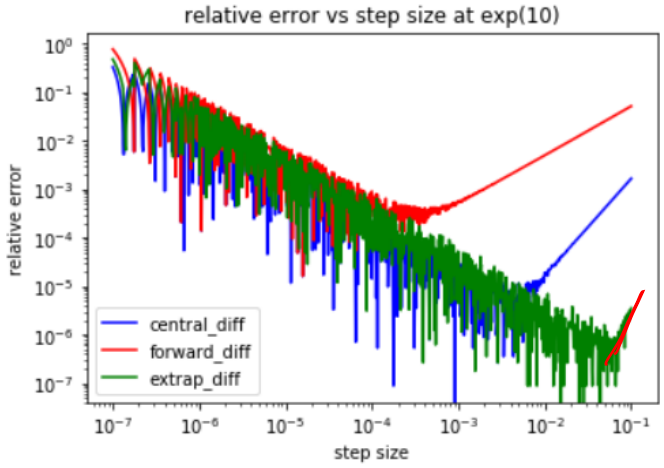
b) Relative error has the expression $\epsilon_{\text{rel}} = \frac{|f' - f'_{\text{num}}|}{f'}$.Vary $h$, we have several plots below to show some patterns.




looks like you need to go to h > 0.1






For each method, the roundoff error is proportional to $\epsilon_m \frac{f}{h}$. The truncation errors for forward difference is $O(h)$, for central difference is $O(h^2)$, and for extrapolated difference is $O(h^2)$ which explains the discrepancy in performance.

c) For each method, the roundoff error is proportional to $\epsilon_m \frac{f}{h}$, so when the step size $h$ becomes smaller, roundoff error grows larger. The truncation errors for forward, central, and extrapolated difference are all proportional to powers of $h$ as discussed in the previous part, so the smaller the step size, the smaller the truncation error.

# Problem 2

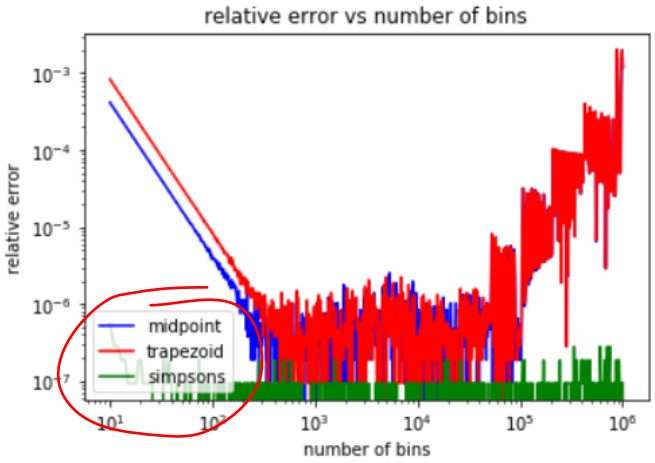
a) Code in Python 3.

```
def mid_int(f,a,b,n):
    h=(np.float32(b)-np.float32(a))/np.float32(n)
    result=np.float32(0)
    for i in range(n):
        result=result+f((np.float32(a)+h/np.float32(2))+np.float32(i)*h)
    result=result*h
    return result

def trap_int(f,a,b,n):
    h=(np.float32(b)-np.float32(a))/np.float32(n)
    result=np.float32(0)
    for i in range(n-1):
        result=result+np.float32(2)*f((np.float32(a)+h)+np.float32(i)*h)
    result=result+f(np.float32(a))+f(np.float32(b))
    result=result*h/np.float32(2)
    return result

def simp_int(f,a,b,n):
    if n%2==1:
        n=n+1
    h=np.float32((b-a)/n)
    x=np.linspace(a,b,n+1,dtype=np.float32)
    y=f(x)
    result=(h/np.float32(3))*np.sum(y[0:-1:2] + 4*y[1::2] + y[2::2])
    return result
```

Fixing number of bins $N = 100, 1000$, we summarize the results in the table below.

| | midpoint | trapezoid | simpsons |
|---|---|---|---|
| numerical value($N = 100$) | 0.6321179 | 0.63212585 | 0.63212055 |
| numerical value($N = 1000$) | 0.6321205 | 0.63212085 | 0.63212055 |

b) $\int_0^1 e^{-t}dt = 1 - \frac{1}{e}$ mathematically. Relative error has the expression $\epsilon_{\text{rel}} = \frac{|I - I_{\text{num}}|}{I}$.Vary number of bins $N$, we have a plot below to show some patterns.




