# HomeWork 1 - Computational Physics

Giorgi Arsenadze

due to: 27 September 2019

9/10

## 1

In this problem we have to implement three algorithms to find differential of functions $\exp(x)$ and $\cos(x)$ at the point 0.1.

### 1.1

For this section of the Problem 1 I used MATLAB as my main coding software. Codes for my Forward, Central and Extrapolated Difference algorithms could be seen in the Appendix A, respectively: Fig 7, Fig 8 and Fig 9.

### 1.2

For this section I did log-log plot of the relative error $\epsilon$ vs step size h. You can see which color corresponds to which algorithm in the description of the plot. For each of them h was changing from 0.1 to around $10^{-28}$. Obviously this is too much but MATLAB works in double precision by default and to change it in the single precision is relatively hard. Fortunately, double precision still gives us the right general behaviour of relative error. It is easy to see that relative errors have minimum values for some values of h. In the next section I will try to explain why.

### 1.3

As we know from the second Lecture in this class total error for derivatives in the worst case has its specific formula:

$$e = hf'' + \epsilon f/h$$

from here we can easily solve the value of h which gives us the minimum relative error. After solving the equation we will get the value for h:

$$h = \sqrt{\epsilon_m f/f''}$$

In our problem one can clearly see that for some values of h errors have minimal value. Usually this is how one should choose h but here we clearly showed that this value exists.

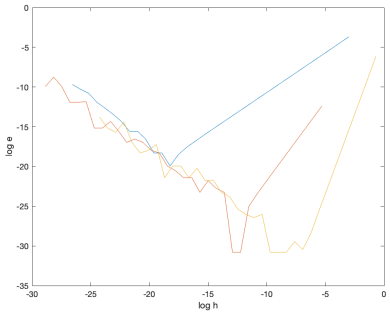


Figure 1: We see log-log plot of the relative error $\epsilon$ vs step size h when calculating differential of $\exp(x)$ the point 0.1, blue-Forward, orange-Central, Yellow-Extrapolated Difference.

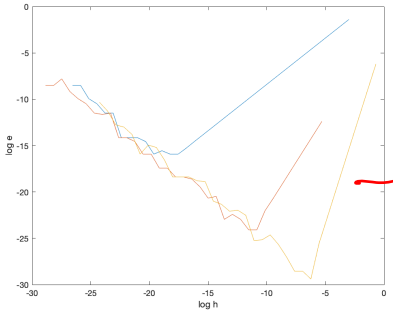


Figure 2: We see log-log plot of the relative error $\epsilon$ vs step size h when calculating differential of $\cos(x)$ the point 0.1, blue-Forward, orange-Central, Yellow-Extrapolated Difference.

For Central difference method, this formula looks a bit different:

$$\epsilon = 2C|f(x)|/h + 1/24h^2|f'''(x)|$$

Therefore, h, which causes minimum relative error has a different formula:

$$h = (24C||f(x)/f'''(x)|)^{1/3}$$

this is why the relative error becomes minimum in different points on our plots. Obviously each method has its own formula for the best value of h. For the Extrapolated difference method as it is the most precise, its relative error gets its maximum value with much more higher h compare to other two methods.

## 2

In this problem we had to take the following integral:

$$I = \int_0^1 \exp(x) dx$$

Using three algorithms: Midpoint, Trapezoid and Simpsons Rule.

### 2.1

I wrote codes for this problem in MATLAB, Phyton and C and as it needs a lot of calculations, especially for the graphs, C worked the best. But still, as C does not have any good plotting functions I plotted it in Phyton. Implementations of each of these methods could be seen in the Appendix B, respectively Fig 11, Fig 12, Fig 13.

### 2.2

In this section we will see log-log plot of relative error $\epsilon$ vs number of bins which are used for integration.

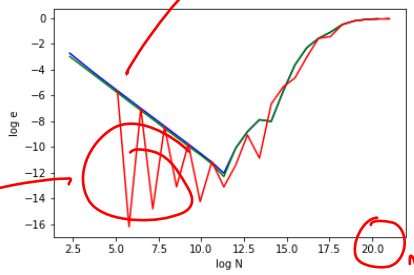


Figure 3: You see log-log plot of relative error vs number of bin for midpoint (green), trapezoid (blue), Simpsons (red)

### 2.3

What we see on this graph is the dependence of the relative error on the number of bins. It could be seen that as number of bins get higher relative error increases. In the middle it seems that Trapezoid method is the worse than midpoint. To see this properly see the Fig 4 which is close up of the midpoint and trapezoid.

