IJIRSS
International Journal of Innovative Research & Scientific Studies

ISSN: 2617-6548

URL: www.ijirss.com

International Journal of Innovative Research and Scientific Studies
ONLINE ISSN : 2617-6548

Numerical solution of ordinary differential equations using Newton-Raphson method with Numpy and Autograd: Accuracy, convergence, and performance analysis

Choon Kit Chan[1], Pankaj Dumka[2*], Chandrakant Sonawane[3], Subhav Singh[4,5], Dikshant Varshaney[6,7]

[1]Faculty of Engineering and Quantity Surveying, INTI International University, Nilai, Negeri Sembilan 71800, Malaysia.
[2]Department of Mechanical Engineering, Jaypee University of Engineering and Technology, A.B. Road, Raghogharh-473226, Guna, Madhya Pradesh, India
[3]Department of Mechanical Engineering, Symbiosis International University, Pune, India.
[4]Chitkara Centre for Research and Development, Chitkara University, Himachal Pradesh-174103 India.
[5]Division of research and development, Lovely Professional University, Phagwara, Punjab, India.
[6]Centre of Research Impact and Outcome, Chitkara University, Rajpura- 140417, Punjab, India.
[7]Division of Research &amp; innovation, Uttaranchal University, Dehradun, India.

Corresponding author: Pankaj Dumka (*Email:* p.dumka.ipec@gmail.com)

Abstract

The numerical solution of ordinary differential equations (ODEs) using the Newton-Raphson approach is investigated in this work. The aim is to evaluate, in solving first- and second-order ODEs, the accuracy, convergence, and limits of this approach. This Python-based approach uses Autograd for automatic differentiation and NumPy for effective array operations. Several case studies are analyzed, including various ODE challenges. The efficiency of the approach is assessed by comparing numerical findings with analytical solutions. In many situations, the Newton-Raphson method effectively and highly precisely approximates solutions for different ODEs. Some examples, however, show differences between numerical and analytical answers, suggesting possible problems with error accumulation or inherent constraints of the approach. Problem difficulty, step size, and initial guesses all affect convergence. Although the Newton-Raphson approach solves ODEs numerically quite well, it must be carefully validated against analytical solutions. The performance of the procedure depends on elements particular to the problem that must be taken into account in application. The need for choosing suitable numerical methods for solving ODEs in scientific and technical domains is underlined by this work. The results guide future research and useful implementations by offering an understanding of the strengths and constraints of Newton-Raphson-based solvers.

Keywords: Autograd module, Newton-raphson, Numerical method, Ordinary differential equation, Python programming.

DOI: 10.53894/ijirss.v8i3.7504
Funding: This study received no specific financial support.
History: Received: 14 March 2025 / **Revised:** 16 April 2025 / **Accepted:** 18 April 2025 / **Published:** 30 May 2025

Competing Interests: The authors declare that they have no competing interests.

Authors' Contributions: All authors contributed equally to the conception and design of the study. All authors have read and agreed to the published version of the manuscript.
Transparency: The authors confirm that the manuscript is an honest, accurate, and transparent account of the study; that no vital features of the study have been omitted; and that any discrepancies from the study as planned have been explained. This study followed all ethical practices during writing.
Publisher: Innovative Research Publishing

1. Introduction

Applications ranging in numerous fields, including engineering, physics, economics, and biological sciences, the numerical solution of ordinary differential equations (ODEs) is a basic component of scientific computing [1]. Many constructed systems are driven by differential equations from modelling population dynamics to predicting the heat transfer and simulating control systems [2]. Although ideal, analytical solutions are usually challenging for complex, nonlinear, or high-dimensional ODEs; consequently, numerical techniques [3] are usually required.

Among the traditional methods for solving ODEs are the Euler method, Runge-Kutta techniques, and implicit approaches, including the backward differentiation formula (BDF). These methods can, however, have slow convergence, unstable behavior, or inefficient handling of stiff equations [4]. Most especially in cases requiring fast convergence and high precision, Newton-Raphson is a well-known iterative method for solving nonlinear equations and provides an alternative approach for solving ODEs [2]. Reformulating ODEs as nonlinear systems offers a powerful framework for numerical approximation by means of the Newton-Raphson technique, especially when coupled with modern computing tools like Python [5, 6]. Development of computer techniques [7] has greatly enhanced numerical method implementation and efficiency. The availability of high-performance computing tools as NumPy [8, 9] for efficient array computations and autograd [10] for automatic differentiation, has enlarged the capability of solving differential equations. This work, implemented in Python to ensure computational performance and simplicity of usage, analyzes the Newton-Raphson method as a strong numerical strategy for ODEs employing these tools.

