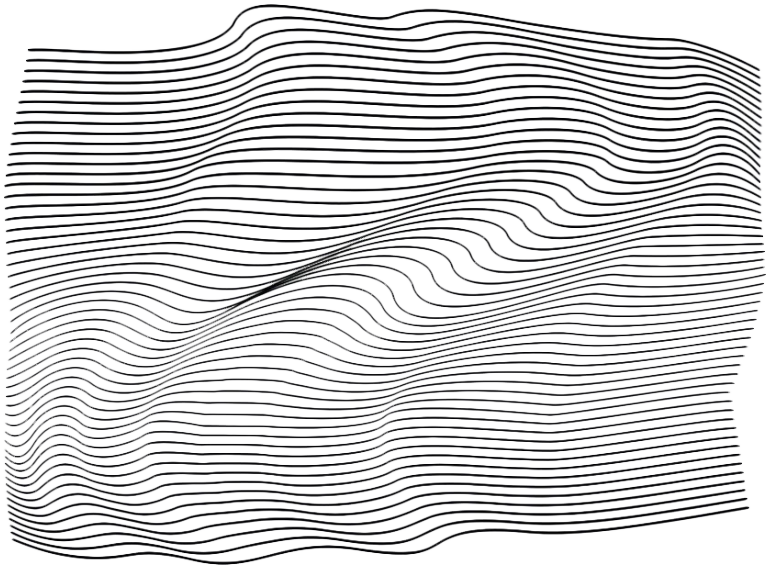
Homework 2
Greens function method

Finn Joel Bjervig*, Thomas Herard†, and Anton Palm Ekspong‡

Computational Physics,
Department of Physics and Astronomy,
Uppsala university, Sweden

Februrary 15, 2021



*Electronic address: joelfbjervig@gmail.com
†Electronic address: herardthomas@gmail.com
‡Electronic address: anton.palm.ekspong@gmail.com

1 Introduction

In physics the charge distribution from a particle is a common problem to study, where in many cases only the radial dependencies are of interest. In this homework we have a differential equation with radial dependency and given boundary values. The numerical solutions will be compared to the analytical solution to see how well it can be approximated using Green's function method.

2 Theory

The differential equation (D.E) given reads

$$\left(\frac{\partial^2}{\partial r^2} - a^2\right)\phi = -4\pi\rho \tag{1}$$

$$\rho = \frac{1}{8\pi}e^{-r} \tag{2}$$

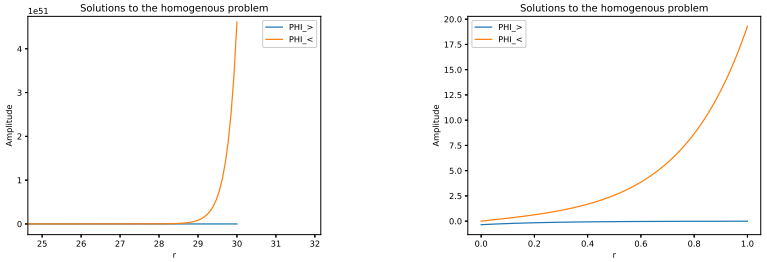
$$\phi(0) = 0 \tag{3}$$

$$\phi(r \to \infty) = 0 \tag{4}$$

The obvious procedure is to study the homogeneous solution to the D.E. which is given in the problem

$$\phi^h_<(r_<) = \frac{1}{\sqrt{2a}}\left(e^{ar_<} - e^{-ar_<}\right) \tag{5}$$

$$\phi^h_>(r_>) = -\frac{1}{\sqrt{2a}}e^{-ar_>} \tag{6}$$



(a) regular view

(b) zoomed in for detail

Figure 1: The two solutions behave very differently. $\phi_>(r)$ stagnates early while $\phi_>(r)$ continues to grow

When the solutions differ significantly, as in this case where one is dominant for small r and the other for large r, it becomes very difficult to use the method

of inward and outward integration. Rather, one can apply the greens function to the equation. The D.E. is linear, and so the solution can be expressed as follows

$$\phi(r) = \int_0^\infty G(r, r')S(r')dr' \tag{7}$$

To find $G(r, r')$, consider the two homogeneous solutions, such that their Wronskian is equal to unity

$$W = \frac{d\phi_>}{dr}\phi_< \frac{d\phi_<}{dr}\phi_> = 1 \tag{8}$$

Then its easy to see that the Greens function is

$$G(r, r') = \phi_>(r_>)\phi_<(r_<) \tag{9}$$

Now we can substitute this expression into equation 7 yielding an explicit formula, which can be solved numerically by for example Bodes/Booles rule.

