Newton's Method

Tamas Kis | tamas.a.kis@outlook.com | https://tamaskis.github.io

CONTENTS

1 UNIVARIATE ROOTING FINDING

1.1 Basic Theory

Newton's method is a technique used to find the root (based on an initial guess[1] x_0) of a *differentiable*, univariate function $f(x)$. The equation of the tangent line to the curve $y = f(x)$ at $x = x_0$ is

$$y = f'(x_0)(x - x_0) + f(x_0)$$

where $f'(x_0)$ is the derivative of $f(x)$ evaluated at x_0. The x-intercept of this tangent line, $x = x_1$, can be solved by setting $y = 0$.

$$0 = f'(x_0)(x_1 - x_0) + f(x_0)$$

$$\therefore x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

x_1 is an updated estimate of the root of $f(x)$. To keep refining our estimate, we can keep iterating through this procedure using Eq. (1).

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}} \tag{1}$$

1.2 Implementation

So how do we actually use Eq. (1)? Given an initial guess x_0, we can keep coming up with new estimates of the root. But how do we know when to stop? To resolve this issue, we define the **error**[2] as

$$\boxed{\varepsilon = |x_{i+1} - x_i|} \tag{2}$$

Once ε is small enough, we say that the estimate of the root has **converged** to the true root, x^*, within some **tolerance** (which we denote as TOL). Therefore, if we predetermine that, at most, we can *tolerate* an error of TOL, then we will keep iterating Eq. (1) until $\varepsilon < \text{TOL}$. In some cases, the error may never decrease below TOL, or take too long to decrease to below TOL. Therefore, we also define the **maximum number of iterations** ($i_{\max}$) so that the algorithm does not keep iterating forever, or for too long of a time [1, 4].

There are two basic algorithms for implementing Newton's method. The first implementation, given as Algorithm 1 in Section 1.2.1, does *not* store the result of each iteration. On the other hand, the second implementation, given as Algorithm 2 in Section 1.2.2, *does* store the result of each iteration. `newtons_method` implements both of these algorithms.

Since Algorithm 2 first needs to preallocate a potentially huge array to store all of the intermediate solutions, Algorithm 1 is significantly faster. Even if $i_{\max}$ (determines size of the preallocated array) is set to be a small number (for example, 10), Algorithm 1 is still faster. The reason we still consider and implement Algorithm 2 is so that convergence studies may be performed.

[1] Often, a function $f(x)$ will have multiple roots. Therefore, Newton's method typically finds the root closest to the initial guess x_0. *However*, this is not always the case; the algorithm depends heavily on the derivative of $f(x)$, which, depending on its form, may cause it to converge on a root further from x_0.

[2] Note that ε is an *approximate* error. The motivation behind using this definition of ε is that as i gets large (i.e. $i \to \infty$), $x_{i+1} - x_i$ approaches $x_{i+1} - x^*$ (*assuming* this sequence is convergent), where x^* is the true root (and therefore $x_{i+1} - x^*$ represents the *exact* error).

1.2.1 "Fast" Implementation

Algorithm 1:
Newton's method ("fast" implementation).

Given:

- $f(x)$ - differentiable, univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $f'(x)$ - derivative of $f(x)$
- $x_0 \in \mathbb{R}$ - initial guess for root
- $\text{TOL} \in \mathbb{R}$ - tolerance
- $i_{\text{max}} \in \mathbb{Z}$ - maximum number of iterations

Procedure:

1. Manually set the root estimate at the first iteration based on the initial guess.

$$x_{\text{old}} = x_0$$

2. Initialize x_{new} so its scope will not be limited to within the while loop.

$$x_{\text{new}} = 0$$

3. Initialize the error so that the loop will be entered.

$$\varepsilon = (2)(\text{TOL})$$

4. Find the root using Newton's method.

$i = 1$

while $(\varepsilon > \text{TOL})$ **and** $(i < i_{\text{max}})$

(a) Update root estimate.

$$x_{\text{new}} = x_{\text{old}} - \frac{f(x_{\text{old}})}{f'(x_{\text{old}})}$$

(b) Calculate error.

$$\varepsilon = |x_{\text{new}} - x_{\text{old}}|$$

(c) Store the current root estimate for the next iteration.

$$x_{\text{old}} = x_{\text{new}}$$

(d) Increment loop index.

$$i = i + 1$$

end

Return:

- $x^* = x_{\text{new}} \in \mathbb{R}$ - converged root

1.2.2 "Return All" Implementation

Algorithm 2:
Newton's method ("return all" implementation).

