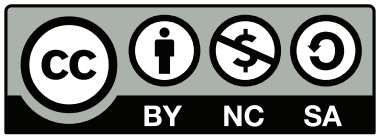
Root-Finding Methods

Tamas Kis | tamas.a.kis@outlook.com

Tamas Kis
https://tamaskis.github.io

Copyright © 2022 Tamas Kis.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ *or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.*



Contents

Contents iii

List of Algorithms iv

1 Fixed-Point Iteration 1
- 1.1 Basic Theory . . . 1
- 1.2 Termination Conditions . . . 1
- 1.3 Algorithm . . . 2
- 1.4 Iterative Approaches in Engineering . . . 3
- 1.5 Root Finding Using Fixed-Point Iteration . . . 4

2 Univariate Root Finding 5
- 2.1 Newton's Method . . . 5
- 2.2 Secant Method . . . 7
- 2.3 Bisection Method . . . 8

3 Multivariate Root Finding 11
- 3.1 Newton's Method in n Dimensions . . . 11
  - 3.1.1 Approximating the Jacobian . . . 12
- 3.2 Algorithm . . . 12
- 3.3 Solving Nonlinear Systems . . . 13

References 14

List of Algorithms

Algorithm 1	`fixed_point_iteration`	Fixed-point iteration for finding the fixed point of a univariate, scalar-valued function	2
Algorithm 2	`newtons_method`	Newton's method for finding the root of a differentiable, univariate, scalar-valued function	6
Algorithm 3	`secant_method`	Secant method for finding the root of a univariate, scalar-valued function	7
Algorithm 4	`bisection_method`	Bisection method for finding the root of a univariate, scalar-valued function	9
Algorithm 5	`newtons_method_n`	Newton's method for finding the root of a differentiable, multivariate, vector-valued function	12

1 Fixed-Point Iteration

1.1 Basic Theory

Consider a univariate function $f(x)$. A **fixed point** of $f(x)$, which we denote as c, satisfies

$$f(c) = c$$

Essentially, at the fixed point, the value of the dependent variable is the same as the value of the independent variable. Another way to view fixed points are as the intersections of the curve $y = f(x)$ with the curve $y = x$.

Consider the case where we can't explicitly solve for c. Let's assume we make an initial guess, c_0, and then iteratively generate a sequence of refined estimates for the true value of c.

$$\{c_0, c_1, c_2, ..., c_k, ...\}$$

If this sequence of refined estimates for the fixed point converges to the true fixed point, c, then

$$c = \lim_{k \to \infty} c_k \tag{1.1}$$

The sequence for the refined fixed point estimates (the c_k's) can be iteratively generated using the function, $f(x)$, whose fixed point we are trying to find.

$$\boxed{c_{k+1} = f(c_k)} \tag{1.2}$$

To show that this is valid, from Eqs. (1.1) and (1.2), we have [1, p. 60]

$$c = \lim_{k \to \infty} c_k = \lim_{k \to \infty} f(c_{k-1}) = f\left(\lim_{k \to \infty} c_{k-1}\right) = f(c)$$

1.2 Termination Conditions

Given an initial guess, c_0, we can keep coming up with new estimates of the fixed point. To terminate the iterative procedure, it is easiest to use the **absolute error**, ε, defined as

$$\boxed{\varepsilon = |x_{k+1} - x_k|} \tag{1.3}$$

Once ε is small enough, we say that the estimate of the fixed point has **converged** to the true fixed point, c, within some **tolerance** (which we denote as TOL). For all the algorithms discussed in this document, we assume a tolerance of

$$\text{TOL} = 10^{-10}$$

unless otherwise specified. Note that a relative error, ε_r, is also often used in iterative algorithms such as fixed point iteration.

$$\varepsilon_r = \frac{|x_{k+1} - x_k|}{|x_k|}$$

ε_r does a better job in comparing iterates with respect to each other, since depending on the problem, we could consider a difference of $x_{k+1} - x_k = 100$ to be really small, while for other problems, that difference is very large. However, using ε_r can lead to issues when $|x_k|$ is 0 (or numerically very small) since it appears in the denominator. Additionally, when manually setting the tolerance, it is more intuitive to set an absolute tolerance rather than a relative tolerance; an absolute tolerance gives a direct measure of the numerical precision of the result, while a relative tolerance provides an estimate of the number of correct digits.

If we predetermine that, at most, we can *tolerate* an error of TOL, then we will keep iterating Eq. (1.2) until $\varepsilon < \text{TOL}$. In some cases, the error may never decrease below TOL, or take too long to decrease below TOL. To account for this, we also define the **maximum number of iterations**, $k_{\max}$, so that the algorithm does not keep iterating forever, or for too long of a time [1, pp. 49–50]. For all the algorithms discussed in this document, we assume a maximum number of iterations of

$$k_{\max} = 200$$

unless otherwise specified.

