3AN Project 1: Newton Shooting

KDSMIL001 — May 2021

1 Introduction

In this project we will investigate two different numerical methods for finding the solution to the second order nonlinear boundary value problem

$$\begin{gathered}\frac{d^2u}{dr^2}+\frac{1}{r}\frac{du}{dr}+\frac{u}{1-u^2}\left[\left(\frac{du}{dr}\right)^2-\frac{n^2}{r^2}\right]+u(1-u^2)=0 \\ u(0)=0,\quad u(\infty)=1\end{gathered} \tag{1.1}$$

where $n=1,2,3,\ldots$ is the vorticity. The methods we will use to solve this BVP are nonlinear shooting with Newton's method, and Newton-Kantorovich method with linear finite differences. We will implement both and assess their benefits and drawbacks. We are using `Python 3.9`.

2 Implementation

2.1 Nonlinear Shooting

The nonlinear shooting method works by replacing the BVP with a system of initial value problems and guessing one of the initial conditions, then correcting that guess using some root finding method. In our case we have the IVPs

$$\begin{gathered}u''=f(r,u,u')=-\frac{1}{r}\frac{du}{dr}-\frac{u}{1-u^2}\left[\left(\frac{du}{dr}\right)^2-\frac{n^2}{r^2}\right]-u(1-u^2) \\ u(0)=0,\quad u'(0)=p \\ z''=\frac{df}{du'}z'+\frac{df}{du}z \\ z(0)=0,\quad z'(0)=1\end{gathered} \tag{2.1}$$

where p is our shooting parameter, which we will update using Newton's method. The details of this replacement can be found in *Numerical Analysis, Section 11.2* [1]. This kind of IVP is relatively easy to solve using a numerical integrator such as `scipy.integrate.solve_ivp`, seen below.

```
32 def f(r,u,v):
33     return -((1/r)*v + (u/(1-u**2))*(v**2 - (n**2)/(r**2)) + u*(1-u**2))
34
35 def df_du(r,u,v):
36     return -((1+u**2)/((1-u**2)**2)*(v**2 - (n**2)/(r**2)) + (1-3*u**2))
37
38 def df_dv(r,u,v):
39     return -((1/r) + u/(1-u**2)*(2*v))
```

```
40
41 # The function solve_ivp will use to solve the 2 IVPs
42 def du_dr(R,y):
43     U=y[0]
44     V=y[1]
45     Z=y[2]
46     W=y[3]
47     return np.array([V, f(R,U,V), W, df_du(R,U,V)*Z + df_dv(R,U,V)*W])
48 ICs=np.array([alpha,p,0,1])
49 soln=solve_ivp(du_dr, r_span, ICs, t_eval=rs, method='DOP853')
50 p=p-(u_1-beta)/(z_1)
51 ICs[1]=p
```

Code used to solve the IVP in Equation 2.1

Note `u_1` and `z_1` are the values of u and z at the right BC, line 50 is the updating of the shooting parameter by Newton's method, and `alpha` and `beta` are the value of u at the left and right BC respectively.

Regarding the left and right boundaries, we chose to only look at the interval $r \in [0.001, 10]$. We began at 0.001 because we couldn't have a value of $r = 0$ since our equations include a $\frac{1}{r}$ term and that would cause complications, and we ended at 10 since going any larger led to instability. The array of points along which we integrate, our r's, had 100 points. This could not be below 50 as it would lead to the integrator failing to properly approximate the function, and couldn't be much larger than 100 as that leads to instability for an unknown reason, particularly with higher vorticities.

We iterated the code above with the end condition being 400 iterations or `u_1-beta` $> 1 \times 10^{-5}$ for $n = 1, 2, 3$ with the results in Figure 2.1.