Given:
- $f(x)$ - differentiable, univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $f'(x)$ - derivative of $f(x)$
- $x_0 \in \mathbb{R}$ - initial guess for root
- $\text{TOL} \in \mathbb{R}$ - tolerance
- $i_{\text{max}} \in \mathbb{R}$ - maximum number of iterations

Procedure:
1. Preallocate $\mathbf{x} \in \mathbb{R}^{i_{\text{max}}}$ to store the estimates of the root at each iteration.
2. Manually set the root estimate at the first iteration based on the initial guess (note that x_1 is the first element of $\mathbf{x}$, while x_0 is the input initial guess).

$$x_1 = x_0$$

3. Initialize the error so that the loop will be entered.

$$\varepsilon = (2)(\text{TOL})$$

4. Find the root using Newton's method.

$$i = 1$$

while $(\varepsilon > \text{TOL})$ **and** $(i < i_{\text{max}})$

(a) Update root estimate.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

(b) Calculate error.

$$\varepsilon = |x_{i+1} - x_i|$$

(c) Increment loop index.

$$i = i + 1$$

end

Return:
- $\mathbf{x} \in \mathbb{R}^n$ - vector where the first element is the initial guess for the root (x_0), the subsequent elements are the intermediate root estimates, and the final element is the converged root (x^*)

2 SOLVING A SYSTEM OF NONLINEAR EQUATIONS

2.1 Basic Theory

Consider a system of n nonlinear equations in n unknowns:

$$\begin{aligned} g_1(x_1, ..., x_n) &= h_1(x_1, ..., x_n) \\ g_2(x_1, ..., x_n) &= h_2(x_1, ..., x_n) \\ &\vdots \\ g_n(x_1, ..., x_n) &= h_n(x_1, ..., x_n) \end{aligned}$$

Let's rewrite the argument of each univariate function in terms of the vector variable $\mathbf{x} \in \mathbb{R}^n$, where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Additionally, let's move all the h equations to the left hand side. Then we have

$$\begin{aligned} g_1(\mathbf{x}) - h_1(\mathbf{x}) &= 0 \\ g_2(\mathbf{x}) - h_2(\mathbf{x}) &= 0 \\ &\vdots \\ g_n(\mathbf{x}) - h_n(\mathbf{x}) &= 0 \end{aligned}$$

Let's define $f_i(\mathbf{x}) = g_1(\mathbf{x}) - h_1(\mathbf{x})$. Then

$$\begin{aligned} f_1(\mathbf{x}) &= 0 \\ f_2(\mathbf{x}) &= 0 \\ &\vdots \\ f_n(\mathbf{x}) &= 0 \end{aligned}$$

Defining $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ as a vector-valued function,

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix}$$

We have thus converted this problem into solving

$$\boxed{\mathbf{f}(\mathbf{x}) = \mathbf{0}} \tag{3}$$

In Section 1, we introduced Newton's method as an algorithm for finding the root of a univariate function $f(x)$. Finding the root of $f(x)$ is, by definition, solving the equation

$$f(x) = 0$$

for x. Note the similarity of this equation to Eq. (3). We can extend Newton's method to the case of a multivariate, vector-valued function whose input and output dimensions are the same (i.e. same number of equations and unknowns). For the univariate case, we used the update equation

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

In the multivariate, vector-valued case, this becomes

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}(\mathbf{x}_i)^{-1}\mathbf{f}(\mathbf{x}_i)$$

