Assignment on Romberg Integration

Arunima Sarkar, Roll: EE21S062

October 2021

1 The function

The given function that will be integrated after variable change, $u = r/a$ is :

$$I = 2\int_0^1 J_v^2(ku)udu + 2\left|\frac{J_v(k)}{K_v(g)}\right|\int_1^\infty K_v^2(gu)udu \tag{1}$$

here k=2.7 and g=1.2

2 Q2: Plot of function

The function is continuous everywhere but it is not smooth at u=1.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.special as sp

def func(u):
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

val=[]
xval=list(np.linspace(0,15,1000))

for i in xval:
    val.append(func(i))

plt.clf()
plt.semilogy(xval,val,'g')
plt.grid()
```

```
plt.savefig("Function_semilog")
plt.clf()
plt.plot(xval,val,'g')
plt.grid()
plt.savefig("Function")
```

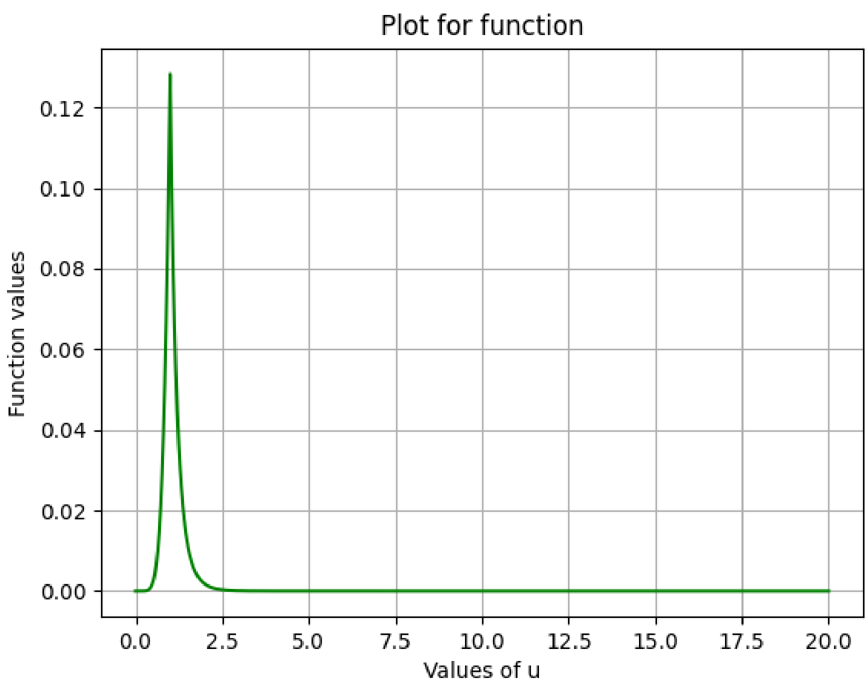

Figure 1: Total power in Electromagnetic mode: For Dielectric Fibres

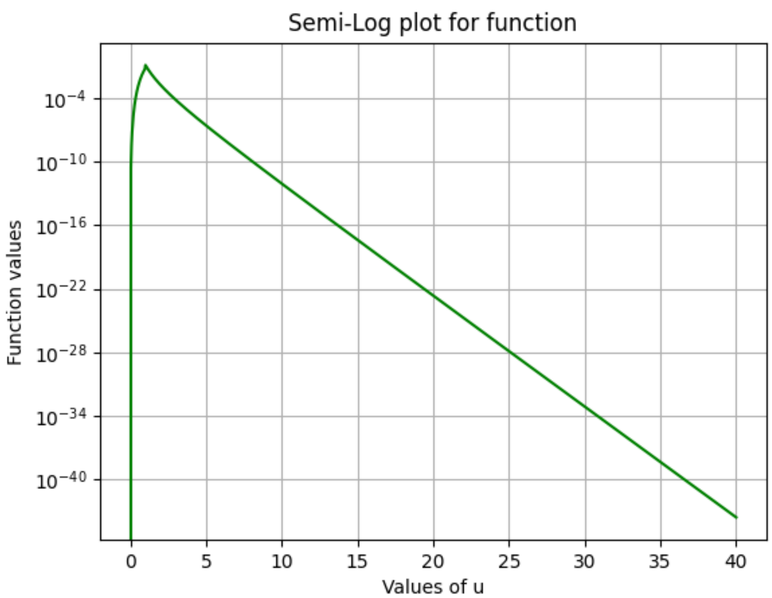

Figure 2: Total power in Electromagnetic mode: For Dielectric Fibres(Semilog plot)

3 Q3

The below figure shows that for upper bound above 20 will keep the error somewhat stable , thus the function is integrated from 0 to 20.

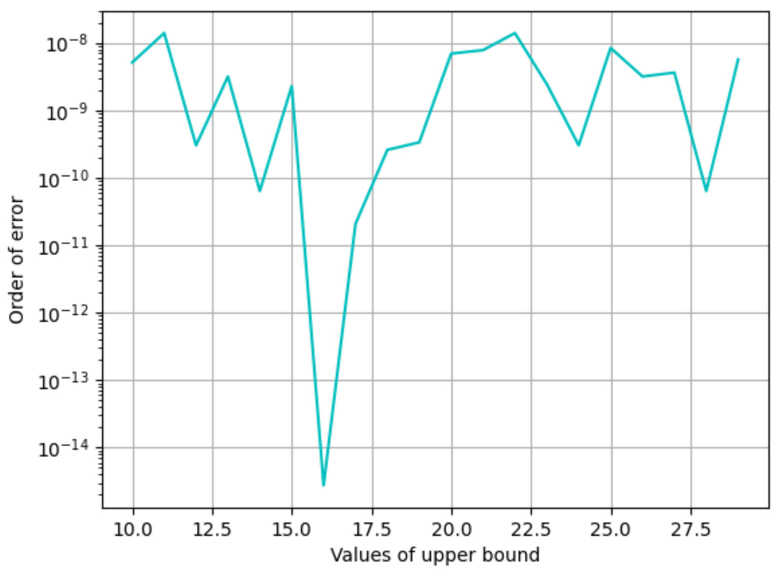

Figure 3: Graph for choice of upper bound of integration

4 Q4

The following code uses the built-in quad function of Scipy.The number of evaluations or function calls required is n=651.

The output from code is:

