Lecture 1: Finding Solutions

Introduction to Data Analysis Techniques

1 Data Analysis in Science

Definition 1.1 (Analytical Method and Analytical Solution)**.** [def:analytical-method] When we use *analytical methods* to solve a problem, we are manipulating the mathematical expressions until unknowns are being isolated into a standard mathematical form.
An *analytical solution* is a solution for a problem such that the unknowns are expressed in a closed form.

Example 1.1 (Analytical Solution of a Quadratic Equation)**.** [eg:quadratic-equation] Consider the quadratic equation

$$ax^2 + bx + c = 0, \tag{1}$$

where a, b, and c are real constants and $a \neq 0$. When $b^2 - 4ac \geq 0$, the real solutions are given by the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2}$$

Definition 1.2 (Numerical Method and Numerical Solution)**.** [def:numerical-method] *Numerical methods* are methods to obtain numerical solutions.
A *numerical solution* is a solution for a problem such that the unknowns are expressed in an numerical form.

Example 1.2 (Numerical Solution of a Trigonometric Equation)**.** [eg:trig-equation] Consider the trigonometric equation

$$x = \beta \cos x \tag{3}$$

where β is a constant.
It is very difficult to find an analytical solution for this equation using algebraic manipulations. However, we can use numerical methods to find an approximate solution for specific β values. The numerical solutions can be

$$\beta = 1, \quad x \approx 0.73909, \quad \beta = 0.5, \quad x \approx 0.45018$$

Remark 1.1 (Advantages and Disadvantages of Numerical Methods)*.* [rem:numerical-methods]

1. A numerical run uses specified input values; unknown parameters may themselves be quantities to estimate
2. Changing the input values may require another run
3. Numerical solutions are generally approximate; their accuracy depends on the method and stopping rule
4. Analytical formulas can explain parameter dependence, while numerical methods can provide useful values even when a closed form exists

Redirect (Finite Index Set) [def:finite-index-set]
Moved to definition finite index set.

Redirect (Pairing of Functions) [def:pairing-functions]
Moved to definition pairing functions.

2 Fixed Point Iteration Method

Definition 2.1 (Fixed Point Iteration Method)**.** [def:fixed-point-iteration-method] The idea for *fixed point iteration method* is to first find a function $g(x)$ such that root r is a fixed point of $g(x)$, i.e., $g(r) = r$

Convention 2.1 (Procedure of Fixed Point Iteration Method)**.** [conv:fixed-point-iteration] The classical procedure of fixed point iteration method is as follows:

1. Find a function $g(x)$ such that root r is a fixed point of $g(x)$, i.e., $g(r) = r$.
2. Choose an initial guess x_0.
3. Choose a maximum number of iterations N
4. Iterate $x_{n+1} = g(x_n)$ for N times

Remark 2.1 (Error Analysis of Fixed Point Iteration Method). [rem:fixed-point-iteration-error] Successive iterates need not lie on opposite sides of a fixed point, and their distance alone is not a bound on the root error
Suppose g is differentiable on an open interval containing a closed interval $I = [a, b]$, where $a < b$, and

$$g(I) \subseteq I, \qquad x_0 : I, \qquad \exists q : \mathbb{R}, \quad 0 \le q < 1, \quad \forall t : I, \quad |g'(t)| \le q$$

Then $x_{n+1} = g(x_n)$ converges to the unique fixed point $r : I$, and

$$|x_{n+1} - r| \le q \cdot |x_n - r|, \qquad |x_{n+1} - r| \le \frac{q}{1-q} \cdot |x_{n+1} - x_n|$$

Thus a certified root-error tolerance $\varepsilon > 0$ is met when $\frac{q}{1-q} \cdot |x_{n+1} - x_n| \le \varepsilon$

Pf. Convergence Intuition.
Mean value theorem and completeness of the real numbers; real-analysis

- The derivative bound shrinks distances by at most a factor q
- Successive changes are bounded by a geometric sequence, so the iterates converge within the closed interval; continuity makes their limit a fixed point
- Summing the remaining changes gives the displayed root-error bound. Two distinct fixed points would contradict the distance bound

□

Convention 2.2 (Optimized Fixed Point Iteration)**.** [conv:optimized-fpi] Set a step tolerance tol > 0 and stop when $|x_{n+1} - x_n| <$ tol. This test alone does not certify a root error below tol; use the assumptions and bound in the fixed-point error analysis