2.2 Newton-Kantorovich

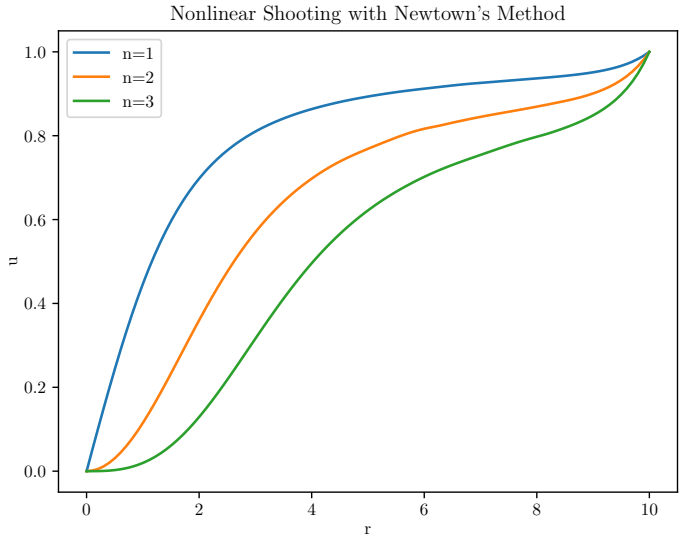
This method works by first linearising the BVP in question, and then we are able to use any method we would like to solve that linear problem. We choose the method of linear finite differences.

The linearisation of Equation 1.1 gives us the system

$$\begin{gathered} z'' - \frac{df}{du'}(r, u^n, (u^n)')z' - \frac{df}{du}(r, u^n, (u^n)')z = -(u^n)'' + f(r, u^n, (u^n)') \\ z(0) = -u^n(0) + u(0) \\ z(\infty) - u^n(\infty) + u(\infty) \\ \text{where} \\ f(r, u^n, (u^n)') = -\frac{1}{r}(u^n)' - \frac{u^n}{1-(u^n)^2}\left[((u^n)')^2 - \frac{n^2}{r^2}\right] - u^n(1-(u^n)^2) \end{gathered} \tag{2.2}$$

where u^n is the current guess of the solution u, and we update our guess at every step with $u^{n+1} = u^n + z$.

This linear system is solvable at each iteration by the method of finite differences, where solve a matrix equation. The details of this can be found in *Numerical Analysis, Section 11.3* [1] but the essential parts of the implementation can be found in Appendix 1. `rs` is our vector of mesh points and `alpha` and `beta` are the left and right BCs respectively.



Vorticity	$n = 1$	$n = 2$	$n = 3$
Initial p	0.1	0.1	0.1
Final p	1	5×10^{-4}	1.3×10^{-7}
Iterations	227	196	210
Time Taken (s)	6.52	5.58	6.33

Figure 2.1: The solutions to Equation 1.1 for $n = 1, 2, 3$ solved using the nonlinear shooting method, as well as details of the convergence to the solution.

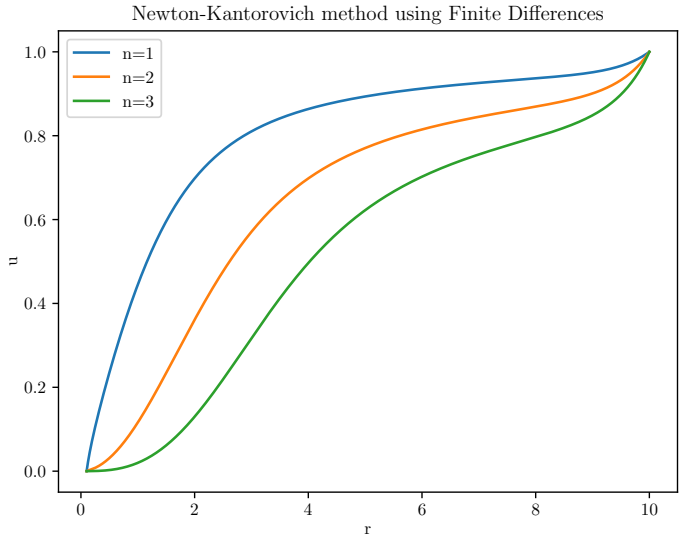
We solve the matrix equation $A\vec{z} = \vec{b}$ where A and $\vec{b}$ are described on pg. 686 of *Numerical Analysis* [1] and $\vec{z}$ is the solution we want, besides the start and end points, which are defined by the BCs in Equation 2.2. Thus we have our correction z to our current guess u^n and we repeat until we reach 100 iterations or $|\vec{b}| < 1 \times 10^{-5}$. For all vorticities the method is stable enough to reach that tolerance, but it comes at the cost of having to increase the number of mesh points by quite a lot. We found that 4000 mesh points on the interval [0.1,10] were needed in order to reach the tolerance before we see blow-up. This blow-up is most likely a result of the inherent instability of the method seeing as it relies on approximations to the derivative in many areas and thus instability can be prolonged but never quite avoided entirely.

