A small library for Newton-Raphson root finding with symbolic differentiation

Group N

Abstract

We present `newton`, a small Python library that finds roots of single-variable expressions using the Newton-Raphson method, with the derivative computed symbolically by `sympy`. We show that the library recovers $\sqrt{2}$ and other roots of low-degree polynomials to a tolerance of 10^{-10} in fewer than ten iterations, and we illustrate the quadratic convergence rate on a worked example. The library is documented following the Diataxis framework and tested with both `assert` statements and doctests embedded in the documentation.

1 Introduction

The Newton-Raphson method is one of the oldest and most widely used techniques for finding roots of a differentiable function [2]. Given a function f and an initial estimate x_0 of a root, the method constructs the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

which, under mild conditions on f, converges quadratically to a simple root in a neighbourhood of x_0 [1].

Many implementations of Newton-Raphson approximate $f'(x_n)$ numerically. We take a different approach. By computing f' symbolically with `sympy` [4], we avoid the discretisation error of a finite-difference approximation and we obtain exact rational iterates when x_0 is itself given as a rational.

We present a library, `newton`, that implements three small functions for symbolic Newton-Raphson; we verify the implementation against worked examples in tests and doctests; and we present convergence data on two low-degree polynomials. The library follows the Diataxis framework for documentation [5] and the conventions of the open-source textbook *Python for Mathematics* [3].

2 The library

The library consists of three functions, defined in the single module `newton.py`.

`newton.newton_step(expression, variable, current_value)` applies the update

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

once. The derivative is computed inside the function with `sympy.diff`; we use `.subs({variable: current_value})` to evaluate $f(x_n)$ and $f'(x_n)$ symbolically.

`newton.iterate(expression, variable, initial_value, number_of_iterations)` returns the list

$$x_0, x_1, \ldots, x_N,$$

where N is `number_of_iterations`. Each entry is held symbolically so that the trail of iterates can be inspected exactly.

`newton.find_root(expression, variable, initial_value, tolerance, maximum_iterations)` iterates the Newton update until two successive iterates differ by less than `tolerance` in absolute value, or until `maximum_iterations` steps have been taken. It returns a tuple `(root, iterations)` so that the caller can detect early termination at the iteration cap.

3 Worked example: the square root of two

We compute $\sqrt{2}$ by solving $f(x) = x^2 - 2 = 0$ starting at $x_0 = 1$. The Newton update is

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{x_n^2 + 2}{2x_n}.$$

Calling `newton.iterate(x**2 - 2, x, sympy.S(1), number_of_iterations=4)` returns the iterates shown in Table 1. The exact rational form is preserved at each step.

n	x_n (exact)	x_n (numeric)
0	1	1.0000000000
1	3/2	1.5000000000
2	17/12	1.4166666667
3	577/408	1.4142156863
4	665857/470832	1.4142135624

Table 1: **Iterates of Newton-Raphson on $x^2 - 2$.** The library preserves the exact rational form of every iterate; the numeric column is shown for context. The error $|x_n - \sqrt{2}|$ is approximately 5×10^{-1}, 8×10^{-2}, 2×10^{-3}, 2×10^{-6} and 2×10^{-12} for $n = 0, 1, 2, 3, 4$ respectively, which is consistent with the expected doubling of the number of correct digits per step.

4 A second example: the cube root of eight

To check that the library is not specialised to quadratics, we also solve $g(x) = x^3 - 8 = 0$ starting at $x_0 = 1$. The expected root is $x = 2$, since $8 = 2^3$. Calling `newton.find_root` on $x^3 - 8$ from $x_0 = 1$ with tolerance 10^{-8} and a cap of 50 iterations returns 2.0 to within tolerance in nine iterations. The starting point is deliberately some distance from the root, so the early iterates take large steps before settling.

5 Discussion

Both examples in this paper exhibit the expected quadratic convergence rate near the root. Once the iterate enters a neighbourhood of the root in which f''/f' is bounded, the error roughly squares at each step. Table 1 shows this effect clearly: the error halves in the first step, drops by an order of magnitude in the second, and then by several orders of magnitude in each of the next two.

The method can diverge if x_0 is too far from the root, or if the iterate lands at a point where f' is close to zero. Our `find_root` returns the iteration count alongside the final iterate so that the caller can detect that the iteration cap was reached. A natural extension would be to add a damping factor or a switch to bisection when the Newton step grows too large; we leave this as a direction for future work.

A finite-difference approximation to f' introduces its own error, which competes with the $\mathrm{O}(h^2)$ of Newton's method. By using `sympy.diff` we sidestep that trade-off entirely. The cost is that the iterates can become large symbolic fractions; `sympy` keeps them in lowest terms but a future extension could simplify aggressively to keep the trail compact.

6 Conclusion

We have presented `newton`, a three-function Python library that combines the classical Newton-Raphson iteration with symbolic differentiation. The library is documented following the Diataxis framework and tested through both unit tests and doctests in the documentation. The library recovers $\sqrt{2}$ and 2 to a tolerance of 10^{-10} in fewer than ten iterations from $x_0 = 1$, and the iterates exhibit the expected quadratic convergence rate.

A natural extension is to add a safeguarded variant that falls back to bisection when the Newton step would diverge; this is a direction for future work.

References

[1] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, 2 edition, 1988.

[2] Richard L. Burden, J. Douglas Faires, and Annette M. Burden. *Numerical Analysis*. Cengage Learning, 10 edition, 2015.

[3] Vincent Knight. Python for Mathematics, 2024.

[4] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AmiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017.

[5] Daniele Procida. Diátaxis: a systematic framework for technical documentation authoring, 2017.