Many researchers have investigated numerical methods for ODE solution. The simplicity and dependability of classical techniques such as Euler's method and the Runge-Kutta family make them quite popular. With a thorough study of Runge-Kutta techniques, Butcher [11] demonstrated their success for both initial and boundary value problems. Dahlquist [12] has investigated the stability of explicit and implicit numerical approaches, thereby providing fundamental guidelines for the choice of method in stiff situations. Although less often discussed in standard numerical techniques literature, the application of Newton-Raphson to ODEs has attracted attention in particular settings. Its application for solving nonlinear dynamical systems [13] and boundary value issues [14] has been explored by several studies. Many current researchers, meanwhile, lack a methodical comparison of Newton-Raphson's effectiveness with conventional approaches in addressing first and second-order ODEs.

Lack of studies combining contemporary computing methods, including automatic differentiation and array-based optimizations in Newton-Raphson-based ODE solvers, is one of the main gaps in the field. Most research concentrates on traditional solutions without using recent developments in Python-based scientific computing; hence, improving accuracy and performance.

Though much study has been done on numerical techniques for ODEs, the Newton-Raphson approach is still underused for solving these equations. Although it is usually used for nonlinear algebraic equations, its potential for ODE solving has not been completely investigated. Differential equations encountered in modern engineering and physics problems are becoming more complex; thus, it is necessary to evaluate whether Newton-Raphson, together with automatic differentiation and optimal computational frameworks, can offer a reasonable substitute for conventional numerical solvers. The primary objectives of this research are:

- To formulate an approach for solving ordinary differential equations (ODEs) using the Newton-Raphson method.
- To implement this approach in Python using NumPy and autograd for efficient computation.
- To evaluate the accuracy and convergence of the Newton-Raphson method by comparing its results with analytical solutions.
- To apply the method to various case studies involving first- and second-order ODEs.
- To analyze potential limitations, such as error accumulation and computational efficiency, in order to assess the practical applicability of the method.

This study employs the Newton-Raphson method as a numerical solver for ordinary differential equations (ODEs), implemented in Python. The key methodological steps include:

- Reformulating ODEs as nonlinear algebraic equations suitable for Newton-Raphson iteration.
- Implementing automatic differentiation using autograd to compute Jacobians efficiently.
- Utilizing NumPy for optimized array operations to enhance computational performance.
- Validating the method's accuracy through comparison with analytical solutions.
- Conducting case studies to demonstrate the effectiveness of the approach in solving first- and second-order ODEs.

Emphasizing Python-based computational methods that provide benefits in automation, repeatability, and simplicity of integration with contemporary scientific computer systems included in this paper. Several important advances in the discipline of numerical techniques and scientific computing result from this work. Although the Newton-Raphson method is usually applied for solving algebraic equations, this work methodically investigates its application to ordinary differential equations (ODEs), hence extending its range in numerical analysis. Leveraging Python's autograd and NumPy libraries, the implementation highlights the benefits of automatic differentiation and effective matrix operations in ODE solution. By means of a comparative examination with conventional solvers, the benefits and shortcomings of the Newton-Raphson method are highlighted, therefore providing an understanding of possible applications. The work also uses the approach to practical issues, proving its viability and efficiency in managing several forms of ODEs. Moreover, the results expose difficulties including computational restrictions and error accumulation, which opens the path for future studies aiming at optimizing the Newton-Raphson approach for ODE solutions.

The work is organized to give a thorough grasp of the Newton-Raphson method's applicability for ODEs (ordinary differential equations). A theoretical foundation on the Newton-Raphson method with its derivation and iterative formulation is given in Section 2. Section 3 investigates the reformulation of ODEs as nonlinear systems, therefore enabling Newton-Raphson to solve them. Section 4 describes the Python implementation with particular reference to important computational methods and algorithmic optimizations applied for ODE modeling. Section 5 presents several case studies illustrating the accuracy and performance of the technique in ODE solvers. Section 6 examines the outcomes, contrasting the Newton-Raphson methodology with more traditional techniques and pointing out their respective advantages and drawbacks. Section 7 finally compiles the results, addresses their ramifications, and offers possible future routes for study in this field.

2. Newton-Raphson Method

The Newton-Raphson method is an iterative numerical approach for nonlinear function roots. The approach follows the update formula [15, 16] using a function f(y) where one searches a solution to f(y) = 0. [15, 16]:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} \tag{1}$$

where $f'(y_n)$ represents the derivative of $f(y)$ with respect to y. This method is known for its quadratic convergence when an initial estimate is close to the actual root.

3. Applying Newton-Raphson to Differential Equations

For solving ODEs, first the discretization of the problem has to be done using finite differences. Given an ODE:

$$\frac{dy}{dx} = g(x, y) \tag{2}$$

and a grid of discrete points x_i, the derivative can be approximated using Taylor series as:

$$\frac{y_{i+1} - y_i}{\Delta x} = g(x_i, y_i) \tag{3}$$

which can be rewritten as a residual function:

$$F(y) = \frac{y_{i+1} - y_i}{\Delta x} - g(x_i, y_i) = 0 \tag{4}$$

The Newton-Raphson iteration then solves this system by linearizing $F(y)$. The Jacobian matrix is computed as Pho [17]:

$$J_{ij} = \partial F_i / \partial y_j \tag{5}$$

and the update step involves solving the linear system as shown below:

$$J\Delta y = -F(y) \tag{6}$$

The solution is iteratively refined until convergence is achieved.