```
(0.04603886037070528, 8.404300699815792e-09) 1.3555328130673161e-13
```

<u>Code</u>

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.special as sp
import scipy.integrate as s
```

```
def func(u):
    f1=0
    if (u<1):
        f1=2*u*(sp.jv(3,2.7*u))**2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

out=s.quad(func,0,20,full_output=0)
err=out[0]-efunc()
print(out, err)
```

5 Q5:Trapeziodal Method

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

count =0
def func(u):
    global count
    count+=1
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u*2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

s1=0
c=[]
err=[]
```

```
for i in range(1,20):

    s1=r.trapzd(func,0,20,s1,i)
    print (i,s1,s1-efunc())
    err.append(abs(s1-efunc()))
    c.append(count)

plt.clf()
plt.loglog(c,err,'m')
plt.grid()
plt.xlabel("No. of function Calls")
plt.ylabel("Error in Trapeziodal")
plt.savefig("Trapeziodal-err")
```

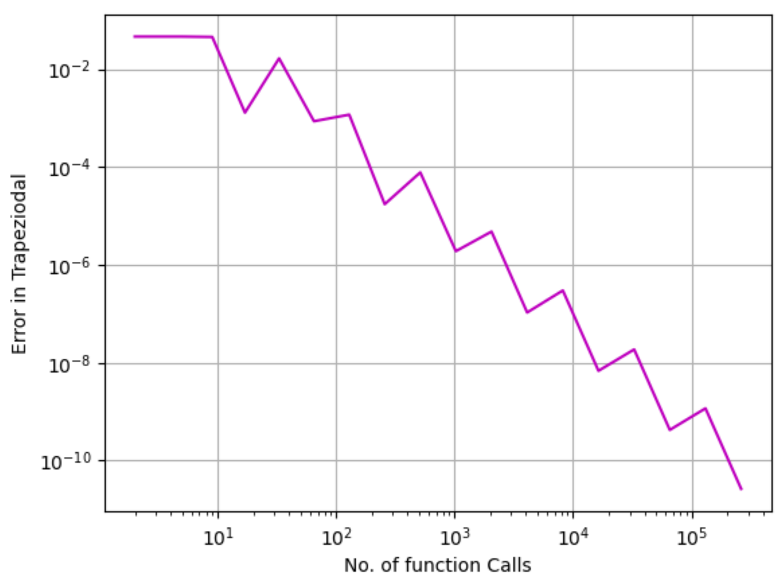

Figure 4: Trapeziodal Error

While using the quad function from scipy.integrate the number of function call is n=651 where quad gives a accuracy or around order 8. For the same order of accuracy from trapezoidal our function call varies a order of 4 to 5.
The trapezoidal algorithm is not as good as Gaussian Quad but its faster.The error decreases with increasing function calls.

6 Q6: Using qromb from Romberg module

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

count=0
def func(u):
    global count
    count+=1
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u*2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

val,err,c=r.qromb(func,0,20,1e-10)

error=[]
call=[]
value=[]

for i in range(-1,-11,-1):
    value.append(r.qromb(func,0,20,10**i)[0])
    error.append(abs(r.qromb(func,0,20,10**i)[1]))
    call.append(r.qromb(func,0,20,10**i)[2])

plt.clf()
plt.loglog(call,error,'r')
plt.grid()
plt.savefig('Error in qromb')
```

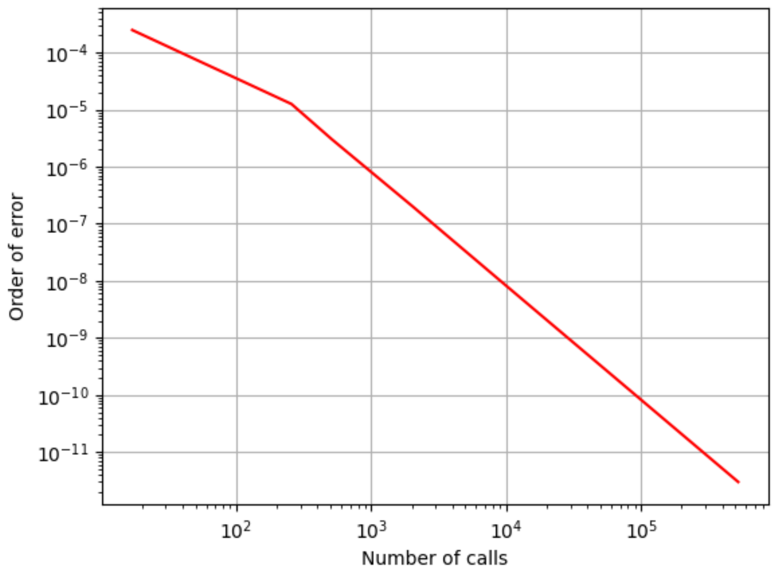

Figure 5: Error from qromb

The error and number of calls required for tolerance level of 1×10^{-10} is :

$3.002\,531\,412\,972\,229\,4 \times 10^{-12}$, N = 524289

Here the number of calls is almost 800 times more or around 3 orders more than quad.

7 Q7: Split the romberg integrals into (0,1) and(1,20)

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

count1=0
count2=0
```

```
def func1(u):
    global count1
    count1+=1
    return (sp.jv(3,2.7*u)**2)*u*2

def func2(u):
    global count2
    count2+=1
    return (2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u

def q():
   out1=s.quad(func1,0,1,full_output=0)[0]
   out2=s.quad(func2,1,20,full_output=0)[0]
   result=out1+out2
   error=result-efunc()    # nevals=147+21=168!
   return result, error

def t():
    err=[]
    c=[]
    s1=s2=0
    for i in range(1,20):