1.3 Algorithm

Algorithm 1 is adapted from [1, pp. 60–61].

Algorithm 1: `fixed_point_iteration`

Fixed-point iteration for finding the fixed point of a univariate, scalar-valued function.

Given:

- $f(x)$ - univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $x_0 \in \mathbb{R}$ - initial guess for fixed point
- $\text{TOL} \in \mathbb{R}$ - *(OPTIONAL)* tolerance
- $k_{\max} \in \mathbb{Z}$ - *(OPTIONAL)* maximum number of iterations

Procedure:

1. Default the tolerance to $\text{TOL} = 10^{-12}$ if not specified.
2. Default the maximum number of iterations to $k_{\max} = 200$ if not specified.
3. Fixed point estimate at the first iteration.

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

4. Initialize x_{next}.

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

5. Find the fixed point using fixed-point iteration.

   for $k = 1$ **to** $k_{\max}$

(a) Update the fixed point estimate.

$$x_{\text{next}} = f(x_{\text{curr}})$$

(b) Terminate if converged.

if $|x_{\text{next}} - x_{\text{curr}}| < \text{TOL}$
  break
end

(c) Store the updated fixed point estimate for the next iteration.

$$x_{\text{curr}} = x_{\text{next}}$$

end

6. Converged fixed point.

$$c = x_{\text{next}}$$

Return:

- $c \in \mathbb{R}$ - fixed point of $f(x)$

1.4 Iterative Approaches in Engineering

Fixed-point iteration is especially useful for solving many problems in engineering. Consider the case where we have two unknown quantities, and two highly nonlinear equations that relate them to one another. Mathematically, we have two variables, x and y, and the following two functions:

$$y = f(x) \tag{1.4}$$

$$x = g(y) \tag{1.5}$$

In one function, you input x and get y, while in the other function, you input y and get x. Since $f(x)$ and $g(y)$ are nonlinear (as previously mentioned), we cannot obtain closed-form solutions for x and y.

Let's say we're primarily interested in the variable x, where the variable y mainly serves to place a constraint on x. We want to find the value $x = c$ such that both equations are satisfied simultaneously (i.e. $y = f(c)$ and $c = g(y)$). Then we can define a new function $h(x)$ as the function composition $h = g \circ f$ by substituting Eq. (1.4) into Eq. (1.5).

$$h(x) = g(f(x))$$

The solution to our problem, $x = c$, is just the fixed point of $h(x)$. See Examples #4a and #4b on the "Examples" tab of https://www.mathworks.com/matlabcentral/fileexchange/86992-fixed-point-iteration-fixed_point_iteration for an implementation of this approach to a pipe flow problem[1].

[1] This example is adapted from my personal solutions to Problem 8.96 in [4, p. 476] using the Haaland equation [4, p. 431]. However, this fluid mechanics text, in general, does not take a computational approach to such problems. Rather, it performs a "trial and error" procedure (including hand calculations and reading values off of a chart), which essentially follows the same process as fixed-point iteration – an example of this can be found in Example 8.7 [4, p. 444].

1.5 Root Finding Using Fixed-Point Iteration

Consider a function, $f(x)$. To find a root of $f(x)$, we need to find a value of x that satisfies

$$f(x) = 0$$

Similarly, to find a fixed-point of $f(x)$, we know we need to find a value of x that satisfies

$$f(x) = x$$

Therefore, it is simple to convert a root-finding problem to a fixed-point-finding problem.

> Finding the root of $f(x)$ is equivalent to finding the fixed-point of $g(x)$, where $g(x)$ is defined as [1, p. 56]
>
> $$g(x) = x - f(x)$$

2

Univariate Root Finding

2.1 Newton's Method

Newton's method is a technique used to find the root (based on an **initial guess**[1], x_0) of a *differentiable*, univariate function $f(x)$. Consider the Taylor series expansion of $f(x)$ about the point $x = x_0$.

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots$$

Ignoring higher order terms, we have

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) \tag{2.1}$$

Equation (2.1) essentially approximates $f(x)$ as the tangent line to the curve $y = f(x)$ at the point $x = x_0$. The x-intercept of this tangent line (i.e. its **root** or **zero**, x_1, can be found by setting the left hand side of Eq. (2.1) equal to 0.

$$0 = f'(x_0)(x_1 - x_0) + f(x_0) \quad \rightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

x_1 represents an *updated* estimate of the root of $f(x)$, given an initial guess x_0. To keep refining our estimate, we can keep iterating through this procedure. Eq. (2.2) below finds the $(k+1)$th iterate given the kth iterate.

