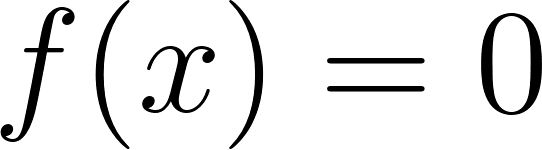
# Finding Zeros

Solving [nonlinear equations](https://en.wikipedia.org/wiki/Nonlinear_system) can be a challenging task, but finding the zeros, or the values of [](https://www.codecogs.com/eqnedit.php?latex=x#0) for which [](https://www.codecogs.com/eqnedit.php?latex=f(x)%20%3D%200#0), is crucial in various applications. In this content, we explore several techniques to determine the zeros of complicated nonlinear functions. We'll discuss the pros and cons of each method and provide examples to help you understand the underlying concepts. The methods covered include:

* Trial-and-Error: A basic approach, though not very efficient, involving plugging in various [](https://www.codecogs.com/eqnedit.php?latex=x#0)-values to see if [](https://www.codecogs.com/eqnedit.php?latex=f(x)#0) equals 0.
* [Bisection Method](https://en.wikipedia.org/wiki/Bisection_method): A divide-and-conquer approach that iteratively narrows down the search interval by bisecting it and identifying the subinterval containing the zero.
* [Newton's Method](https://en.wikipedia.org/wiki/Newton%27s_method): A popular and fast-converging technique that uses linear approximations and iterative calculations to find the zero of a function.

By understanding these techniques, you'll be better equipped to tackle complicated nonlinear functions and find their zeroes with confidence.

Finding the zeros of functions is a fundamental problem in many areas of science, engineering, finance, and more. Understanding techniques such as bisection and Newton's method to find zeros (or roots) of functions is significant to the concept of simulation for several reasons:

* Fundamental Problem in Many Simulations: Many simulation problems boil down to finding the zeros of some function. For instance, in computational fluid dynamics, one might want to find the mach number at which a shock occurs, or in financial mathematics, one might want to find an implied volatility that matches market prices.
* Nonlinear Equations: Many physical, biological, and economic systems are inherently nonlinear. Nonlinear equations frequently don't have closed-form solutions, making numerical methods like bisection and Newton's method essential for finding solutions.
* Convergence Properties: Bisection and Newton's method have different convergence properties. Bisection has a linear convergence rate but is guaranteed to converge if the function changes sign in the interval. Newton's method can have quadratic convergence (which is faster) but isn't always guaranteed to converge. Understanding these properties can help one choose the appropriate method for a particular simulation.
* Iterative Nature of Simulations: Simulations often involve iterative methods, and both bisection and Newton's method are iterative in nature. Gaining a feel for how iterative methods work, converge, or diverge is crucial for developing and troubleshooting simulations.
* Initial Guess Dependency: Methods like Newton's method are sensitive to the initial guess. Understanding this dependency is essential in simulations, as it can guide the choice of initial conditions or highlight the need for multiple initial guesses to ensure robust results.
* Hybrid Methods: In many real-world scenarios, practitioners combine methods to exploit the advantages of each. For instance, they might use bisection to bracket a root and then switch to Newton's method for faster convergence. Understanding the basic techniques makes it easier to develop or employ such hybrid methods.
* Understanding Limitations and Pitfalls: By studying methods like bisection and Newton's method, one gains insights into potential pitfalls in numerical simulations, such as divergence or getting stuck at a local extremum. Recognizing these issues helps in developing more robust simulation tools.
* Transferability of Concepts: The ideas behind root-finding, such as bracketing a solution, iterative refinement, and checking for convergence, are transferable to other areas of simulation and numerical methods. Thus, mastering these concepts in the context of root-finding prepares the practitioner for other simulation challenges.
* Educational Value: Just as with differential equations, techniques like bisection and Newton's method are often introduced early in numerical analysis courses because they offer foundational insights into how numerical methods can approach complex problems. This foundational knowledge aids in understanding more advanced numerical techniques.

The ability to find zeros of functions numerically is a fundamental skill in the toolkit of anyone involved in simulation, providing the capability to solve complex nonlinear equations that often arise in real-world problems.

# Bisection

## Bisection Example R Code

This R code defines a function find\_bisection\_root to find the root (zero) of a given nonlinear function fn within a specified interval [x1, x2] using the bisection method. The function takes six parameters:

* fn: The nonlinear function whose root is to be found.
* x1: The lower bound of the interval.
* x2: The upper bound of the interval.
* max\_iter (default 100): The maximum number of iterations allowed.
* tolerance (default 0.0001): The tolerance level to determine the convergence of the method.
* verbose (default FALSE): A boolean flag to control the display of intermediate values at each iteration.