        s1=r.trapzd(func1,0,1,s1,i)
        s2=r.trapzd(func2,1,20,s2,i)
        err.append(abs((s1+s2)-efunc()))
        c.append(count1+count2)
    return s1+s2,err,c

def qr():
    error=[]
    call1=call2=[]
    value1=value2=0

    for i in range(-1,-11,-1):
        value1=(r.qromb(func1,0,1,10**i)[0])
        value2=(r.qromb(func2,1,20,10**i)[0])
        #error.append(abs((value1+value2)-efunc()))
        error.append(abs((r.qromb(func1,0,1,10**i)[1])+
        r.qromb(func2,1,20,10**i)[1]))
        call1.append(r.qromb(func1,0,1,10**i)[2]+r.qromb(func2,1,20,10**i)[2])

    return value1+value2,error,call1

result_q,error_q=q()
```

```
result_t ,error_t ,numcalls=t()

result_qr ,error_qr ,numcalls_qr=qr()

plt.clf()
plt.loglog(numcalls_qr ,error_qr ,"m")
plt.grid()
plt.savefig("Error in qromb with split")
```

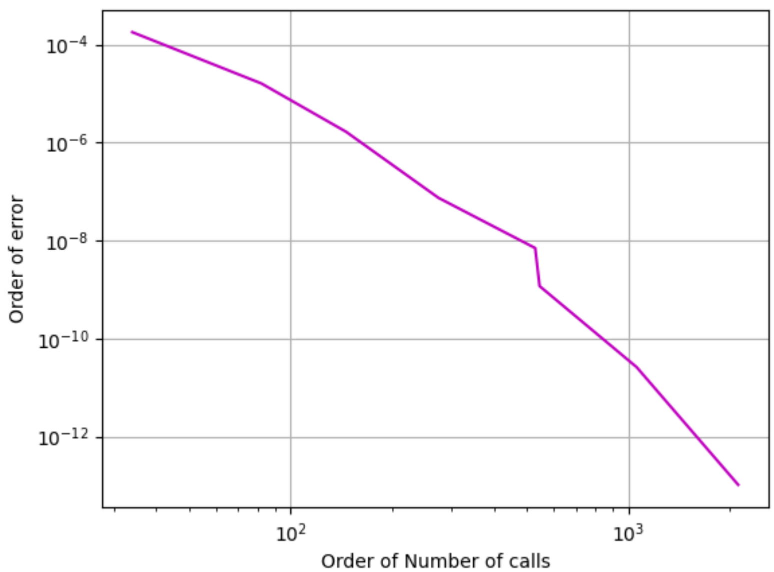

Figure 6: Error from qromb with split, (0,1) and (1,20)

With split in integral of, (0,1) and (1,20), scipy.quad() shows remarkable improvement in it's performance. Its error value reaches machine precision of order 15($1.804\,112\,415\,01 \times 10^{-16}$) and number of evaluations are 168 (147+21).

Trapezoidal doesn't perform better under splitting, but qromb from romberg module shows improvement.The error is $1.042\,586\,306\,836\,673\,5 \times 10^{-13}$ and the number of function calls is 2114.

8 Q8 and Q9: Python implementation of qromb

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)
    -sp.kv(3,1.2)**2)


count=0
def func(u):
    global count
    count+=1
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u*2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

def qrombp(f,a,b):
    out=error=0
    k=5
    xx=yy=[]
    for i in range(1,k+1):
        out=r.trapzd(f,a,b,out,i)
        xx.append(((b-a)/(2**(i-1)))**2)
        yy.append(out)
    out,error=r.polint(xx,yy,0)
    return out,error

out,error=qrombp(func,0,20)
c=x=[]
for i in range(3,21):
    count=0
    x=r.qromb(func,0,20,1e-8,i)[1]
    c.append(count)

plt.clf()
plt.semilogy(range(3,21),c,'c')
```

```
plt.xlabel("order from qromb")
plt.ylabel("Number of function calls")
plt.grid()
plt.savefig("9")
```