We also had some issues when choosing an initial guess, finding that we needed to have an initial guess that was close to satisfying both BCs as well as had the rough shape of the solution, so we chose $u^0 = -e^{-r} + 1$ as it's monotonically increasing and is 1 at ∞.

The solutions found using this method can be found in Figure 2.2.

3 Comparing the two Methods

The first thing to notice about the results of using these two methods is the difference in time taken to find the solution to the same tolerance. Clearly the nonlinear shooting method is quicker, even though it requires far more iterations. However, through implementing these methods we found that the time taken to find a solution using the



Vorticity	$n = 1$	$n = 2$	$n = 3$
Iterations	11	13	14
Time Taken (s)	128.94	164.23	183.19

Figure 2.2: The solutions to Equation 1.1 for $n = 1, 2, 3$ solved using the Newton-Kantorovich method with the linear finite difference method, as well as details of the convergence to the solution.

Newton-Kantorovich method is a small price to pay for the stability that the method seems to bring to the table.

When implementing the nonlinear shooting method, we found that the conditions under which the method finds a stable solution, and finds it to the desired tolerance, are extremely sensitive. If we looked at any interval other that [0.001,10], leaving all other parameters the same (vorticity, number of points to integrate on, tolerance), we would see a wide range of unstable solutions. There didn't seem to be a way to mitigate these instabilities, but with Newton-Kantorovich there were.

When implementing Newton-Kantorovich we found that it can converge to a solution given pretty much any interval, provided we didn't start at 0. That's not to say this method doesn't have instability, but it seems that the only instability it had came as a result of the nature of the method, along with the nature of the linear finite differences method, being that we approximate the solution on a mesh and at some point these approximations get inaccurate. We found that if the number of mesh points was too low then the solution would blow up due to these inaccuracies before we could reach the desired tolerance, but by simply increasing the number of mesh points we found that the tolerance could be reached. Granted this comes with the matrix equation taking a while to solve and thus the method taking much longer than the nonlinear shooting method, but we believe that this trade-off is worth it.

Regarding the number of iterations taken to reach a solution, it's clear that the Newton-Kantorovich method converges to a solution "quicker" than nonlinear shooting, in terms of iterations, and this is definitely a benefit.

Lastly we consider the sensitivity of each method to the initial guess. Nonlinear shooting requires an initial guess of the shooting parameter, in this case the gradient of the solution at the left boundary, and we found that an initial guess of 0.1 led to convergence to a solution for all 3 vorticities. This guess was chosen as it is the gradient of a straight line from (0,0) to (10,1) and this seemed like a reasonable guess. Other guesses in the neighbourhood of 0.1 also led to stable solutions. Newton-Kantorovich was not so well behaved. We initially used the same initial guess as with nonlinear shooting, a straight line connecting the BCs, but this led to some strange solutions that were either not monotonically increasing or weren't stable at all. After some thinking about the fact that the real solution goes to 1 at ∞, we chose the function $-e^{-r}+1$ as our initial guess as it has an asymptote at $u = 1$. This turned out to be a great guess as we began to see stable and correct looking solutions after that.