Poor placement of key

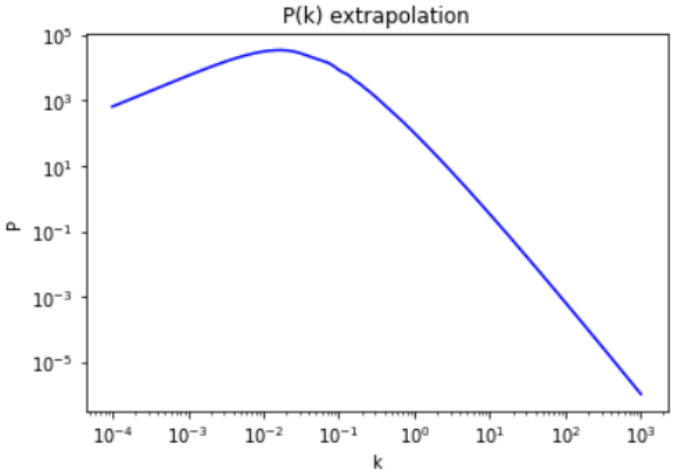
Once again, it looks like you need to go to smaller N for simpsons.

c) When $N$ enters the realm of $10^4$, roundoff errors becomes significant for both midpoint and trapezoid method. When $N$ enters the realm of $10^5$, roundoff errors become dominant and the total relative error starts to increase steadily. As for simpson's method, it's performance is very stable, the relative error is constantly lower than the other two methods and truncation error. The reason is that truncation errors for midpoint and trapezoid method are $O(h^2)$ while it is $O(h^4)$ for simpsons method so simpsons appear to have better performance error-wise. Rounding error becomes significant when $N \approx (e_m)^{-\frac{1}{2}}$ for trapezoid and midpoint error, that's we see the oscillation after $N$ reaches $10^4$. While rouding error becomes significant for simpson's method when $N \approx (e_m)^{-\frac{1}{4}}$ which is why we see it when $N$ reaches $10^2$.

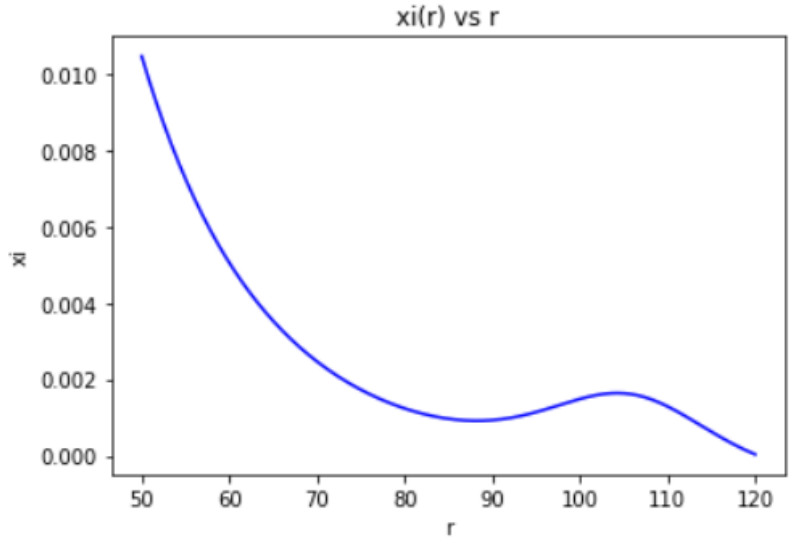
do you really show this?

## Problem 3

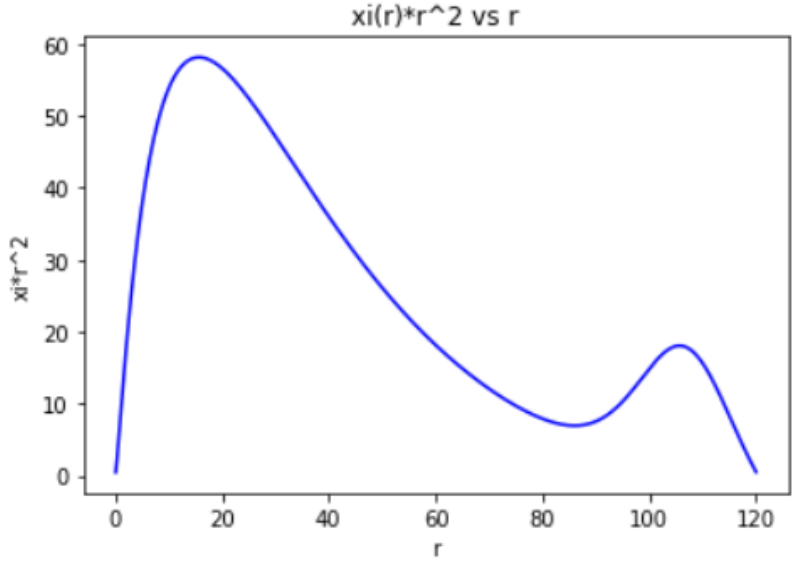
a) Since we don't have the explicit expression for $P(k)$, we use the tabulated data to extrapolate $P(k)$ for $k \in [10^{-4}, 10^3]$ using SciPy's cubic spline.




Then we use simpson's method to do numerical integration, acquiring $\xi(r)$. Note that we integrate $k$ from $10^{-4}$ to $10^3$, there is no need to increase the upper bound as the result is robust for $r \in [50, 120]$.




b) The bump becomes more visually significant when we plot $r^2\xi(r)$ as a function of $r$.




Use SciPy's optimization package, we found that the bump of $\xi(r)r^2$ in $[50, 120]$ occurs at $r = 105.64481$ with value 18.067139.