

University of Wales Trinity Saint David
Wales Institute of Science & Art



Root Solving Methods or Engine Modelling Part B

Engine Performance and Emissions ACEA7005

Diego Alexander Valencia Flores
2405760
January 2025

Contents

1. Introduction	3
2. MATLAB Functions and Mathematical Techniques	4
2.1 Bisection Method . VPA Solve	4
2.2 Secant Method	4
2.3 Newtons Raphson Method	5
2.4 Fzero (Matlab Built-in)	5
2.5 Fsolve	5
2.7 False Position (Regular Falsi) Method	6
2.8 VPS Solve	6
3. Category Selection of Methodologies	7
4. Results.....	10
5. Conclusion	12
6. References.....	13

1. Introduction

Many fields—ranging from mathematics and computer science to engineering, agriculture, and biomedical research—frequently need solutions to nonlinear equations. When a closed-form expression is unavailable, a numerical approach becomes essential. However, no single algorithm can efficiently handle all types of nonlinear problems. Consequently, root-finding techniques can be broadly categorized into **pure**, **metaheuristic**, and **blended** methods. Pure methods include well-established techniques like bisection, false position, secant, and Newton–Raphson (Badr, Almotairi and Ghamry, 2021).

Root-finding methods are also **fundamental** to the development of **new technologies**, as they tackle **nonlinear equations** that often lack closed-form solutions. In **control systems**, for instance, these algorithms clarify how a system behaves under varied conditions, enabling the design of **reliable** and **efficient** controllers. In **numerical optimization**, similarly, root-finding is used to identify solutions that describe a system’s behavior, which paves the way for **robust** and **high-performance** optimization strategies (Qureshi et al., 2023).

2. MATLAB Functions and Mathematical Techniques

2.1 Bisection Method . VPA Solve

The bisection method is **widely regarded** as the **simplest** and **most robust** approach for identifying the root of a **one-dimensional continuous function** on a **closed interval**. Its principle is straightforward: assume f is continuous over $[a, b]$ and $f(a)$ and $f(b)$ have opposite signs. By the **intermediate value theorem**, there must be at least one $r \in [a, b]$ such that $f(r) = 0$. The method proceeds **iteratively**: each iteration splits the **current interval**, known to contain the root, into two equal parts (Kopecky, 2007).

$$x^{new} = \frac{x_l + x_u}{2}$$

The formula above finds the mid-point value of x between the two original guesses (Chapra and Canale, n.d.).

2.2 Secant Method

The **secant method** is often considered the most widely used **quasi-Newton** approach for **single-variable** problems. It works similarly to Newton's method but **estimates** the derivative of f at each iteration by using the **function values** (and corresponding x -values) from the previous two steps. Specifically, at iteration n , the derivative $f'(x^{(n)})$ is approximated by (Kopecky, 2007):

$$f'(x^{(n)}) \approx \frac{f(x^{(n)}) - f(x^{(n-1)})}{x^{(n)} - x^{(n-1)}}$$

With this approximation, the Newton-like update becomes:

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)}) - (x^{(n)} - x^{(n-1)})}{\frac{f(x^{(n)}) - f(x^{(n-1)})}{x^{(n)} - x^{(n-1)}}}$$

Because the secant method does **not** require an explicit expression for the derivative, it only needs **two initial guesses** to begin the iteration process.

2.3 Newtons Raphson Method

Among all root-finding methods, **Newton–Raphson** is often considered **the most widely used**. The procedure starts with an **initial guess** x_i for the root; from there, one draws a **tangent** at the point $[x_i, f(x_i)]$. Where that tangent crosses the x -axis provides a **better estimate** of the root. Geometrically, the slope at x_i matches the derivative $f'(x_i)$, and we can write (Chapra and Canale, n.d.):

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

Rearranging leads to the Newton-Raphson formula:

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

This derivation can also be framed through **Taylor series** expansion, but the geometric interpretation is often more intuitive.