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}} \tag{2.2}$$

The iteration can be termination in the same manner as described in Section 1.2 [1, pp. 67–68].

[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.

Algorithm 2: `newtons_method`

Newton's method for finding the root of a differentiable, univariate, scalar-valued function.

Given:

- $f(x)$ - differentiable, univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $f'(x)$ - derivative of $f(x)$ ($f' : \mathbb{R} \to \mathbb{R}$)
- $x_0 \in \mathbb{R}$ - initial guess for root
- $\text{TOL} \in \mathbb{R}$ - *(OPTIONAL)* tolerance (defaults to 10^{-10})
- $k_{\max} \in \mathbb{Z}$ - *(OPTIONAL)* maximum number of iterations (defaults to 200)

Procedure:

1. Default the tolerance to $\text{TOL} = 10^{-10}$ if not input.
2. Default the maximum number of iterations to $k_{\max} = 200$ if not input.
3. Root estimate at the first iteration.

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

4. Initialize x_{next}.

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

5. Find the root using Newton's method.

for $k = 1$ **to** $k_{\max}$

(a) Update the root estimate.

$$x_{\text{next}} = x_{\text{curr}} - \frac{f(x_{\text{curr}})}{f'(x_{\text{curr}})}$$

(b) Terminate if converged.

if $|x_{\text{next}} - x_{\text{curr}}| < \text{TOL}$
break
end

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

$$x_{\text{curr}} = x_{\text{next}}$$

end

6. Converged root.

$$x^* = x_{\text{next}}$$

Return:

- $x^* \in \mathbb{R}$ - root of $f(x)$

2.2 Secant Method

Recall Newton's method for iteratively solving for the root of a differentiable function:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2.3}$$

If we don't know $f'(x)$ (for example, it could be very complicated and tedious to derive by hand), then we can instead approximate it using some numerical method. Specifically, for the secant method, we use the backward approximation of a derivative, given by Eq. (2.4) below.

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \tag{2.4}$$

Substituting Eq. (2.4) into Eq. (2.3),

$$\begin{aligned} x_{k+1} &= x_k - \left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right] f(x_k) = \frac{[f(x_k) - f(x_{k-1})]\, x_k}{f(x_k) - f(x_{k-1})} - \frac{(x_k - x_{k-1}) f(x_k)}{f(x_k) - f(x_{k-1})} \\ &= \frac{x_k f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})} + \frac{x_{k-1} f(x_k) - x_k f(x_k)}{f(x_k) - f(x_{k-1})} = \frac{x_k f(x_k) - x_k f(x_k) + x_{k-1} f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})} \end{aligned}$$

$$\boxed{x_{k+1} = \frac{x_{k-1} f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}} \tag{2.5}$$

Equation (2.5) iteratively defines the **secant method**, which can be essentially thought of as a finite difference approximation of Newton's method for finding the root of a univariate function. The implementation of the secant method is very similar to that of Newton's method, but there are some important items of note [1, pp. 71–72]:

- Like with Newton's method, we first have to make an initial guess x_0 for the root. Additionally, we need to set the root estimate at the first iteration (i.e. x_1) to a value slightly different than x_0 – otherwise, the denominator of the right hand side of Eq. (2.5) will be 0 (since $f(x_1) - f(x_0) = f(x_0) - f(x_0) = 0$ if $x_1 = x_0$) and x_2 will be undefined. We can think of this as "kick-starting" the algorithm.
- Function evaluations (i.e. evaluating $f(x)$) are typically the most expensive part of the solution procedure. Note that for the secant method, $f(x_{\text{curr}})$ at the kth iteration is identical to $f(x_{\text{prev}})$ at the $(k+1)$th iteration. Therefore, it is prudent to save function evaluations for the subsequent iteration.

Algorithm 3: `secant_method`

Secant method for finding the root of a univariate, scalar-valued function.