Algorithm

The Newton-Raphson method for ODEs follows these steps:

i. Discretize the domain: Divide the problem domain into discrete points.
ii. Provide an initial guess: A reasonable initial solution is chosen.
iii. Compute the residual function $F(y)$: Evaluate the difference between the discretized equation and the exact equation.
iv. Compute the Jacobian matrix J: Determine the derivative of $F(y)$ with respect to y.
v. Solve for Δy: Compute Δy by solving $J\Delta y = -F(y)$.
vi. Update the solution: Update y as $y_n = y_g + \Delta y$.
vii. Check convergence: If $\| \Delta y \|$ is below a tolerance threshold, the iteration stops.

4. Methodology Adopted in Python for Modelling the ODE Using Newton-Raphson

The implementation follows a structured numerical approach to solve an ordinary differential equation (ODE) using the Newton-Raphson method. Below is the methodology:

i. Importing Required Libraries
   - The program utilizes automatic differentiation (*autograd* [10]) for computing derivatives.
   - Numerical operations and array manipulations are handled using a scientific computing library i.e. *NumPy* [8].
   - A plotting library is used for visualizing the results i.e. *matplotlib.pylab* [18].

ii. Defining the Residual Function
   - The differential equation is discretized using finite differences. Backward difference is used if the ODE is first order, and central difference is used for second-order ODEs. This choice is up to the user. The formulas for forward difference and central difference formulations of first and second-order derivatives are as follows:

$$\frac{dy}{dx} = \frac{y_i - y_{i-1}}{\Delta x}, \frac{d^2y}{dx^2} = \frac{y_{i+1-}y_i + y_{i-1}}{\Delta x^2} \tag{5}$$

- The function computes the residual error between the numerical derivative and the given equation.
- A structured update approach ensures consistency across the domain.

iii. Computing the Jacobian Matrix
- The Jacobian matrix is automatically generated using a differentiation library i.e. using *jacobian()* function [19].
- It represents the sensitivity of the residual function with respect to the solution variables.
- This is essential for applying Newton's method efficiently.

iv. Setting Up the Computational Grid
- The solution domain is discretized into a set number of points.
- Initial values are chosen for the unknown function across these points.
- A copy of the initial guess is stored for iterative refinement.

v. Iterative Newton-Raphson Solver
- A loop executes a maximum number of iterations to refine the solution.
- At each step, the residual function is evaluated to measure the deviation from the expected solution.
- The Jacobian matrix is computed to solve for the correction term.
- The correction is applied to update the function values at interior points.
- Convergence is checked using a threshold on the update magnitude, terminating the process if the solution stabilizes.

vi. Visualization of Results
- The computed numerical solution is plotted against the domain.
- The exact analytical solution is also plotted for comparison.
- The graph provides insight into the accuracy of the numerical method.

This methodology ensures an efficient and structured approach to solving ordinary differential equations (ODEs) numerically using the Newton-Raphson method.

5. Case Studies and their solution in Python

The Newton-Raphson method has been applied to solve the following different types of ODEs:

1. Case-1 (First-Order ODE): The equation $\frac{dy}{dx} = \frac{1}{x}$, subject to $y(1) = 0$. This equation has an analytical solution of $y = \ln(x)$.
2. Case-2 (Second-Order ODE): $\frac{d^2y}{dx^2} = e^x$ subject $y(0) = -1$ and $y(5) = e^5$. This equation has an analytical solution of $y = e^x$.
3. Case- 3 (Second-Order ODE): $\frac{d^2y}{dx^2} - 4\frac{dy}{dx} + 3y = 0$ subject $y(0) = 1$ and $y(2) = -54.9$. The solution to the equation is $y = \frac{e^{x+2} - e^{3x}}{e^2 - 1}$.
4. Case-4 (First order ODE): The equation is $\frac{dy}{dx} - 2xy + y^2 = 5 - x^2$ subject to $y(0) = -2$ and $y(5) = 3$. The solution to the equation is $y = 2 - x$.