2.4 Fzero (Matlab Built-in)

The `fzero` command is a function file. The algorithm, created by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods (Mathworks, 2020). `x = fzero(fun,x0)` tries to find a point x where $\text{fun}(x) = 0$ where fun is a function to solve, specified as a handle to a scalar-valued function or the name of such a function. `fun` accepts a scalar x and returns a scalar $\text{fun}(x)$ (Mathworks, 2020).

2.5 Fsolve

Nonlinear system solver that solves a problem specified by

$$F(x) = 0$$

for x , where $F(x)$ is a function that returns a vector value. There are a couple of limitations like:

1. The function to be solved must be continuous
2. When successful, `fsolve` only give one root

3. The default trust-region dogleg method can only be used when the system of equations is square.

The Levenberg-Marquardt and trust-region methods are based on the nonlinear least-squares algorithms also used in [lsqnonlin](#) (Mathworks, 2020).

2.7 False Position (Regular Falsi) Method

False position (also called *regula falsi* or the **linear interpolation method**) uses a **graphical insight**: if $f(x_l)$ is much closer to zero than $f(x_u)$, it is more likely the root lies nearer x_l . To capture this, the method draws a **straight line** between $[x_l, f(x_l)]$ and $[x_u, f(x_u)]$ and computes where it **intersects** the x -axis. Using similar triangles, the new approximation x_r can be found via (Chapra and Canale, n.d.):

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

2.8 VPS Solve

[S](#) = [vpasolve](#)([eqn,var](#)) numerically solves the equation eqn for the variable var using variable-precision arithmetic and returns the solutions with 32 significant digits by default. [S](#) = [vpasolve](#)([eqn,var,init_param](#)) numerically solves the equation eqn for the variable var using the initial guess or search range init_param (Mathworks, 2020).

3. Category Selection of Methodologies

Table 1 Root - Solver Equations

Method	Accuracy	Efficiency	Ease of Use	Complexity	Restrictions / Limitations
Bisection	- Converges reliably	- Inefficient if intervals are large	- Easiest to implement (requires $f(a)*f(b) < 0$).	- Very low algorithmic complexity	- Must bracket the root ($f(a)$ and $f(b)$ must differ in sign).
	- Slow linear convergence rate.	- Requires many iterations for tight tolerances.	- Very robust if bracket is known.	(simple mid-point halving).	- Function must be continuous on $[a,b]$.
	- Error halves each iteration.				- Not exploiting derivative information.
Secant	- Generally good accuracy near root	- Faster than bisection if guesses are good.	- Moderate: requires two initial points, no derivative formula needed.	- Quasi-Newton update (approx. slope).	- Can fail if denominator ($f(x_n) - f(x_{n-1})$) is near 0.
	- Similar to Newton but may converge more slowly.	- Does not always converge if guesses are poor.		- Slightly more code than bisection.	- No guaranteed bracket $\rightarrow$ possible divergence.
					- Not as robust as bisection.
Newton-Raphson	- Potentially very accurate near root	- Extremely fast when derivative is well-defined	- Requires user to provide derivative or derivative code.	- Involves computing or approximating $f'(x)$.	- Derivative must be continuous or near root.
	- Quadratic convergence if well-conditioned.	- But can diverge if derivative is small or guess is poor.	- More advanced than bisection.	- Higher complexity in coding derivative.	- Poor initial guess can lead to divergence. - Not always suited for non-smooth or piecewise functions.