However, in its implementation, we avoid computing the inverse of the Jacobian matrix. Instead, we solve the rearranged equation

$$\mathbf{J}(\mathbf{x}_i)(\mathbf{x}_{i+1} - \mathbf{x}_i) = -\mathbf{f}(\mathbf{x}_i)$$

for the unknown $\mathbf{x}_{i+1} - \mathbf{x}_i$, and then find $\mathbf{x}_{i+1}$ accordingly. In two steps, this can be written as

$$\boxed{\begin{aligned}&\mathbf{J}(\mathbf{x}_i)\mathbf{y}_i = -\mathbf{f}(\mathbf{x}_i)\\&\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{y}_i\end{aligned}} \tag{4}$$

2.2 Approximating the Jacobian

To approximate the Jacobian, $\mathbf{J}(\mathbf{x}_i)$, we can use the `ijacobian` function from the *Numerical Differentiation Toolbox* [2], which provides a numerical approximation typically accurate to within double precision.

$$\mathbf{J}(\mathbf{x}_i) \approx \texttt{ijacobian}(\mathbf{f}, \mathbf{x}_i)$$

However, there are a few functions that special care must be taken with. Notably, the "complexified" versions of the absolute value, four-quadrant inverse tangent, and 2-norm functions should be used:

$$\begin{aligned}\texttt{abs} &\rightarrow \texttt{iabs}\\ \texttt{atan2} &\rightarrow \texttt{iatan2}\\ \texttt{atan2d} &\rightarrow \texttt{iatan2d}\\ \texttt{norm} &\rightarrow \texttt{inorm}\end{aligned}$$

Additionally, the MATLAB implementations of following functions do *not* currently work with the `ijacobian` function [3]:

- $\text{arccsc}\,(x)$ for $x < -1$
- $\text{arcsec}\,(x)$ for $x < -1$
- $\text{arccoth}\,(x)$ for $0 < x < 1$
- $\text{arctanh}\,(x)$ for $x > 1$
- $\text{arcsech}\,(x)$ for $-1 < x < 0$
- $\text{arccoth}\,(x)$ for $-1 < x < 0$
- $\text{arccosh}\,(x)$ for $x < -1$
- $\text{arctanh}\,(x)$ for $x < -1$

2.3 Implementation

Like in the univariate case, there is a "fast" implementation of Newton's method and a "return all" implementation of Newton's method. The former is described in Section 2.3.1 while the latter is described in Section 2.3.2.

2.3.1 "Fast" Implementation

Algorithm 3:
Newton's method ("fast" implementation).

Given:

- $\mathbf{f}(\mathbf{x})$ - multivariate, vector-valued function ($\mathbf{f}: \mathbb{R}^n \to \mathbb{R}^n$)
- $\mathbf{J}(\mathbf{x})$ - *(OPTIONAL)* Jacobian of $\mathbf{f}(\mathbf{x})$
- $\mathbf{x}_0 \in \mathbb{R}^n$ - initial guess for solution
- $\mathrm{TOL} \in \mathbb{R}$ - tolerance
- $i_{\max} \in \mathbb{Z}$ - maximum number of iterations

Procedure:

1. Define the Jacobian using the `ijacobian` function if it is not input.

    if $\mathbf{J}(\mathbf{x})$ not specified

    $\quad \mathbf{J}(\mathbf{x}) \approx \texttt{ijacobian}(\mathbf{f}, \mathbf{x})$

    end

2. Manually set the solution estimate at the first iteration based on the initial guess.

    $\mathbf{x}_{\text{old}} = \mathbf{x}_0$

3. Initialize $\mathbf{x}_{\text{new}}$ so its scope will not be limited to within the while loop.

    $\mathbf{x}_{\text{new}} = \mathbf{0}$

4. Initialize the error so that the loop will be entered.

    $\varepsilon = (2)(\mathrm{TOL})$

5. Find the solution using Newton's method.

    $i = 1$

    while $(\varepsilon > \mathrm{TOL})$ **and** $(i < i_{\max})$

    (a) Solve the linear system below for $\mathbf{y}$.