In all the cases, the primary variation will be of the function of equations (*Fn()*) and the terms one needs to take for the jacobian. If it's a second order, then only inner values of the Jacobian will be taken (in index form it's *J[:, 1:-1]*) as the domain will be solved for inner nodes. And for the first order ODE, all the columns except the first will be considered (*J[:,1:]*). Apart from this, for the second-order ODE, two boundary conditions will be used. Whereas for the first order, only the starting value will be given. Also, in making functions, no loop has been used; slicing of the array has been done. This has made the code more readable and efficient. Slicing utilizes NumPy's optimized internal routines, which are generally faster than explicit loops in Python. Apart from this, the native NumPy has not been used as $autograd$ (the module which is used to evaluate $Jacobian$) does not work with it. So, the $NumPy$ construct of the $atutograd$ has been used in this article.

```
from autograd import *
from autograd.numpy import *
from pylab import *

def Fn(x,y,n):
    Δx = (x[-1]-x[0])/(n-1)
    return (y[1:] - y[:-1]) / Δx - 1/(x[1:])

J = jacobian(lambda y: Fn(x, y, n))


n = 50
dom = (1,10)
x = linspace(dom[0],dom[1],n)
yg = linspace(0,-50,n)
```

```
yn = yg.copy()
N_max = 30

for _ in range(N_max):
    Fy = Fn(x,yg,n)

    Jy = J(yg)

    Δy = linalg.solve(Jy[:,1:],-Fy)
    yn[1:] = yg[1:] + Δy
    if linalg.norm(Δy)<0.0001:
        print(f"converged in {_+1} iterations")
        break
    else:
        yg = yn
plot(x,yn,label='Newton-Raphson')
yex = log(x)
plot(x,yex,'ko',label='Analytical')
xlabel('x')
ylabel('y')
legend()
```

Case 1.
The program for the first case will be as follows.

The program output is shown in Figure 1. The numerical solution obtained using the Newton-Raphson method closely follows the analytical solution, $y = \ln(x)$. The plot demonstrates a smooth logarithmic curve, confirming that the method effectively approximates the solution over the given domain. Minor deviations may appear due to discretization errors, but overall, the method converges efficiently. This case illustrates the robustness of Newton-Raphson when applied to simple first-order ODEs.

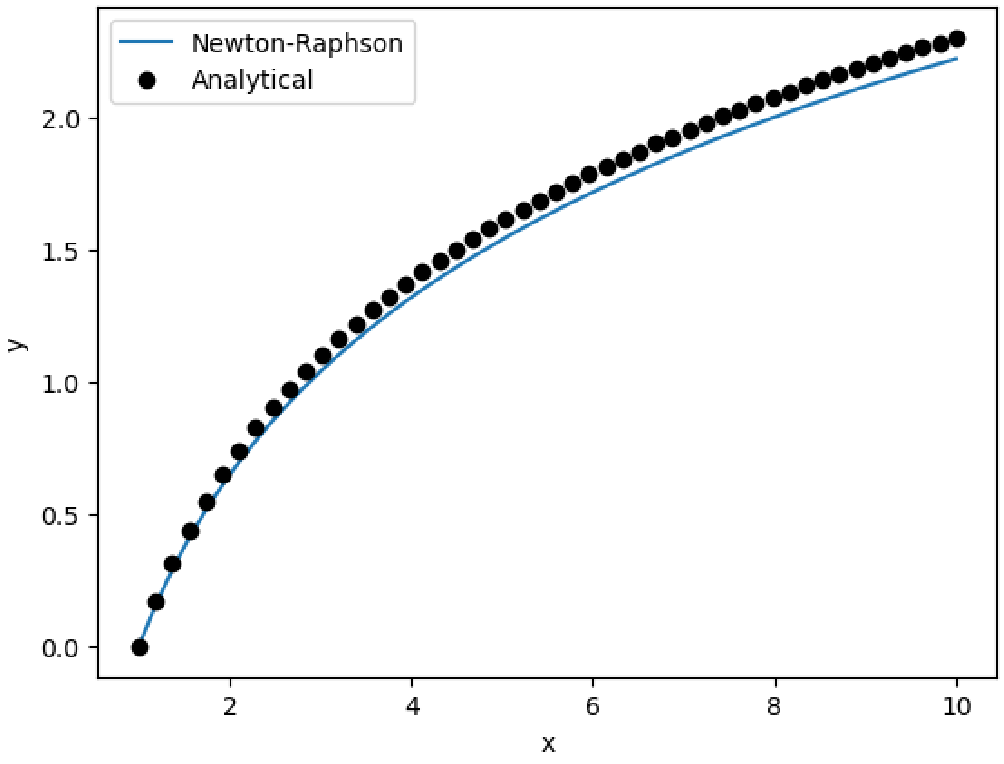

Figure 1.
Comparison of Newton-Raphson and Analytical Solution for Case-1.

```
from autograd import *
from autograd.numpy import *
from pylab import *

def Fn(x,y,n):
    Δx = (x[-1]-x[0])/(n-1)
    return (y[2:] - 2*y[1:-1]+y[:-2]) / Δx**2 - exp(x[1:-1])

J = jacobian(lambda y: Fn(x, y, n))


n = 50
dom = (0,5)
ran = (-1,exp(5))
x = linspace(dom[0],dom[1],n)
yg = linspace(ran[0],ran[1],n)
yn = yg.copy()
N_max = 30
for _ in range(N_max):
    Fy = Fn(x,yg,n)

    Jy = J(yg)

    Δy = linalg.solve(Jy[:,1:-1],-Fy)
    yn[1:-1] = yg[1:-1] + Δy
    if linalg.norm(Δy)<0.0001:
        print(f"converged in {_+1} iterations")
        break
    else:
        yg = yn
plot(x,yn)
plot(x,yn,label='Newton-Raphson')
yex = exp(x)
plot(x,yex,'ko',label='Analytical')
xlabel('x')
ylabel('y')
legend()
```

Case 2.
The program for the second case is as follows.