Table 2 Root - Solver methodologies

Method	Accuracy	Efficiency	Ease of Use	Complexity	Restrictions / Limitations
False Position (Regula Falsi)	- Generally more accurate than simple bisection in fewer steps.	- Typically more efficient than bisection (if $f(x_l)$ & $f(x_u)$ differ in magnitude).	- Simple to implement once bracket is known.	- Low complexity—similar to bisection but with a linear weighting.	- Must bracket the root.
	- Uses linear interpolation for next estimate.		- Variation of bisection with linear approximation.		- If $f(x_l)$ or $f(x_u)$ is extremely close to 0, progress can stall.
					- Uses no derivative info.
fzero (MATLAB built-in)	- Generally high accuracy if bracket or good guess is given	- Efficient for 1D problems	- Very easy: $x = fzero(@myFunction, [xLower, xUpper])$	- Internally a hybrid method (Dekker's approach).	- Must provide bracket or near-root guess.
	- Combines bisection, secant, and other interpolation.	- Typically converges quickly if bracket is correct.	- Minimal code.		- Only solves scalar equations (no systems).
fsolve (MATLAB built-in)	- Can achieve high accuracy if the system is well-conditioned.	- Often used for <i>systems</i> of equations	- Moderate: user must handle function handle returning vector $F(x)$, or for single eqn, it can be used but is overkill.	- More code overhead for vector equations, but can handle single eqn too.	- Requires continuous function(s).
		- Efficiency depends on solver choice (trust-region, etc.).			- May converge to non-root if Jacobian is singular or system is ill-posed.
					- By default solves square systems.
vpaNewton (Variable Precision)	- Potentially very high accuracy (arbitrary precision).	- Potentially slower than double-precision Newton	- More advanced setup (Symbolic Toolbox, vpa calls).	- Similar formula to Newton, but internal arithmetic done at arbitrary precision.	- Requires Symbolic Math Toolbox license.
	- Limits floating-point round-off near the root.	- Speed depends on chosen digit precision.	- Derivative or symbolic expression recommended.		- Still reliant on good initial guess to avoid divergence.
					- Overkill for many engineering tasks if double precision is sufficient.

The numerical 1–5 scale provides a straightforward way to rate each method’s performance in different categories. A score of **1** indicates the least favorable condition (for example, a method that converges very slowly or imposes substantial coding overhead), whereas **5** signifies the most advantageous scenario (such as lightning-fast convergence or minimal user setup). In between, scores like **3** or **4** capture middling characteristics—methods that have moderate requirements, reasonable convergence rates, or partial constraints. By quantifying these qualitative features, this scale helps compare methods at a glance without delving into extensive technical details.

Table 3 Criteria Selection 1-5 (low - high)

Method	Accuracy	Efficiency	Ease of Use	Complexity	Restrictions/ Limitations	Total / 25
Bisection	4	2	5	2	2	15
Secant	4	3	4	3	3	17
Newton–Raphson	5	5	3	4	3	20
False Position	4	3	4	2	2	15
fzero (Built-in)	5	4	5	4	3	21
fsolve (Built-in)	5	4	3	4	4	20
vpaNewton (Symbolic)	5	2	3	4	3	17

In the context of a broader discussion on root-finding techniques, these numerical scores reinforce the central message that **no single algorithm** excels in every aspect—some methods are highly precise and efficient but more challenging to implement, while others are easy to use but slower to converge.

4. Results

The script designed to test and by applied to the seven different root-finding methods—Bisection, Secant, Newton–Raphson (double precision), False Position, the MATLAB built-ins `fzero` and `fsolve`, and a custom “`vpaNewton`” method (utilizing variable-precision arithmetic)—to a simple function:

$$f(x) = 10\log_{10}(x) - x$$

This function is defined for $x > 0$. To avoid attempting to evaluate $\log_{10}(0)$, the initial or lower guess is set to a small positive value, typically 10^{-6} . For methods requiring an interval, the range $[10^{-6}, 4]$ is used.

Each method records how many iterations it completes (up to a maximum) and measures how long it takes using `tic` and `toc`. The final computed roots, iteration counts, function values at those roots, and total execution times are all assembled into a summary table.