To realize what is happening on the plots and why does the error behave like that we have to remember how error behaves while using trapezoid method. In the book of Mark Newman "Computational Physics" it is nicely calculated how an error behaves for integrals. and it is following:

$$\epsilon = 1/12h^2[f'(a) - f'(b)]$$

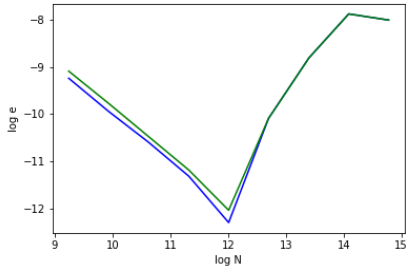


Figure 4: You see log-log plot of relative error vs number of bin for midpoint (green), trapezoid (blue),

which is why first half of the graph shows that increasing N, therefore decreasing h, results decreasing $\epsilon$. As we have log-log graph it is linear rather than quadratic. However whatever is happening after the minimum point, which is around 12 is also extremely interesting. Decreasing h only helps us up to a point where approximation error and round off error become equal which in our case is $10^{12}$. In other words after this point machine is not precise anymore. It is important to emphasize that approximation error becomes equal to round off error way early for Simpsons method as it is the most precise. For us it seams that it is around $N = 10^4 - 10^5$. This is a short description what happens on these plots and why the relative error behaves as it does.

# 3

In this last problem we had to find following function:

$$E(r) = 1/2\pi^2 \int dk k^2 P(k) \sin kr/kr$$

interval in which we have to find it is $r = [50 - 120]$. Before we start to find $E(r)$ we actually have to find function $P(k)$. For this we are using given data to interpolate it using cubic spline. This method actually work fine, for cubic spline I used Scipy's built in function "interp1d". Putting in all the data it gives us following plot for $P(k)$ (see Fig.5).

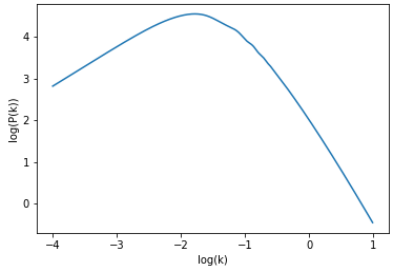


Figure 5: This is the log-log graph of $P(k)$ vs $k$

So it looks as it's suppose to be. Which means that our interpolation method works fine.

Now lets calculate $E(r)$. For this problem I coded in Phyton. Every time I needed integral I used Trapezoid method. Following code could be seen in the Appendix C.

After we got $P(k)$ now we find $E(r)$. To do this, I integrated integrand for different values of $r$. This way actually worked and the plot I got for the $r^2 * E(r)$ vs $r$ looks like this:

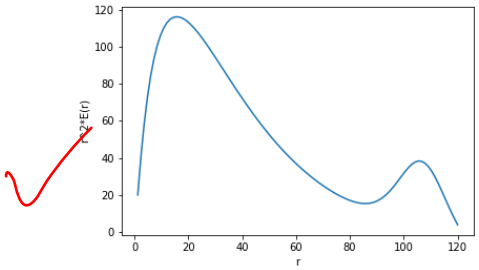


Figure 6: This is the plot of $r^2E(r)$ vs $r$

It is important to notice that in the graph of $P(k)$ we have small wiggles on top of $k = -1$ which are called "baryon wiggles" and these wiggles cause the bumps we have in the Fig.6.

