Numerical Analysis Series

Zeros of Polynomials & Müller's Method

(Finding Real and Complex Roots)

Author
Shahad Uddin
shahaduddin.com

Date
February 2, 2026
Version 1.0

1. Theoretical Background

Fundamental Theorem of Algebra

A polynomial of degree n has the form:

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

If $P(x)$ is a polynomial of degree $n \geq 1$, then $P(x) = 0$ has at least one (possibly complex) root. While Newton's method finds roots effectively, it struggles with complex roots if the starting point is real. **Müller's Method** generalizes the Secant Method to address this.

The Logic of Müller's Method

The Secant Method approximates the function using a **straight line** passing through two points. Müller's Method takes this a step further by approximating the function using a **parabola** passing through three points (x_0, x_1, x_2).

The root of this parabola gives the next approximation, x_3.

Advantages:

- It converges faster than the Secant method (Convergence order ≈ 1.84).
- **It can find complex roots**, because the quadratic formula naturally produces complex numbers (square root of a negative discriminant).

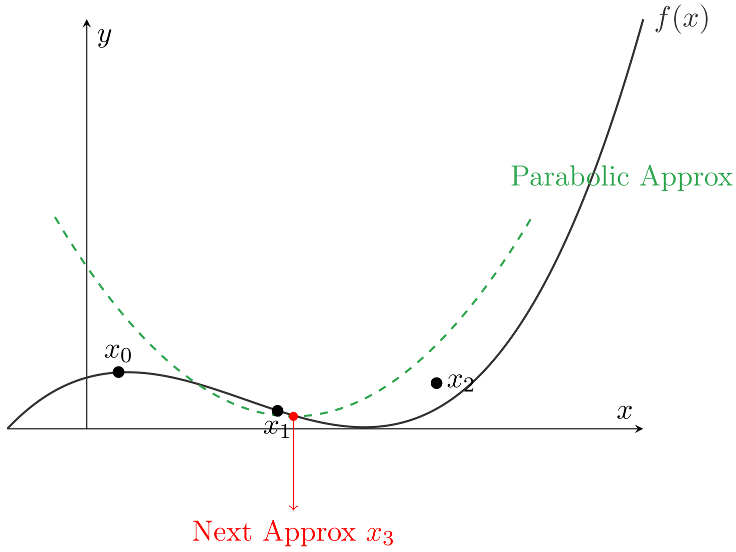

Fig 1. Müller's method fits a parabola through x_0, x_1, x_2.

The Iterative Formula: Given three points x_0, x_1, x_2, we calculate:

$$x_3 = x_2 - \frac{2c}{b + \operatorname{sgn}(b)\sqrt{b^2 - 4ac}} \tag{1}$$

The sign in the denominator is chosen to maximize the magnitude of the denominator (for stability).

2. Code Implementations

The following implementations solve $x^3 + 1 = 0$. Note that the roots are -1 and the complex pair $0.5 \pm 0.866i$.

>PythonImplementation *(.py)*

```python
import cmath

def muller_method(f, x0, x1, x2, tol=1e-6, max_iter=100):
    """
    Muller's Method: Uses a parabolic arc to find roots.
    Supports complex roots automatically.
    """
    for i in range(max_iter):
        h1 = x1 - x0
        h2 = x2 - x1
        d1 = (f(x1) - f(x0)) / h1
        d2 = (f(x2) - f(x1)) / h2
        a = (d2 - d1) / (h2 + h1)
        b = a * h2 + d2
        c = f(x2)

        # Quadratic formula to find root of parabola
        disc = cmath.sqrt(b**2 - 4*a*c)
        # Choose denominator for stability
        if abs(b + disc) > abs(b - disc):
            den = b + disc
        else:
            den = b - disc

        dx = -2 * c / den
        x3 = x2 + dx

        if abs(dx) < tol:
            return x3

        x0, x1, x2 = x1, x2, x3

    return x2

# Example Usage
f = lambda x: x**3 + 1
root = muller_method(f, -1.5, -0.5, 0.5)
print(f"Root: {root}")
```

>FortranImplementation (.f90)

```fortran
program muller_method
    implicit none
    complex :: x0, x1, x2, x3, h1, h2, d1, d2, a, b, c, disc, den, dx
    real :: tol = 1e-6
    integer :: i

    ! Muller's method requires 3 starting points
    x0 = (-1.5, 0.0); x1 = (-0.5, 0.0); x2 = (0.5, 0.0)

    do i = 1, 100
        h1 = x1 - x0
        h2 = x2 - x1
        d1 = (f(x1) - f(x0)) / h1
        d2 = (f(x2) - f(x1)) / h2

        a = (d2 - d1) / (h2 + h1)
        b = a * h2 + d2
        c = f(x2)

        ! Parabolic discriminant
        disc = sqrt(b**2 - 4.0 * a * c)

        ! Choose denominator sign for stability
        if (abs(b + disc) > abs(b - disc)) then
            den = b + disc
        else
            den = b - disc
        end if

        dx = -2.0 * c / den
        x3 = x2 + dx

        if (abs(dx) < tol) exit
        x0 = x1; x1 = x2; x2 = x3
    end do
    print *, "Root: ", x3

contains
    complex function f(x)
        complex, intent(in) :: x
        f = x**3 + 1.0
    end function f
end program
```

>C++Implementation (.cpp)

```cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <iomanip>
#include <functional>

/**
 * Muller's Method for root finding.
 * Capable of finding complex roots even for real coefficients.
```

```cpp
 */
std::complex<double> muller_method(
    std::function<std::complex<double>(std::complex<double>)> f,
    std::complex<double> x0, std::complex<double> x1, std::complex<double> x2,
    double tol = 1e-6, int max_iter = 100)
{
    for (int i = 0; i < max_iter; ++i) {
        std::complex<double> h1 = x1 - x0;
        std::complex<double> h2 = x2 - x1;
        std::complex<double> d1 = (f(x1) - f(x0)) / h1;
        std::complex<double> d2 = (f(x2) - f(x1)) / h2;

        std::complex<double> a = (d2 - d1) / (h2 + h1);
        std::complex<double> b = a * h2 + d2;
        std::complex<double> c = f(x2);

        std::complex<double> disc = std::sqrt(b * b - 4.0 * a * c);

        std::complex<double> den1 = b + disc;
        std::complex<double> den2 = b - disc;
        // Choose largest denominator
        std::complex<double> den = (std::abs(den1) > std::abs(den2)) ? den1 : den2;

        std::complex<double> dx = -2.0 * c / den;
        std::complex<double> x3 = x2 + dx;

        if (std::abs(dx) < tol) return x3;
        x0 = x1; x1 = x2; x2 = x3;
    }
    return x2;
}

int main() {
    auto f = [](std::complex<double> x) { return std::pow(x, 3) + 1.0; };
    std::complex<double> root = muller_method(f, -1.5, -0.5, 0.5);
    std::cout << "Root: " << root.real() << " + " << root.imag() << "i" << std::endl;
    return 0;
}
```