The function starts by checking if there is a root within the given interval by testing the signs of fn(x1) and fn(x2). If the signs are the same, there is no root in the interval, and the function returns NULL.

If there is a root, the function iteratively bisects the interval and updates either x1 or x2 based on the sign of the function at the midpoint x3. The process continues until the difference between x1 and x2 is less than or equal to the specified tolerance or the maximum number of iterations is reached. The function then returns the midpoint x3 as the root.

The condition if (fn(x3) \* fn(x1) > 0) checks the sign of the product fn(x3) \* fn(x1) to determine whether the root lies in the interval [x3, x2] or [x1, x3]. Here's why:

In the bisection method, we start with an initial interval [x1, x2] that contains the root. At each iteration, we calculate the midpoint x3 of the interval and evaluate the function fn at x3. Then, based on the sign of fn(x3), we update the interval by setting either x1 = x3 or x2 = x3.

However, if fn(x3) has the same sign as fn(x1), then the root cannot be in the interval [x1, x3] because the function does not change sign between these points. In other words, the function values at x1 and x3 have the same sign, and therefore, there cannot be a root in between them. Therefore, we must update x1 to x3 and continue searching in the interval [x3, x2].

On the other hand, if fn(x3) has the opposite sign to fn(x1), then the root must lie in the interval [x1, x3]. This is because the function changes sign between x1 and x3, and by the intermediate value theorem, there must be a root between these points. In this case, we update x2 to x3 and continue searching in the interval [x1, x3].

Hence, the condition if (fn(x3) \* fn(x1) > 0) checks whether fn(x3) has the same sign as fn(x1), which tells us which subinterval to update.

*# Define the find\_bisection\_root function to find the root of a nonlinear*   
*# function using the bisection method*  
find\_bisection\_root <- **function**(fn, x1, x2, max\_iter = 100, tolerance = .0001,   
 verbose = FALSE) {  
   
 *# Check if there is a root within the given interval*  
 **if** (sign(fn(x1)) == sign(fn(x2))) {  
 message("No zero in this interval! :)")  
 return(NULL)  
 }  
   
 *# Iterate through the specified number of maximum iterations*  
 **for** (i **in** 1:max\_iter) {  
 *# Calculate the midpoint of the current interval*  
 x3 <- (x1 + x2) / 2  
   
 *# Update the interval based on the function's sign at the midpoint*  
 **if** (fn(x3) \* fn(x1) > 0) {  
 x1 <- x3  
 } **else** {  
 x2 <- x3  
 }  
   
 *# Check if we are close enough to 0*  
 **if** (all.equal(fn(x3), 0, tolerance = tolerance) == TRUE) {  
 return(x3)  
 }  
   
 *# Print the intermediate values if verbose mode is enabled*  
 **if** (verbose) {  
 cat(i, ": [", x1, ", ", x2, "]\n", sep = "")  
 }  
 }  
   
 *# Return the final midpoint as the root*  
 return(x3)  
}  
  
*# Define the test function*  
fn <- **function**(x) {  
 x^2 - 2  
}  
  
*# Call the find\_bisection\_root function with the test function*  
find\_bisection\_root(fn, 1, 2, max\_iter = 4, tolerance = 1e-6,   
 verbose = TRUE)

## 1: [1, 1.5]  
## 2: [1.25, 1.5]  
## 3: [1.375, 1.5]  
## 4: [1.375, 1.4375]

## [1] 1.4375

find\_bisection\_root(fn, 1, 2, max\_iter = 100, tolerance = 1e-6,   
 verbose = TRUE)

## 1: [1, 1.5]  
## 2: [1.25, 1.5]  
## 3: [1.375, 1.5]  
## 4: [1.375, 1.4375]  
## 5: [1.40625, 1.4375]  
## 6: [1.40625, 1.421875]  
## 7: [1.414062, 1.421875]  
## 8: [1.414062, 1.417969]  
## 9: [1.414062, 1.416016]  
## 10: [1.414062, 1.415039]  
## 11: [1.414062, 1.414551]  
## 12: [1.414062, 1.414307]  
## 13: [1.414185, 1.414307]  
## 14: [1.414185, 1.414246]  
## 15: [1.414185, 1.414215]  
## 16: [1.4142, 1.414215]  
## 17: [1.414207, 1.414215]  
## 18: [1.414211, 1.414215]  
## 19: [1.414213, 1.414215]  
## 20: [1.414213, 1.414214]