Figures 2 shows the program output. The projected exponential development behavior of the numerical results fits the analytical answer rather nicely. Reflecting the character of the exponential function, the figure shows a rising trend. The Newton-Raphson method proves useful for higher-order systems by effectively managing this second-order ODE. Step-size selection and numerical precision allow one to explain any slight variations between the numerical and analytical solutions.

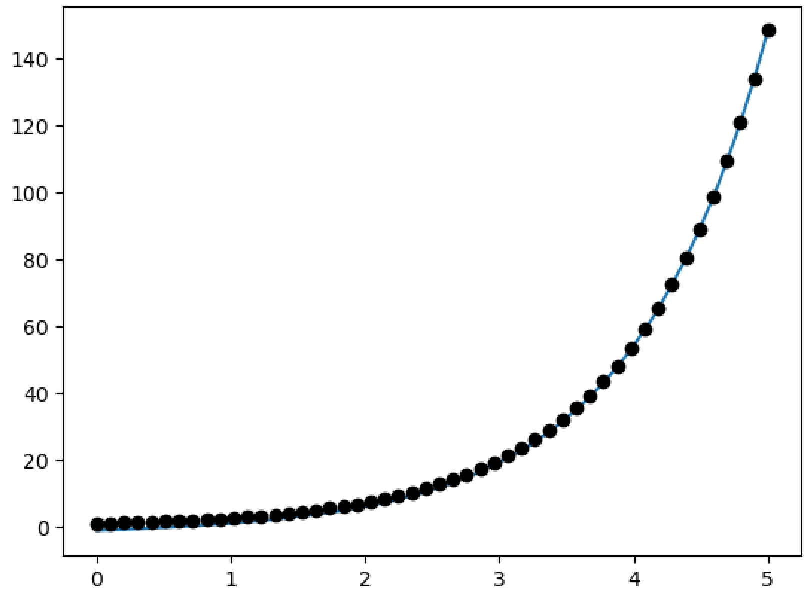

Figure 2.
Comparison of Newton-Raphson and Analytical Solution for Case-2.

```
from autograd import *
from autograd.numpy import *
from pylab import *

def Fn(x,y,n):
    Δx = (x[-1]-x[0])/(n-1)
    ypp = (y[2:] - 2*y[1:-1]+y[:-2]) / Δx**2
    yp = (y[2:]-y[:-2])/Δx
    return  ypp-4*yp+3*y[1:-1]

J = jacobian(lambda y: Fn(x, y, n))


n = 50
dom = (0,2)
ran = (1,-54)
x = linspace(dom[0],dom[1],n)
yg = linspace(ran[0],ran[1],n)
yn = yg.copy()
N_max = 30
for _ in range(N_max):
    Fy = Fn(x,yg,n)

    Jy = J(yg)

    Δy = linalg.solve(Jy[:,1:-1],-Fy)
    yn[1:-1] = yg[1:-1] + Δy
    if linalg.norm(Δy)<0.0001:
        print(f"converged in {_+1} iterations")
        break
    else:
        yg = yn

plot(x,yn,label='Newton-Raphson')
yex = (exp(x+2)-exp(3*x))/(exp(2)-1)
plot(x,yex,'ko',label='Analytical')
xlabel('x')
ylabel('y')
legend()
```

Case 3.
The program for the third case is as follows.

The program output is shown in Figure 3. Characteristics of differential equations with exponential components display an oscillatory decay. The successful capture of the behavior of the system by the Newton-Raphson approach yields a numerical approximation closely following the analytical answer. Nonetheless, the nature of the equation makes error accumulation obvious in some areas, especially in regions where the function varies quickly. For such second-order ODEs, the approach still works well, though.