$$\phi(r) = \phi_>(r) \int_0^r \phi_<(r')S(r')dr' + \phi_<(r) \int_r^\infty \phi_>(r')S(r')dr' \tag{10}$$

3 Discussion

two different algorithms came into fruition to solve equation 7 above that react differently with respect to the step of number of points given. The idea behind the first method is to keep the number of partitioning points constant throughout the changing intervals $r \in [0, r_m]$ and $r \in [rm_m, 30]$ (r_m is changing for each iteration in the for loop, see appendix for code) For Bodes rule, this means that both integrals in equation 7 is divided into the same number of sub-intervals. 400 points was used for each interval to calculate the integrals regardless of how big or small r was in each iteration of the for loop.

It is precisely clear if we consider the calculation of $\Phi(r_m = h)$, where – with the previously algorithm – we would equally segment the intervals $[0, r_m]$ and $[r_m, r = 30]$ with N points while obviously one interval is much larger than the other. To make it more dependent on the size of the interval in question and more computationally friendly, we have set the number to be proportional to the index of Phi we are currently computing with the form `n=scale*4*i+1` (see code in appendix). Thus, the algorithm adapts itself to the interval size, hence the name given "adaptive algorithm" For Bode's method to work, we need the number of points to be a multiple of 4 (plus one). The scale allow us to be more precise to have a finer mesh even for very small intervals like the first between $r = 0$ and $r = h$ (let us note that the h is the smallest step for the computation of Φ but it is not the infinitesimal size used for integration).

While the error of the "constant points" is significantly smaller than for the adaptive algorithm presented above (on the order of 10^-8), and takes a

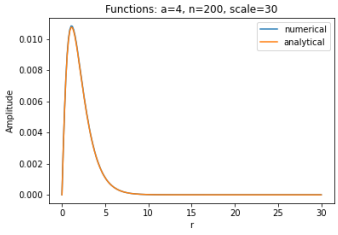


Figure 2: Analytical solution plotted alongside the numerical solution computed over r ranging from 0 to 30 with the adaptive algorithm.

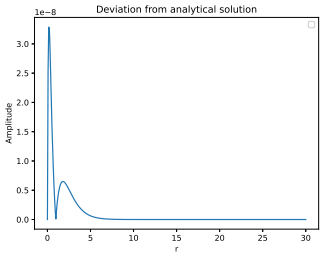


Figure 3: The absolute difference between the analytical and the numerical solution, using constant number of partitions for the intervals.

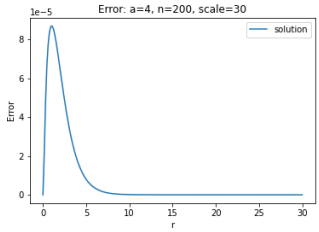


Figure 4: Difference between the analytical solution and the numerical one with the adaptive algorithm.

relatively short time to compute[1], the method presents a curious deviation from the analytical solution, see figure 3. The error increases rapidly in the beginning and then declines just as fast. Thereafter a much smaller spike appears and fades away slowly converges to zero. Additionally the error is extremely close to zero at approximately $r = 1$. As to why the error curve has this particular shape, remains unanswered. The error from the adaptive method on the other hand, follows the behavior of the analytical solution more predictably, as it is

[1]Running seven times and averaging the time taken for the loop to compute the solution is $t = 2.808$ seconds.

proportional to the amplitude of the function computed. This means that the maxima of any function computed with this method will have the highest error as well.

Appendix

Python code

3.0.1 Constant point algorithm

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 12 09:37:12 2021

@author: Finn Joel Bjervig, Anton Palm Ekspong, Thomas Herard
"""
import time
import numpy as np
from math import *
import matplotlib.pyplot as plt

import sys
sys.path.insert(1, '/Users/joelbjervig/documents/universitet/kurser/pagaende/com

from library import bode

# const def
a = 4;
r_max = 30

n = 100
N = 4*n

# function def

#homogenous solutions
phi_GT  = lambda r: -1/sqrt(2*a)*(np.exp(-a*r))
phi_LT  = lambda r: 1/sqrt(2*a)*(np.exp(a*r)-np.exp(-a*r))

#source terms
S       = lambda r: -1/2*r*np.exp(-r)