----- RESULTS -----				
Method	Root	f_at_root	Iterations	Time_sec
{'Bisection' }	{'1.371289' }	{'5.3899114e-08' }	{'25' }	{'0.001986' }
{'Secant' }	{'1.371289' }	{'2.7351454e-12' }	{'31' }	{'0.002236' }
{'Newton-Raphson' }	{'1.371289' }	{'0' }	{'13' }	{'0.0009252' }
{'False Position' }	{'1.371289' }	{'9.6589566e-08' }	{'349' }	{'0.001424' }
{'fzero (Built-in)' }	{'1.371289' }	{'-1.0101919e-11' }	{'1' }	{'0.01506' }
{'fsolve (Built-in)' }	{'1.371289' }	{'-2.284839e-13' }	{'1' }	{'0.04479' }
{'vpaNewton (Symbolic)' }	{'1.371289' }	{'0' }	{'13' }	{'0.1136' }

Figure 1 First Iteration

----- RESULTS -----				
Method	Root	f_at_root	Iterations	Time_sec
{'Bisection' }	{'1.371289' }	{'5.3899114e-08' }	{'25' }	{'0.00125' }
{'Secant' }	{'1.371289' }	{'2.7351454e-12' }	{'31' }	{'0.001779' }
{'Newton-Raphson' }	{'1.371289' }	{'0' }	{'13' }	{'0.001231' }
{'False Position' }	{'1.371289' }	{'9.6589566e-08' }	{'349' }	{'0.001164' }
{'fzero (Built-in)' }	{'1.371289' }	{'-1.0101919e-11' }	{'1' }	{'0.0004215' }
{'fsolve (Built-in)' }	{'1.371289' }	{'-2.284839e-13' }	{'1' }	{'0.001201' }
{'vpaNewton (Symbolic)' }	{'1.371289' }	{'0' }	{'13' }	{'0.03615' }

Figure 2 Second Iteration

RESULTS				
Method	Root	f_at_root	Iterations	Time_sec
{'Bisection' }	{'1.371289'}	{'5.3899114e-08' }	{'25' }	{'0.001312' }
{'Secant' }	{'1.371289'}	{'2.7351454e-12' }	{'31' }	{'0.001681' }
{'Newton-Raphson' }	{'1.371289'}	{'0' }	{'13' }	{'0.001238' }
{'False Position' }	{'1.371289'}	{'9.6589566e-08' }	{'349' }	{'0.001245' }
{'fzero (Built-in)' }	{'1.371289'}	{'-1.0101919e-11' }	{'1' }	{'0.0004721' }
{'fsolve (Built-in)' }	{'1.371289'}	{'-2.284839e-13' }	{'1' }	{'0.001278' }
{'vpasolve (Symbolic)' }	{'1.371289'}	{'0' }	{'13' }	{'0.03445' }

Figure 3 Third Iteration

RESULTS				
Method	Root	f_at_root	Iterations	Time_sec
{'Bisection' }	{'1.371289'}	{'5.3899114e-08' }	{'25' }	{'0.0008922' }
{'Secant' }	{'1.371289'}	{'2.7351454e-12' }	{'31' }	{'0.000627' }
{'Newton-Raphson' }	{'1.371289'}	{'0' }	{'13' }	{'0.0006522' }
{'False Position' }	{'1.371289'}	{'9.6589566e-08' }	{'349' }	{'0.0008897' }
{'fzero (Built-in)' }	{'1.371289'}	{'-1.0101919e-11' }	{'1' }	{'0.005792' }
{'fsolve (Built-in)' }	{'1.371289'}	{'-2.284839e-13' }	{'1' }	{'0.002493' }
{'vpasolve (Symbolic)' }	{'1.371289'}	{'0' }	{'13' }	{'0.0326' }