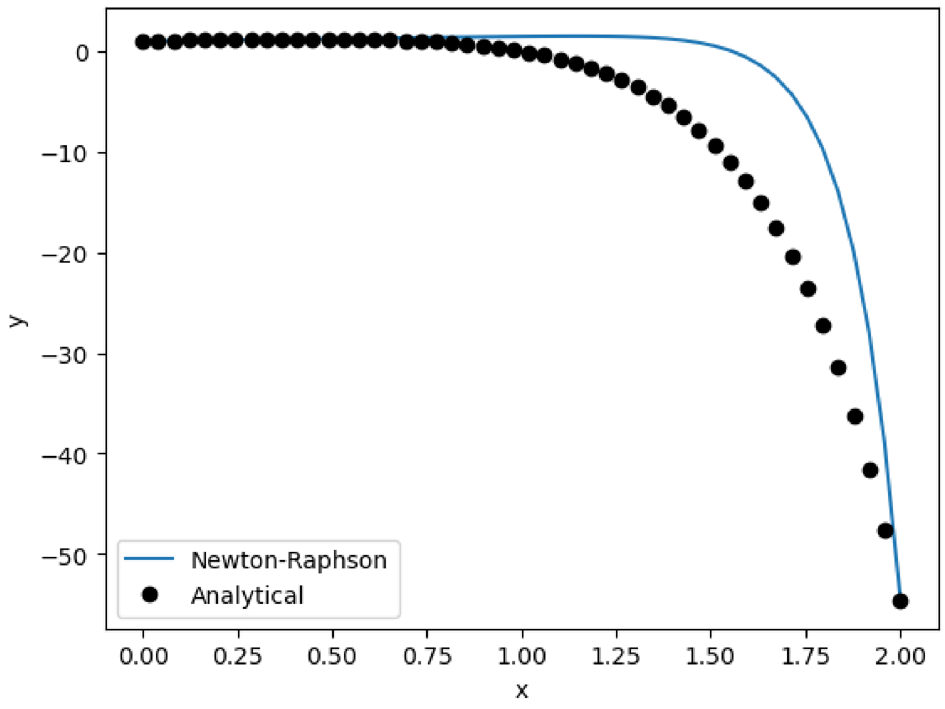

Figure 3.
Comparison of Newton-Raphson and Analytical Solution for Case-3.

```
from autograd import *
from autograd.numpy import *
from pylab import *

def Fn(x,y,n):
    Δx = (x[-1]-x[0])/(n-1)
    return (y[1:] - y[:-1]) / Δx - 2*x[1:]*y[1:]+y[1:]**2-5+x[1:]**2

J = jacobian(lambda y: Fn(x, y, n))


n = 50
dom = (0,5)
x = linspace(dom[0],dom[1],n)
yg = linspace(-2,6,n)
yn = yg.copy()
N_max = 30
for _ in range(N_max):
    Fy = Fn(x,yg,n)

    Jy = J(yg)

    Δy = linalg.solve(Jy[:,1:],-Fy)
    yn[1:] = yg[1:] + Δy
    if linalg.norm(Δy)<0.0001:
        print(f"converged in {_+1} iterations")
        break
    else:
        yg = yn
```

```
plot(x,yn,label='Newton-Raphson')
yex = x-2
plot(x,yex,'ko',label='Analytical')
xlabel('x')
ylabel('y')
legend()
```

Case 4.
The program for the fourth case is as follows.

The program output is shown in Figure 4. Consistent with the mathematical answer y=2-x, the plot for this situation verifies a linear trend. Rapid convergence of the Newton-Raphson method shows its effectiveness in solving nonlinear ODEs. With very little inaccuracy, the findings show a constant numerical solution over the specified range. This example emphasizes how precisely the approach can manage more difficult first-order equations.