## [1] 1.414214

*# Find the root using R's built-in uniroot function*  
uniroot(fn, c(-1,3), tol = 1e-6)$root

## [1] 1.414214

*# Calculate the analytical solution*  
sqrt(2)

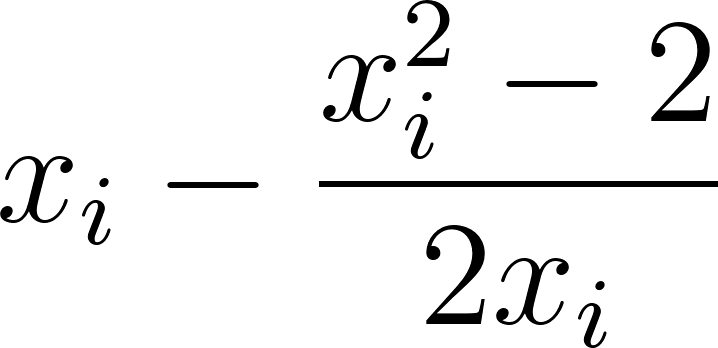
## [1] 1.414214

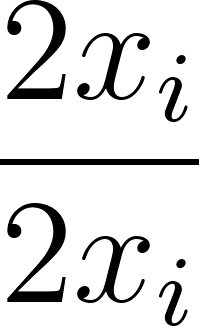
## Bisection Example Python Code

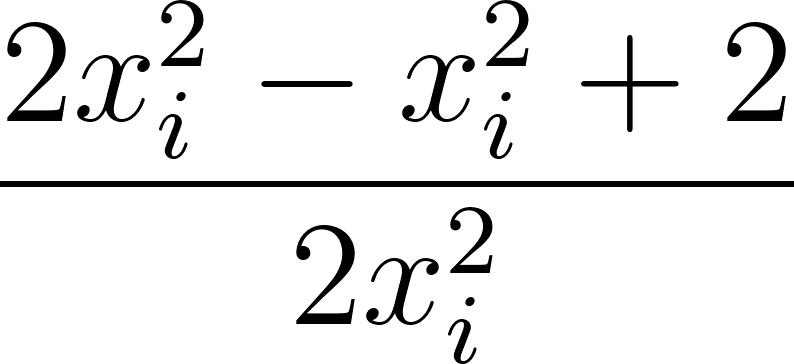
[Python Notebook](https://colab.research.google.com/drive/1vSALbBJbz2Lb7mBuljO9RlUUQ5eBIBQj#scrollTo=LqepsDUQUs0e)

# Newton’s Method

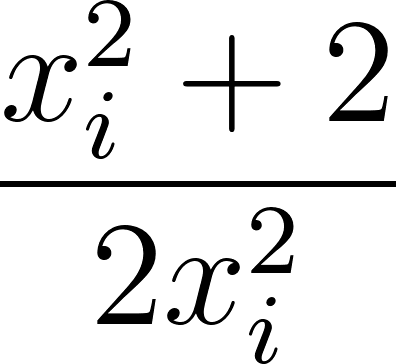
The algebra steps to obtain the equivalent expression shown on the slide are as follows:

[](https://www.codecogs.com/eqnedit.php?latex=x_i-%5Cdfrac%7Bx_i%5E2-2%7D%7B2x_i%7D#0)

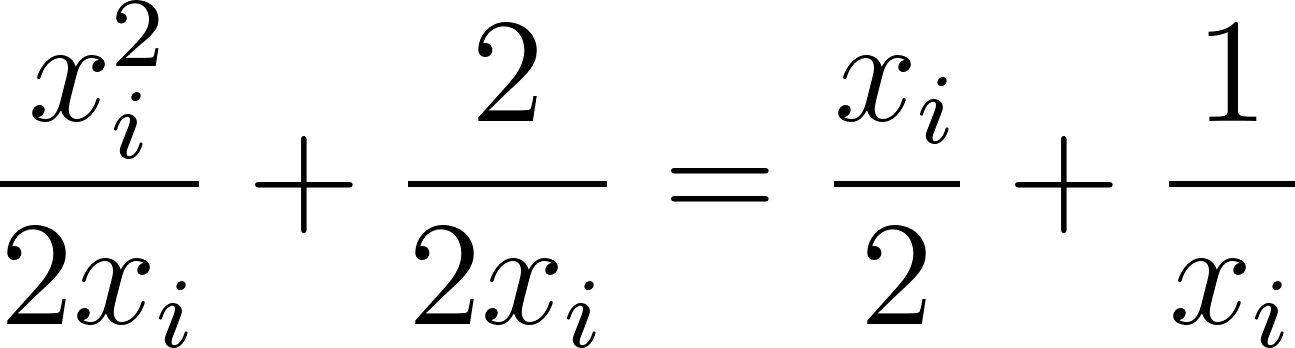
Multiply the first term by [](https://www.codecogs.com/eqnedit.php?latex=%5Cdfrac%7B2x_i%7D%7B2x_i%7D#0) to obtain a common denominator and combine the fractions:

[](https://www.codecogs.com/eqnedit.php?latex=%5Cdfrac%7B2x_i%5E2-x_i%5E2%2B2%7D%7B2x_i%5E2%7D#0)

Combine like terms:

[](https://www.codecogs.com/eqnedit.php?latex=%5Cdfrac%7Bx_i%5E2%2B2%7D%7B2x_i%5E2%7D#0)

Split the fraction into two terms and simplify:

[](https://www.codecogs.com/eqnedit.php?latex=%5Cdfrac%7Bx_i%5E2%7D%7B2x_i%7D%2B%5Cdfrac%7B2%7D%7B2x_i%7D%3D%5Cdfrac%7Bx_i%7D%7B2%7D%2B%5Cdfrac%7B1%7D%7Bx_i%7D#0)

## Newton’s Method Example R Code

This code defines a function find\_newton\_root to find the root of a given nonlinear function fn using the Newton-Raphson method. The function takes several arguments: fn is the nonlinear function to find the root of, x0 is the initial guess, max\_iter is the maximum number of iterations to run the algorithm for, tol is the tolerance level for convergence, and verbose is a logical flag indicating whether to print out intermediate results during the algorithm.

The find\_newton\_root function first extracts the expression inside the function using body(fn) and assigns it to the variable expr. It then defines a new function f\_prime that calculates the derivative of fn using the D function and the expression expr.

Next, the function starts at the initial guess x0 and applies Newton's method to update the guess and get closer to the root. At each iteration, it computes the new guess new\_x using the formula new\_x <- x - (fn(x) / f\_prime(x)). If the new guess is close enough to 0 (within the given tolerance level tol), the algorithm stops and returns the current guess as the root.

Otherwise, the algorithm updates the current guess to the new guess x <- new\_x, and prints out the current guess if verbose mode is enabled. If the maximum number of iterations max\_iter is reached without finding a root, the algorithm returns the final guess as the root.

*# Define the find\_newton\_root function to find the root of a nonlinear*   
*# function using Newton-Raphson method*  
find\_newton\_root <- **function**(fn, x0, max\_iter = 100, tol = 1e-15,   
 verbose = FALSE) {  
   
 *# Get the expression inside the function*  
 expr <- body(fn)  
   
 *# Take the derivative of the function*  
 f\_prime <- **function**(x) eval(D(expr, "x"))  
   
 *# Display some output if verbose mode is enabled*  
 **if** (verbose) {  
 message(paste0("Finding zeros of ", deparse(body(fn))))  
 message(paste0("Derivative is ", deparse(D(expr, "x"))))  
 }  
   
 *# Apply Newton's method starting at x0*  
 x <- x0  
 **for** (i **in** 2:max\_iter) {  
 new\_x <- x - (fn(x) / f\_prime(x))  
   
 *# Check if the proposed new\_x is close enough to 0*  
 **if** (all.equal(fn(new\_x), 0, tolerance = tol) == TRUE) {  
 **if** (verbose) message(paste("Converged after", (i - 1), "iterations"))  
 **break**  
 }   
   
 x <- new\_x  
   
 **if** (verbose) {  
 cat(paste0(i, ": ", x, "\n"))  
 }  
 }  
   
 *# Return the final x value as the root*  
 x  
}  
  
*# Define the test function*  
fn <- **function**(x) x^2 - 2  
  
*# Call the find\_newton\_root function with the test function, initial guess*   
*# x0 = 1, 100 iterations, and verbose mode enabled*  
find\_newton\_root(fn = fn, x0 = 1, max\_iter = 100, tol = 1e-7, verbose = TRUE)

## Finding zeros of x^2 - 2

## Derivative is 2 \* x

## 2: 1.5  
## 3: 1.41666666666667  
## 4: 1.41421568627451

## Converged after 4 iterations

## [1] 1.414216

## Newton’s Method Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1vSALbBJbz2Lb7mBuljO9RlUUQ5eBIBQj#scrollTo=7mXSiqdHUs0i)