Figure 4 Fourth Iteration

RESULTS				
Method	Root	f_at_root	Iterations	Time_sec
{'Bisection' }	{'1.371289'}	{'5.3899114e-08' }	{'25' }	{'0.0001939' }
{'Secant' }	{'1.371289'}	{'2.7351454e-12' }	{'31' }	{'0.0001919' }
{'Newton-Raphson' }	{'1.371289'}	{'0' }	{'13' }	{'0.0001334' }
{'False Position' }	{'1.371289'}	{'9.6589566e-08' }	{'349' }	{'0.0003081' }
{'fzero (Built-in)' }	{'1.371289'}	{'-1.0101919e-11' }	{'1' }	{'0.006362' }
{'fsolve (Built-in)' }	{'1.371289'}	{'-2.284839e-13' }	{'1' }	{'0.003875' }
{'vpasolve (Symbolic)' }	{'1.371289'}	{'0' }	{'13' }	{'0.0367' }

Figure 5 Fifth Iteration

Something to keep as a note is that as the code runs each time without changing anything and the time to run each method is faster and faster as it is shown from the first iteration to the fifth iteration.

Finally method fzero and the bisection methods were used to determine the oxygen balance for task 3. As show in this picture.

Table 4 Method - Iteration - Computational Time - Error for the original problem to determine the Oxygen Balance.

Method	Temperature	Pressure	Fuel Eq. Ratio	Root	Error	Total Iterations	Comp Time (s)	Oxygen Balance
{'Bisection' }	2700	70	0.95	0.1091	6.1772e-10	31	0.0007122	0.93514
{'fzero' }	2700	70	0.95	0.1091	0	4	0.0003526	0.93514
{'Original Function'}	2700	70	0.95	0.099	0.008748	1	2.81e-05	0.93203

5. Conclusion

The analysis of various root-finding techniques indicates that **no single method** offers an optimal solution for every problem. Each approach—from bracket-based algorithms like Bisection and False Position to derivative-dependent methods such as Newton–Raphson and Secant—comes with its own strengths and trade-offs in terms of speed, coding complexity, dependency on derivatives, and guaranteed convergence. Similarly, the choice between built-in MATLAB solvers (fzero, fsolve) and symbolic approaches (e.g., variable-precision arithmetic) hinges on the required accuracy, the nature of the function, and the resources or toolboxes available.

Selecting the “best” root-finding method should therefore be guided by the **specific problem context**—including the function’s behavior, the level of precision needed, and the computational overhead or time constraints. A bracket-based method may be preferable when guaranteed convergence is crucial, whereas a derivative-based technique might excel in scenarios where rapid convergence is paramount and a suitable starting guess or derivative information is available. By carefully assessing these criteria, it becomes possible to match the strengths of each method to the exact requirements of a given problem.

6. References

Badr, E., Almotairi, S. and Ghamry, A.E. (2021). A Comparative Study among New Hybrid Root Finding Algorithms and Traditional Methods. *Mathematics*, [online] 9(11), p.1306. doi:<https://doi.org/10.3390/math9111306>.

Chapra, S. and Canale, R. (n.d.). *Numerical Methods for Engineers SEVENTH EDITION*. [online] Available at: <https://gdcboysang.ac.in/About/Droid/uploads/Numerical%20Methods.pdf> [Accessed 16 Jan. 2025].

Kopecky, K. (2007). *LECTURE NOTES ECO 613/614 FALL 2007 Root-Finding Methods*. [online] Available at: https://paulklein.se/newsite/teaching/Notes_RootFindingMethods.pdf [Accessed 16 Jan. 2025].

Mathworks (2020). Mel Frequency Cepstral Coefficients. *Mathworks.com*. [online] doi:<https://doi.org/1061245463.woff2>.

Qureshi, S., Argyros, I.K., Soomro, A., Krzysztof Gdawiec, Shaikh, A.A. and Evren Hincal (2023). A new optimal root-finding iterative algorithm: local and semilocal analysis with polynomiography. *Numerical Algorithms*, 95(4), pp.1715–1745. doi:<https://doi.org/10.1007/s11075-023-01625-7>.