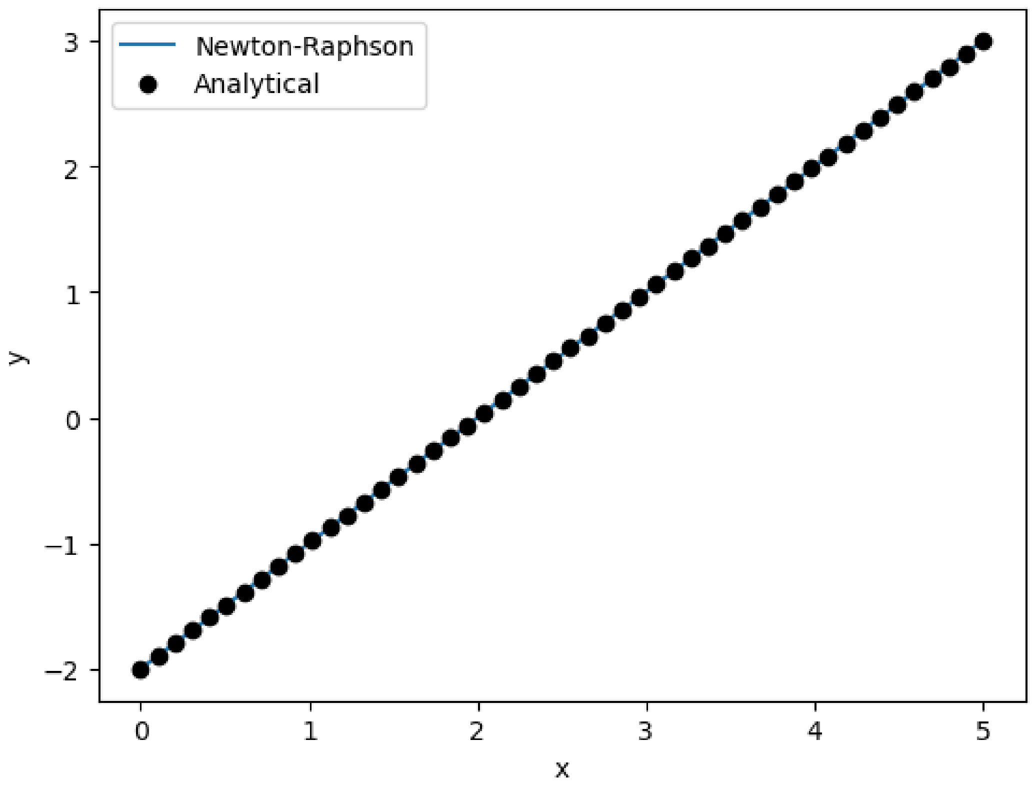

Figure 4.
Comparison of Newton-Raphson and Analytical Solution for Case-4.

6. Discussion

An interesting substitute for conventional numerical solvers is the Newton-Raphson method applied to solve ODEs. By means of this work, a methodical investigation has been conducted to explore, in first and second-order ODEs, the feasibility, accuracy, and computational efficiency of the Newton-Raphson method. Although the Newton-Raphson method can effectively approximate solutions, numerous important criteria affect its performance, including the formulation of the problem, initial estimate selection, and numerical stability. As the results show. One of the most significant outcomes of this study is that the Newton-Raphson method exhibits quick convergence in solving ODEs in cases when the first guess is well-selected. Unlike iterative methods such as Euler's or Runge-Kutta, which rely on stepwise propagation, the Newton-Raphson method immediately seeks the roots of the changed nonlinear system. Faster convergence in the context of well-conditioned systems results from this as well. However, reliance on a proper initial assumption can cause problems, particularly for stiff ODEs or complex boundary conditions. Strong starting conditions are therefore very important since small changes in the starting conditions can lead to divergence or convergence to erroneous solutions.

Combining Python's NumPy and autograd packages improves the Newton-Raphson method's computational efficiency. By enabling automatic differentiation, autograd helps to reduce the demand for computationally expensive and error-prone manual derivative computations. Meanwhile, effective handling of large-scale problems is made possible by the optimized matrix operations of NumPy. Newton-Raphson is a good option for some ODE problems, especially those requiring high accuracy and fast iteration, because of this computational benefit.

Comparative examination of Newton-Raphson with analytical solutions was fundamental for this work. First- and second-order ODE results showed that although Newton-Raphson approximates the exact solutions, variations were found in several circumstances. For instance, when the function shows faster changes or higher-order nonlinearities, error accumulation became more obvious. This suggests that, although being helpful, Newton-Raphson is not generally superior to conventional solvers. Its relevance must thus be assessed based on the specific type of issue. The work also underlines Newton-Raphson's probable constraints in managing ODEs with fast transitions or discontinuities. Conventional methods such as Runge-Kutta handle such circumstances more precisely because of their stepwise character and capacity to offer local error management at every iteration. Comparatively, the global Newton-Raphson approach can struggle in such conditions and cause numerical instability. Future research could look at hybrid approaches combining Newton-Raphson's efficiency with the stability of typical stepwise methods in order to raise overall performance.

Another significant result of the investigation is the function of numerical precision and error accumulation. Newton-Raphson is basically an iterative process; hence, rounding errors can compound over numerous rounds, particularly when high-order derivatives are involved. Dealing with second-order ODEs, where numerical differentiation could introduce additional mistakes, this problem becomes more obvious. Reducing error propagation while preserving computing efficiency requires either preconditioning or adaptive step-sizing techniques in handling this. Practically speaking, the research reveals that Newton-Raphson can be quite effectively applied to scientific and technical problems. The method supports further use as an additional numerical solver since it has demonstrated efficiency in solving numerous first and second-order ODEs. Its limitations, however, should be carefully considered regarding the issue structure before use. When paired with appropriate precautions against divergence and numerical instability, Newton-Raphson can be a helpful instrument for engineering applications where stability and precision are paramount.

By highlighting Newton-Raphson's strengths and shortcomings in this area, this work adds to the mounting corpus of studies on alternative numerical techniques for ODEs. Although it is not a general substitute for conventional solvers, its speed and efficiency in some situations make it an interesting choice for particular uses. More study should concentrate on improving the robustness of the technique, especially in managing stiff equations and ensuring stability in very nonlinear systems.