    $$\mathbf{J}(\mathbf{x}_{\text{old}})\mathbf{y} = -\mathbf{f}(\mathbf{x}_{\text{old}})$$

    (b) Update solution estimate.

    $$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \mathbf{y}$$

    (c) Calculate error.

    $$\varepsilon = \|\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}}\|$$

    (d) Store the current solution estimate for the next iteration.

    $$\mathbf{x}_{\text{old}} = \mathbf{x}_{\text{new}}$$

    (e) Increment loop index.

    $$i = i + 1$$

    end

Return:

- $\mathbf{x}^* = \mathbf{x}_{\text{new}} \in \mathbb{R}^n$ - converged solution

2.3.2 "Return All" Implementation

Algorithm 4:
Newton's method ("return all" implementation).

Given:
- $\mathbf{f}(\mathbf{x})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$)
- $\mathbf{J}(\mathbf{x})$ - *(OPTIONAL)* Jacobian of $\mathbf{f}(\mathbf{x})$
- $\mathbf{x}_0 \in \mathbb{R}^n$ - initial guess for solution
- $\text{TOL} \in \mathbb{R}$ - tolerance
- $i_{\text{max}} \in \mathbb{Z}$ - maximum number of iterations

Procedure:
1. Define the Jacobian using the `ijacobian` function if it is not input.

   if $\mathbf{J}(\mathbf{x})$ not specified

   $\quad \mathbf{J}(\mathbf{x}) \approx \texttt{ijacobian}(\mathbf{f}, \mathbf{x})$

   end

2. Preallocate $\mathbf{x} \in \mathbb{R}^{n \times i_{\text{max}}}$ to store the estimates of the solution at each iteration.
3. Manually set the solution estimate at the first iteration based on the initial guess (note that $\mathbf{x}_1$ is the first column of $\mathbf{x}$, while $\mathbf{x}_0$ is the input initial guess).

   $$\mathbf{x}_1 = \mathbf{x}_0$$

4. Initialize the error so that the loop will be entered.

   $$\varepsilon = (2)(\text{TOL})$$

5. Find the solution using Newton's method.

   $i = 1$

   while $(\varepsilon > \text{TOL})$ **and** $(i < i_{\text{max}})$

   (a) Solve the linear system below for $\mathbf{y}$.

   $$\mathbf{J}(\mathbf{x}_i)\mathbf{y} = -\mathbf{f}(\mathbf{x}_i)$$

   (b) Update solution estimate.

   $$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{y}$$

   (c) Calculate error.

   $$\varepsilon = \|\mathbf{x}_{i+1} - \mathbf{x}_i\|$$

   (d) Increment loop index.

   $$i = i + 1$$

   end

Return:
- $\mathbf{x} \in \mathbb{R}^{n \times i}$ - matrix where the first column is the initial guess for the solution ($\mathbf{x}_0$), the subsequent columns are the intermediate solution estimates, and the final column is the converged solution ($\mathbf{x}^*$)

Note:

- i is the number of iterations it took for the solution to converge.

REFERENCES

[1] Richard L. Burden and J. Douglas Faires. “Newton’s Method and Its Extensions”. In: *Numerical Analysis*. 9th ed. Boston, MA: Brooks/Cole, Cengage Learning, 2011. Chap. 2.3, pp. 67–78.

[2] Tamas Kis. *Numerical Differentiation Toolbox*. 2021. URL: https://github.com/tamaskis/Numerical_Differentiation_Toolbox-MATLAB.

[3] Tamas Kis. *Numerical Differentiation using the Complex-Step Approximation*. 2021. URL: https://tamaskis.github.io/documentation/Numerical_Differentiation_using_the_Complex_Step_Approximation.pdf.

[4] *Newton’s method*. Wikipedia. Accessed: June 10, 2020. URL: https://en.wikipedia.org/wiki/Newton%27s_method.