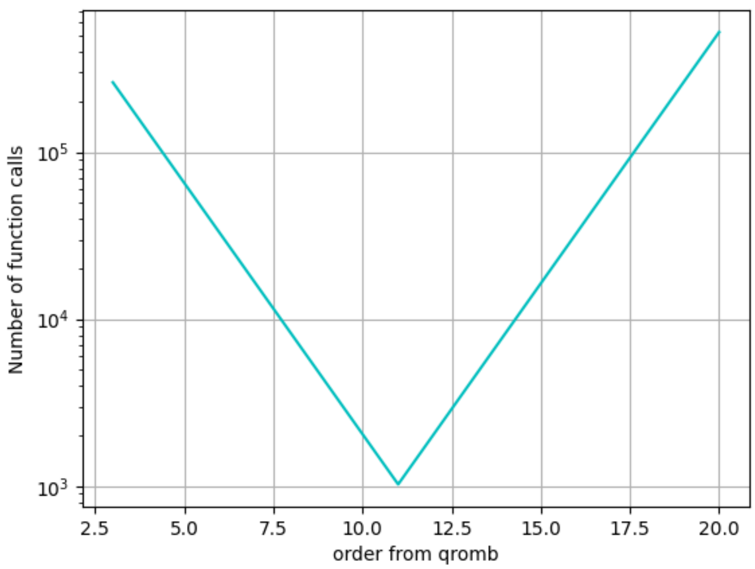

Figure 7: Number of function calls versus order

9 Q10 and Q11: Spline Integration

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r
import scipy.interpolate as si

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

count=0
def func(u):
    global count
    count+=1
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u*2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*
        ((sp.kv(3,1.2*u))**2)*u
    return f1

y=np.vectorize(func)
error=[]
error1=[]

for i in range(3,20):
    x=np.linspace(0,20,2**i)
    tck=si.splrep(x,y(x))
    out=si.splint(0,20,tck)
    error.append(abs(out-efunc()))


for i in range(3,20):
    x1=np.linspace(0,1,2**(i-1))
    x2=np.linspace(1,20,2**(i-1))
    tck1=si.splrep(x1,y(x1))
    tck2=si.splrep(x2,y(x2))
    out=si.splint(0,1,tck1) + si.splint(1,20,tck2)
    error1.append(abs(out-efunc()))
plt.clf()
```

```
plt.semilogy(range(3,20),error,'r',label="Without split")
plt.semilogy(range(3,20),error1,'g',label="With split, (0,1) and (1,20)")
plt.legend(loc="upper right")
plt.xlabel("log2(Number of points)")
plt.ylabel("Order of error")
plt.grid()
plt.savefig("101")
```

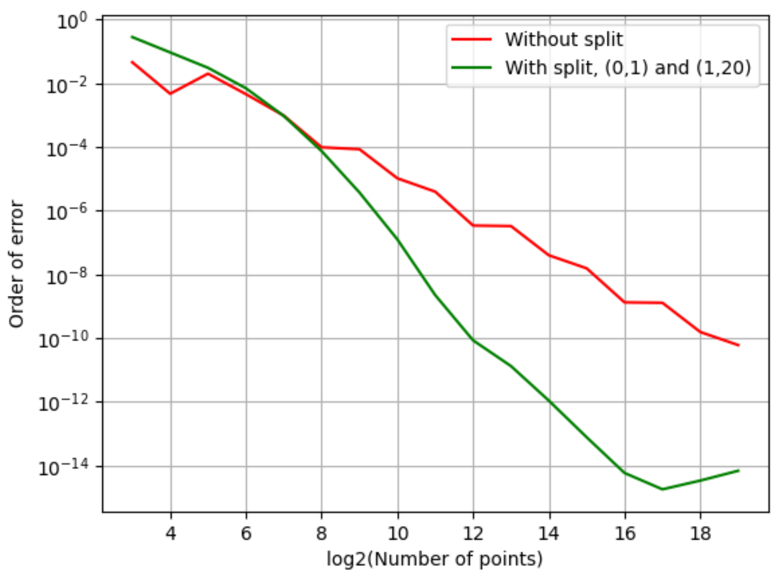

Figure 8: Spline Integration: Error Vs Number of points

The green line shows error while intervals have split and the curve shows for no split , thus it considers the sharp kink at x=1. Thus spline integration works better when sharp kinks are avoided in the integrand. What might happen is the function si.splrep does a cubic interpolation, which is a default operation however than can be changed. Thus the sharp kink that is a higher order polynomial is approximated as cubic but with more sampling the polint error reduces. Thus with splitting of (0,1) and (1,20) , that sharp kink is avoided and it reaches a lower order error faster.

10 Q12 and Q13 : Implementing romberg using h/3 and comparision of various methods

The below mentioned code is implementation of Romberg with h/3 as spacing in trapzd.