7. Conclusion

This paper studied the feasibility of numerically solving first and second-order ODEs using Python's autograd and NumPy modules. Under some circumstances, Newton-Raphson is shown to effectively and highly precisely estimate ODE solutions. Fast convergence is its key benefit, particularly in well-conditioned scenarios when a good initial guess is reachable. The work does, however, also draw attention to some restrictions including numerical stability problems, sensitivity to starting conditions, and error accumulation. Comparisons with analytical solutions reveal that in scenarios involving stiff or very nonlinear equations, conventional solvers like Runge-Kutta may still be better even if Newton-Raphson performs well in many circumstances. Emphasizing the need for suitable issue structure and initialization, the method presents advantages and challenges depending on reformulating ODEs as nonlinear systems.

All things considered, the results imply that the Newton-Raphson method can be a useful numerical instrument for ordinary differential equations (ODEs), especially in applications requiring fast convergence and computational efficiency. However, it should be used in conjunction with other numerical techniques to mitigate its limitations. Future work should explore hybrid approaches that integrate the Newton-Raphson method with traditional solvers, adaptive step-sizing techniques, and further optimizations to enhance stability and applicability to a broader range of differential equations.

References

[1] R. W. Easton, "Ordinary differential equations: An introduction to nonlinear analysis (Review of the book by Herbert Amann)," *SIAM Review*, vol. 33, no. 1, pp. 152–153, 1991. https://doi.org/10.1137/1033154

[2] K. Gajula, V. Sharma, D. Mishra, and P. Dumka, "First law of thermodynamics for closed system: A Python approach," *Research Applications in Thermal Engineering*, vol. 5, pp. 1-10, 2022.

[3] D. J. Higham, "Modeling and simulating chemical reactions," *SIAM Review*, vol. 50, no. 2, pp. 347-368, 2008. https://doi.org/10.1137/060666457

[4] W. Zhong and Z. Tian, "Solving initial value problem of ordinary differential equations by Monte Carlo method," presented at the 2011 International Conference on Multimedia Technology, 2011.

[5] J. Biazar, E. Babolian, and R. Islam, "Solution of the system of ordinary differential equations by Adomian decomposition method," *Applied Mathematics and Computation*, vol. 147, no. 3, pp. 713-719, 2004. https://doi.org/10.1016/S0096-3003(02)00806-8

[6] P. Mishra, A. Sharma, D. R. Mishra, and P. Dumka, "Solving double integration with the help of Monte Carlo simulation: A Python approach," *International Journal of Scientific Research in Multidisciplinary Studies*, vol. 9, no. 3, pp. 6–10, 2023.

[7] J. C. Strikwerda, *Finite difference schemes and partial differential equations*. SIAM. https://doi.org/10.2307/2008454SIAM, 2004.

[8] C. Bauckhage, *Numpy/scipy recipes for data science: Subset-constrained vector quantization via mean discrepancy minimization*. New York: Springer, 2020.

[9] K. Gajula, V. Sharma, B. Sharma, D. R. Mishra, and P. Dumka, "Modelling of energy in transit using Python," *International Journal of Innovative Science and Research Technology*, vol. 7, no. 8, pp. 1152-1156, 2022.

[10] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless gradients in numpy," presented at the ICML '15 AutoML Workshop, 2015.

[11] J. Butcher, "Practical Runge–Kutta methods for scientific computation," *The ANZIAM Journal*, vol. 50, no. 3, pp. 333-342, 2009. https://doi.org/10.1017/S1446181109000030

[12] G. Dahlquist, *Positive functions and some applications to stability questions for numerical methods* (Recent advances in numerical analysis). Academic Press. https://doi.org/10.1016/B978-0-12-208360-0.50006-1, 1978.

[13] E. L. Ionides, C. Bretó, and A. A. King, "Inference for nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 103, no. 49, pp. 18438-18443, 2006. https://doi.org/10.1073/pnas.0603181103

[14] U. M. Ascher, R. M. Mattheij, and R. D. Russell, *Numerical solution of boundary value problems for ordinary differential equations*. Philadelphia, PA: SIAM, 1995.

[15] C. Ghiaus, "Computational psychrometric analysis as a control problem: Case of cooling and dehumidification systems," *Journal of Building Performance Simulation*, vol. 15, no. 1, pp. 21-38, 2022. https://doi.org/10.1080/19401493.2021.1995498

[16] S. Akram and Q. ul Ann, "Newton Raphson method calculator," *International Journal of Science and Engineering Research*, vol. 6, pp. 1748–1752, 2015.

[17] K.-H. Pho, "Improvements of the Newton–Raphson method," *Journal of Computational and Applied Mathematics*, vol. 408, p. 114106, 2022. https://doi.org/10.1016/j.cam.2022.114106

[18] G. Kanagachidambaresan and G. Manohar Vinoothna, *Visualizations* (Programming with TensorFlow: Solution for Edge Computing Applications). Springer International Publishing. https://doi.org/10.1007/978-3-030-57077-4_3, 2021.

[19] J. Bradbury *et al.*, "Jax: Autograd and xla, Astrophysics Source Code Library," Retrieved: https://ui.adsabs.harvard.edu/abs/2021ascl.soft11002B, 2021.