It's hard to say what a reasonable guess would be for either method as it's entirely possible that we just happened upon a guess that worked well, but it certainly seems that the initial guess for nonlinear shooting was much easier to find.

4 Conclusion

Looking at the benefits and drawbacks of these two methods it's clear that the Newton-Kantorovich method is superior in most discernible ways. It converges to a solution in fewer steps, it seems more stable on a wider range of spatial intervals, it's much less sensitive to certain parameters. It requires a bit more computations time but that seems like a reasonable price to pay for a much more stable method in the general sense.

5 Appendix

Appendix 1: Code used to solve the linear BVP in Equation 2.2

```
def UPrime(arr):
    out=[]
    out.append((-3*arr[0]+4*arr[1]-arr[2])/(2*h))
    for ind, elem in enumerate(arr[1:-1]):
        if ind==0:
            pass
        else:
            out.append((arr[ind+1]-arr[ind-1])/(2*h))
    out.append((3*arr[-1]-4*arr[-2]+arr[-3])/(2*h))
    return np.array(out)

def f(r, guess, ind):
    uPr=UPrime(guess)
    return -((1/r)*uPr[ind] + (guess[ind]/(1-(guess[ind])**2))*(uPr[
       ind]**2 - (n**2)/(r**2)) + (guess[ind])*(1-(guess[ind])**2))

def Q(r, guess):
    out=[]
    uPr=UPrime(guess)
    for ind, elem in enumerate(r):
        out.append(-((1+(guess[ind])**2)/((1-(guess[ind])**2)**2)*(
           uPr[ind]**2 - (n**2)/(elem**2)) + (1-3*(guess[ind])**2)))
    return np.array(out)

def P(r, guess):
    out=[]
    uPr=UPrime(guess)
    for ind, elem in enumerate(r):
        out.append(-((1/elem) + (guess[ind])/(1-(guess[ind])**2)*(2*
           uPr[ind])))
    return np.array(out)

def N(r, guess):
    out=[]
    uPr=UPrime(guess)
    uPrPr=UPrime(uPr)
    for ind, elem in enumerate(r):
        out.append(-uPrPr[ind]+f(elem, guess, ind))
    return np.array(out)
while k < maxIter and tol > 2e-5:
    print('
       ----------------------------------------------------------------------
       ')
    print(f'Iteration: {k}')
    # Gets arrays of p, q, and n excluding the endpoints, as i don't
       need the endpoints for the matrix eqn
    ps=P(rs[1:-1], u)
    qs=Q(rs[1:-1], u)
    ns=N(rs[1:-1], u)
    # creates the tridiagonal matrix, note the slicing for p
```

```
71      A=np.diag((-2-(h**2)*qs), k=0)+np.diag((1+(h*ps[1:])/2),k=-1)+np.
            diag((1-(h*ps[:-1])/2),k=1)
72      B=(h**2)*ns # The RHS vector. not complete yet, needs some
            additions which happens in the next few lines
73      # Calculates the endpoints for the correction z as per the NK
            equations and uses them for those extra naughty bits on the
            front and end of the RHS vector
74      newAlpha=-u[0]+alpha
75      print(f'left BC correction: {newAlpha}')
76      B[0]-=newAlpha*(1+(h*ps[0])/2)
77      newBeta=-u[-1]+beta
78      print(f'right BC correction: {newBeta}')
79      B[-1]-=newBeta*(1-(h*ps[-1])/2)
80
81      tol=np.linalg.norm(B) # Updates tolerance
82      print('tol:',tol)
83      z1=np.linalg.solve(A,B) # Solves the matrix eqn
84      z=np.insert(z1,0,[newAlpha]) # These two lines just add the
            endpoints for z onto the front and end of it
85      z=np.append(z,[newBeta])
86      print('correction:',np.linalg.norm(z))
87      u+=z # Updates our guess
88      k+=1
```

References

[1] Burden, R. and Faires, J., 2005. *Numerical Analysis*. 9th ed. Belmont, CA: Thomson Brooks/Cole.