Given:

- $f(x)$ - univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $x_0 \in \mathbb{R}$ - initial guess for root
- $\text{TOL} \in \mathbb{R}$ - *(OPTIONAL)* tolerance (defaults to 10^{-10})
- $k_{\max} \in \mathbb{Z}$ - *(OPTIONAL)* maximum number of iterations (defaults to 200)

Procedure:

1. Default the tolerance to $\text{TOL} = 10^{-10}$ if not input.
2. Default the maximum number of iterations to $k_{\max} = 200$ if not input.
3. Root estimates at the first and second iterations.

$$\begin{aligned} x_{\text{prev}} &= x_0 \\ x_{\text{curr}} &= x_0 + 0.001 \end{aligned}$$

4. Function evaluation at the first iteration.

$$f_{\text{prev}} = f(x_0)$$

5. Initialize x_{next}.

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

6. Find the root using the secant method.

for $k = 2$ **to** $k_{\max}$

(a) Function evaluation at the current iteration.

$$f_{\text{curr}} = f(x_{\text{curr}})$$

(b) Update the root estimate.

$$x_{\text{next}} = \frac{x_{\text{prev}} f_{\text{curr}} - x_{\text{curr}} f_{\text{prev}}}{f_{\text{curr}} - f_{\text{prev}}}$$

(c) Terminate if converged.

if $|x_{\text{next}} - x_{\text{curr}}| < \text{TOL}$
  break
end

(d) Store the next and current root estimates for the next iteration.

$$x_{\text{prev}} = x_{\text{curr}}$$
$$x_{\text{curr}} = x_{\text{next}}$$

(e) Store the current function evaluation for the next iteration.

$$f_{\text{prev}} = f_{\text{curr}}$$

end

7. Converged root.

$$x^* = x_{\text{next}}$$

Return:

- $x^* \in \mathbb{R}$ - root of $f(x)$

2.3 Bisection Method

The **bisection method** can be used to find the root of a univariate function $f(x)$, with no restrictions on the differentiability of f. The basic idea behind the bisection is starting off with some interval $[a, b]$ containing a root, iteratively "shrinking" this interval until it is below some tolerance threshold, and then taking the root to be the midpoint of this interval. The general procedure is as follows:

1. Make an initial guess for the interval $[a, b]$ containing the root.

2. Assume the root, c, is the midpoint of this interval: $c = (a + b)/2$.
3. Evaluate $f(a)$ and $f(c)$.
   (a) If $f(a) < 0$ and $f(c) > 0$ (i.e. they have different sign), we know the true root, x^*, is contained in the interval $[a, c]$. Therefore, we update our interval so that a remains the same, but b is updated to be c.
   (b) If $f(a)$ and $f(c)$ have the same sign (either both negative or both positive), we know the true root, x^*, must be contained in the interval $[c, b]$. Therefore, we update our interval so that b remains the same, but a is updated to be c.
4. Repeating steps 2 and 3, the interval $[a, b]$ will keep shrinking. Once the difference $(b - a)$ is small enough, we say that the estimate of the root has converged to the true root, x^*, within some tolerance, TOL.

In some cases, the difference $(b - a)$ may never decrease below TOL, or take too long to decrease to below TOL. Therefore, like with the previous methods, we also use the maximum number of iterations, $k_{\max}$, as another termination condition [1, pp. 48–49].

Algorithm 4:
Bisection method for finding the root of a univariate, scalar-valued function.

Given:

- $f(x)$ - univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
- $a \in \mathbb{R}$ - lower bound of interval containing root
- $b \in \mathbb{R}$ - upper bound of interval containing root
- $\text{TOL} \in \mathbb{R}$ - *(OPTIONAL)* tolerance (defaults to 10^{-10})
- $k_{\max} \in \mathbb{Z}$ - *(OPTIONAL)* maximum number of iterations (defaults to 200)

Procedure:

1. Default the tolerance to $\text{TOL} = 10^{-10}$ if not input.
2. Default the maximum number of iterations to $k_{\max} = 200$ if not input.
3. Root estimate at the first iteration.

$$c = \frac{a + b}{2}$$

4. Function evaluations at the first iteration.

$$f_a = f(a)$$
$$f_c = f(c)$$

5. Find the root using the bisection method.

for $k = 1$ **to** $k_{\max}$

(a) Update interval.

if $f_c = 0$
  break
else if $f_a f_c > 0$
  $a = c$
  $f_a = f_c$
else
  $b = c$
end

(b) Update the root estimate.

$$c = \frac{a+b}{2}$$

(c) Terminate if converged.

if $(b - a) < \text{TOL}$
  break
end

(d) Function evaluation at updated root estimate.

$$f_c = f(c)$$

end

6. Converged root.

$$x^* = c$$

Return:

- $x^* \in \mathbb{R}$ - root of $f(x)$

3 Multivariate Root Finding

3.1 Newton's Method in n Dimensions

Consider a multivariate, vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Our goal is to find the root, $\mathbf{x}^*$, of this function.

$$\boxed{\mathbf{f}(\mathbf{x}) = \mathbf{0}} \tag{3.1}$$

In Section 2.1, we introduced Newton's method as an algorithm for finding the root of a scalar-valued, univariate function, $f : \mathbb{R} \to \mathbb{R}$. 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. (3.1). 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