```
import numpy as np
import scipy.special as sp
import scipy.integrate as s
import matplotlib.pyplot as plt
import romberg as r

def efunc():
    return sp.jv(3,2.7)**2-sp.jv(4,2.7)*sp.jv(2,2.7)+
    abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2*(sp.kv(4,1.2)*sp.kv(2,1.2)-
    sp.kv(3,1.2)**2)

count=0
def func(u):
    global count
    count+=1
    f1=0
    if (u<1):
        f1=(sp.jv(3,2.7*u)**2)*u*2
    if (u>=1):
        f1=(2*abs(sp.jv(3,2.7)/sp.kv(3,1.2))**2)*((sp.kv(3,1.2*u))**2)*u
    return f1

def trap3(func, a, b, n):
    if(n==1):
        return 0.5*(b-a)*(func(a)+func(b))
    else:
        d = (float)(b-a)/3**(n-1)
        sum=0.0
        x=a+d
        while(x<b):
            sum+=func(x)*d
            x+=d
        sum+=0.5*d*(func(a)+func(b))
        return sum

xx=[]; yy=[];y=[];error=[]

c=[]
```

```
for i in range(1,9):
    #count=0
    xx.append((20.0/3**(i-1))**2)
    yy.append(trap3(func,0,1,i)+trap3(func,1,20,i))
    y.append(r.polint(xx,yy,0)[0])
    error.append(abs(r.polint(xx,yy,0)[1]))


#y,error=r.polint(xx,yy,0)

plt.clf()
plt.loglog(xx,error,"r")
plt.xlabel("order of Decreasing h")
plt.ylabel("Order of error")
plt.grid()
plt.savefig("12")
```

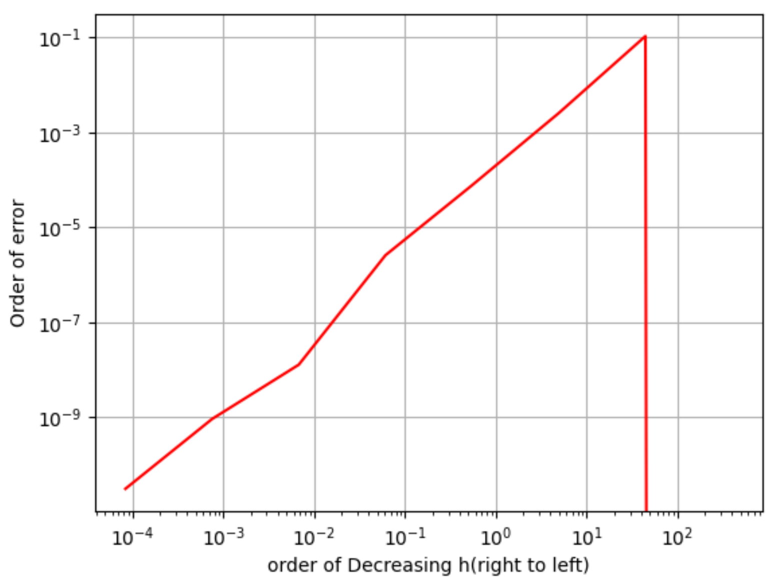

Figure 9: Error in Modified Trapezoidal