#hom. sol multiplied with the source term for defining the
phi_GTS = lambda r: -1/sqrt(2*a)*(np.exp(-a*r)) * (-1/2*r*np.exp(-r))
```

```
phi_LTS = lambda r: 1/sqrt(2*a)*(np.exp(a*r)-np.exp(-a*r)) * (-1/2*r*np.exp(-r))

#analytical solution of the differential equation
phi_true = lambda r : (1/(1-16)**2)*(np.exp(-4*r)-np.exp(-r)*(1+0.5*(1-16)*r))

# create vectors for the numerical solution
r = np.linspace(0,r_max,N)
phi = np.zeros(r.shape)

# boundary values
phi_0 = 0
phi_inf = 0
phi[0] = phi_0
phi[-1] = phi_inf


#loops over r_m

start = time.process_time()      # meassures time taken to loop
for i in range(1,len(r)-1):
    phi[i] = phi_GT(r[i])*bode(0,r[i],N,phi_LTS) + phi_LT(r[i])*bode(r[i],r_max,
print(time.process_time() - start)

# plot of error
plt.figure(1)
plt.title("Deviation from analytical solution")
plt.xlabel("r")
plt.ylabel("Amplitude")
plt.plot(r,abs(phi-phi_true(r)))

plt.legend()
plt.show()


# plot of homogenous solutions
print(r)
plt.figure(2)
plt.title("Solutions to the homogenous problem")
plt.xlabel("r")
plt.ylabel("Amplitude")
plt.plot(r, phi_GT(r), label="PHI_>")
plt.plot(r, phi_LT(r), label="PHI_<")

plt.legend()
plt.show()
```

3.0.2 Adaptive algorithm

```
import time
import numpy as np
from math import *
import matplotlib.pyplot as plt

#calculate integrals between for sets of 5 points
def bode(a,b,N,f):
    if (N-1)%4 != 0:
        print("N is not a multiple of the form 4p+1")
        return None
    s=0
    h=(b-a)/N
    for i in range(0,N,4):
        integ=(7*f(a+i*h)+32*f(a+(i+1)*h)+12*f(a+(i+2)*h)+32*f(a+(i+3)*h)+7*f(a+
        integ*=(2*h)/45
        s+=integ
    return s


a=4

#phi functions "greater than" and "less than" r_m
phi_GT  = lambda r: -1/sqrt(2*a)*(np.exp(-a*r))
phi_LT  = lambda r: 1/sqrt(2*a)*(np.exp(a*r)-np.exp(-a*r))
#source term
S       = lambda r: -1/2*r*np.exp(-r)
#multiplictions of S and phi in the integration
multipLT = lambda x:phi_LT(x)*S(x)
multipGT = lambda x:phi_GT(x)*S(x)

#analytical solution
phi_true = lambda r : (1/(1-a**2)**2)*(np.exp(-a*r)-np.exp(-r)*(1+0.5*(1-a**2)*r

r0=0
rmax=30

#number of points on which phi is computed
n=100

#scale used to get a finer mesh for the integration interval on each sides of r_
scale=1

#initialisation
r=np.linspace(r0,rmax,n)
```

```
phi=np.zeros((n,))
phi[0],phi[-1]=0,0

start = time.process_time()      # meassures time taken to loop

#loop avoiding the boundaries for which we have input values
for i in range(1,n-1):
    #creating intervals on which we'll integrat on each side of r_m
    #adaptative sizes on each side depending on where we are on the r array
    rm=r[i]
    r_LT=np.linspace(r0,rm,scale*4*i+1)
    r_GT=np.linspace(rm,rmax,scale*4*(n-i)+1)
    #calculation of phi(r_m)
    phi[i] = phi_GT(rm)*bode(r0,rm,len(r_LT),multipLT) + phi_LT(rm)*bode(rm,rmax

print(time.process_time() - start)

plt.figure(1)
plt.title("Functions: a=4, n=200, scale=30")
plt.xlabel("r")
plt.ylabel("Amplitude")
plt.plot(r,phi,label="numerical")
plt.plot(r,phi_true(r),label="analytical")
plt.legend()
plt.savefig('numerical_vs_analytical_thomas.png')
plt.show()

plt.figure(2)
plt.title("Error: a=4, n=200, scale=30")
plt.xlabel("r")
plt.ylabel("Error")
plt.plot(r,abs(phi-phi_true(r)),label="solution")
plt.legend()
plt.savefig('error_thomas.png')
plt.show()
```