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

By analogy, in the multivariate, vector-valued case, this becomes

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

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

$$\mathbf{J}(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{f}(\mathbf{x}_k)$$

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

$$\boxed{\begin{aligned} &\underbrace{\mathbf{J}(\mathbf{x}_k)\mathbf{y}_k = -\mathbf{f}(\mathbf{x}_k)}_{\text{solve for } \mathbf{y}_k} \\ &\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{y}_k \end{aligned}} \tag{3.2}$$

For the multivariate case, we define the absolute error for the termination condition using the 2-norm:

$$\varepsilon = \|\mathbf{x}_{k+1} - \mathbf{x}_k\|$$

However, from Eq. (3.2), we also know that $\mathbf{y}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$. Thus, we can rewrite the absolute error as [1, pp. 638-641]

$$\boxed{\varepsilon = \mathbf{y}_k} \tag{3.3}$$

3.1.1 Approximating the Jacobian

Note that Newton's method for multivariate, vector-valued functions requires the Jacobian, $\mathbf{J}(\mathbf{x}_k)$. In many cases, it is difficult/time-consuming to calculate this Jacobian. Instead, we can use the `ijacobian` (complex-step approximation), `cjacobian` (central difference approximation), or `fjacobian` (forward difference approximation) functions from the *Numerical Differentiation Toolbox* [2].

$$\mathbf{J}(\mathbf{x}_k) \approx \texttt{ijacobian}(\mathbf{f}, \mathbf{x}_k)$$
$$\mathbf{J}(\mathbf{x}_k) \approx \texttt{cjacobian}(\mathbf{f}, \mathbf{x}_k)$$
$$\mathbf{J}(\mathbf{x}_k) \approx \texttt{fjacobian}(\mathbf{f}, \mathbf{x}_k)$$

The `ijacobian` function is the most accurate, providing a numerical approximation that is typically accurate to within double precision. However, there are a few functions that special care must be taken with in order to be compatible with the complex-step approximation; see Sections 3.3 and 3.4 of [3].

3.2 Algorithm

Algorithm 5: `newtons_method_n`

Newton's method for finding the root of a differentiable, multivariate, vector-valued function.

Given:

- $\mathbf{f}(\mathbf{x})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$)
- $\mathbf{J}(\mathbf{x})$ - Jacobian of $\mathbf{f}(\mathbf{x})$
- $\mathbf{x}_0 \in \mathbb{R}^n$ - initial guess for solution
- $\text{TOL} \in \mathbb{R}$ - *(OPTIONAL)* tolerance (defaults to 10^{-10})
- $k_{\max} \in \mathbb{Z}$ - *(OPTIONAL)* maximum number of iterations (defaults to 200)

Procedure:

1. Default the tolerance to $\text{TOL} = 10^{-10}$ if not input.
2. Default the maximum number of iterations to $k_{\max} = 200$ if not input.
3. Root estimate at the first iteration.

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

4. Initialize $\mathbf{x}_{\text{next}}$.

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

5. Find the root using Newton's method.

   for $k = 1$ **to** $k_{\max}$

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

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

(b) Update the root estimate.

$$\mathbf{x}_{\text{next}} = \mathbf{x}_{\text{curr}} + \mathbf{y}$$

(c) Terminate if converged.

if $\|\mathbf{y}\| < \text{TOL}$
  break
end

(d) Calculate error.

$$\varepsilon = \|\mathbf{x}_{\text{next}} - \mathbf{x}_{\text{curr}}\|$$

(e) Store the updated root estimate for the next iteration.

$$\mathbf{x}_{\text{curr}} = \mathbf{x}_{\text{next}}$$

end

6. Converged root.

$$\mathbf{x}^* = \mathbf{x}_{\text{next}}$$

Return:

- $\mathbf{x}^* \in \mathbb{R}^n$ - root of $\mathbf{f}(\mathbf{x})$

3.3 Solving Nonlinear Systems

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 the root-finding problem

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

which can be solved using Newton's method.

Bibliography

[1] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. 9th ed. Boston, MA: Brooks/Cole, Cengage Learning, 2011.

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

[3] Tamas Kis. *Numerical Differentiation using Finite Difference and Complex-Step Approximations*. 2021. URL: https://tamaskis.github.io/documentation/Numerical_Differentiation_using_Finite_Difference_and_Complex_Step_Approximations.pdf.

[4] Bruce R. Munson et al. *Fundamentals of Fluid Mechanics*. 7th. Hoboken, NJ: John Wiley & Sons, 2013.