# 4 Appendix A

```
function [k] = forwardDiff(func,x,h)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
    x0=x;
    k=eval((func(x0+h)-func(x0))/h);

end
```

Figure 7: Implementation of Forward Difference method in MATLAB

```
function [k] = centralDiff(f,x,h)
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here
    x0=x;
    k=eval((f(x0+h)-f(x0-h))/(2*h));
end
```

Figure 8: Implementation of Central Difference method in MATLAB

```
function [k] = extrapolatedDiff(f,x,h)
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here
    x0=x;
    k=eval((-f(x0+2*h)+8*f(x0+h)-8*f(x-h)+f(x-2*h))/(12*h));

end
```

Figure 9: Implementation of extrapolation Difference method in MATLAB

```
function [plot1] = relativeErrorPlot(method,f,x0,h,n)
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here
    x=zeros(1,n);
    y=zeros(1,n);
    ff=0;
    fi=0;
    for i=1:n
        h1=h/2^i;
        ff=method(f,x0,h1);
        fi=diff(f);
        e=abs((ff-fi(x0))/fi(x0));
        x([i])=h1;
        y([i])=e;

    end
    plot1=plot(log(x),log(y))
    xlabel('log h')
    ylabel('log e')
    hold on

end
```

Figure 10: This is the code which takes each method as value and gives us back the dependence of relative error to h

## 5 Appendix B

```
float midpoint(float a,float b,int n){

    float I=0;
    float h=(b-a)/n;
    for(int i=0;i<=n;i++){
        float x=(i+1/2)*h;
        I=I+h*f(x);
    }


    return I;



}
```

i<n not i<=n

you did

N+1 bins

Figure 11: This is the implementation of midpoint method in C

```c
float trapezoid(float a,float b,int n){

    float h=(b-a)/(n-1);
    float I=(f(a)+f(b))*h/2;
    for(int i=1;i<=n-1;i++) {
        float x=a+i*h;
        I=I+h*f(x);
    }
    return I;
}
```

Figure 12: This is the implementation of Trapezoid method in C

```
float simpsons(float a,float b,int n){

    float h=(b-a)/n;
    float I=(f(a)+f(b))*h/3;
    for(int i=1;i<=n-1;i=i+2) {
        float x=a+i*h;
        I=I+h*f(x)*4/3;
    }
    for(int i=2;i<=n-2;i=i+2) {
        float x=a+i*h;
        I=I+h*f(x)*2/3;
    }

    return I;

}
```

this looks right-

Figure 13: This is the implementation of Simpsons method in C

```
void relativeErrorPlot(float a,float b,int n,int N){

    float x[N];
    float y[N];
    float ff=0;
    float fi=(1-exp(-1));
    float e;
    for(int i=0;i<=N;i++){
        ff=simpsons(a,b,n);
        e=fabsf((ff-fi)/fi);
        x[i]=n;
        y[i]=e;
        n=n*2^i;
    }
    for(int i=0;i<=N;i++){
        x[i]=logf(x[i]);
        y[i]=logf(y[i]);
    }
    for(int i=0;i<N;++i) {
        printf("%.2f ", x[i]);
    }
    printf("bla");
    for(int i=0;i<N;++i) {
        printf("%.2f ", y[i]);
    }
}
```

Figure 14: This function takes each method and gives us dependence of relative error to the number of bins

# 6 Appendix C

```
pk = interp1d(x, y, kind='cubic');
a1=x[0];
b1=x[700];
E=[];
r=[];
ri=0;
rf=120;
N=100;
n1=100000;
for i in range(0,N+1):
    h=(rf-ri)/N;
    r.append(ri+i*h);



def Func(k,r):
    F=(1/(m.pi)**2)*(k**2)*pk(k)*m.sin(k*r)/(k*r);
    return F;
```

Figure 15: In this code you can see the method used for the interpolation.

```
def trapezoid(a1,b1,n,r):
    h=(b1-a1)/n;
    I=(Func(a1,r)+Func(b1,r))*h/2;
    for i in range(1,n-1):
        x=a1+i*h;
        I=I+h*Func(x,r);
    return I
```

Figure 16: This is just the Trapezoid method used for the third problem

```
def mainFunc(n,r):
    e=trapezoid(a1,b1,n,r);
    E.append(e);
```

```
for i in range(0,N+1):
    r1=r[i];
    mainFunc(n1,r1);


print(E);
len(E);


for i in range(0,N+1):
    E[i]=(r[i]**2)*E[i]

fig = plt.figure()
plt.plot(r,E)
plt.ylabel('r^2*E(r)')
plt.xlabel('r')
fig.savefig('Errr')
```

Figure 17: Here this code just finishes the code and gives us the desired plots

# 7 References

1. Mark E. J. Newman- Computational Physics.
2. Steven Chapra -Applied Numerical Methods with MATLAB for Engineers and Scientists.