1. Find a function $g(x)$ such that root r is a fixed point of $g(x)$, i.e., $g(r) = r$.
2. Choose an initial guess x_0.
3. Choose a maximum number of iterations N
4. Set an error tolerance, tol
5. Iterate $x_{n+1} = g(x_n)$
6. Check if $|x_{n+1} - x_n| <$ tol, if yes, stop the iteration and return the value.
7. If N updates are exhausted without meeting the stopping test, stop with an error

```
def find_fixed_point(
    g,                          # g(x): fixed point r satisfies g(r) = r
    x0: float,                  # initial guess
    tol: float = 1e-5,          # stop when |x_new - x| < tol
    max_iter: int = 100,        # safety cap on iterations
) -> float:
    x = x0
    for i in range(max_iter):
        x_new = g(x)
```

```
        print(f"Iteration {i+1}: x = {x_new}, g(x) = {g(x_new)}")
        if abs(x_new - x) < tol:
            print(f"Step tolerance met after {i+1} iterations.")
            return x_new
        x = x_new
    raise ValueError("Fixed point not found within max_iter.")
```

Example 2.1 (Convergent Case: Fixed Point Iteration on $x-\cos x=0$)**.** [eg:cos-fixed-point] Apply Optimized Fixed Point Iteration to $f(x) = x - \cos x = 0$.

1. Rewrite $f(x) = 0$ as $x = \cos x$, so $g(x) = \cos x$ and $g(r) = r$.
2. Initial guess: $x_0 = 1$.
3. Maximum iterations: $N = 100$.
4. Error tolerance: tol $\in \{5 \times 10^{-2},\ 1 \times 10^{-3}\}$ (two runs for comparison).
5. Iterate $x_{n+1} = \cos x_n$.
6. Stop when $|x_{n+1} - x_n| <$ tol and return x_{n+1}.
7. If N updates are exhausted without meeting the stopping test, stop with an error

In Python:

```
from math import cos
from find_fixed_point import find_fixed_point

# f(x) = x - cos(x)
# We want to solve f(x) = 0, i.e., find the root of f(x).
# So our equation is x - cos(x) = 0
# Which means that we can rewrite it as x = cos(x)
# So we want to find the fixed point of the function g(x) = cos(x)
def g(x):
    return cos(x)

# Initial guess: x = 1
x = 1

# Find the fixed point of g(x) = cos(x) using the fixed-point iteration method
r = find_fixed_point(g, x, tol=5e-2)
print(r)

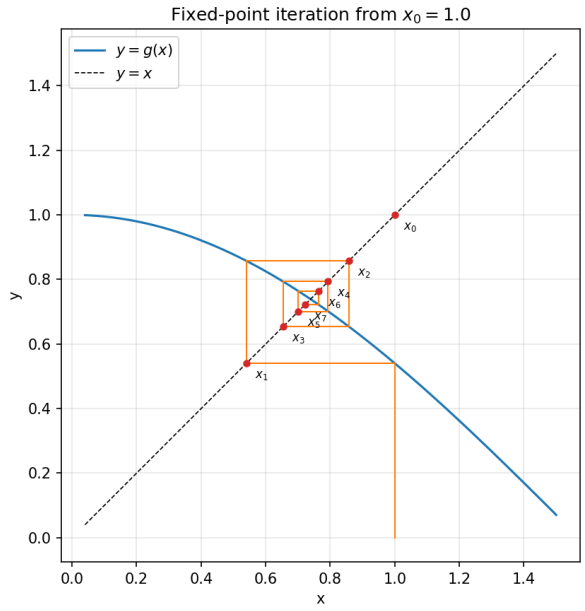
# Find the fixed point of g(x) = cos(x) using the fixed-point iteration method with a
    smaller tolerance
r = find_fixed_point(g, x, tol=1e-3)
print(r)
```

Both runs spiral into the fixed point $r \approx 0.739$; a tighter tolerance just adds more steps near r. The top row shows the iteration geometrically on the (x, y) plane; the bottom row shows the same iterates plotted against their step number.

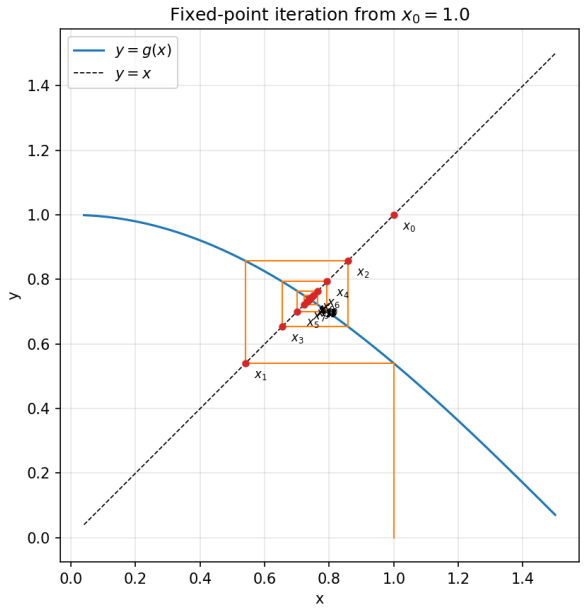
- Figure 2.1a: x-axis is x, y-axis is y; shows $y = g(x) = \cos x$ (blue), the diagonal $y = x$ (dashed), and an orange path that traces the iteration
- Figure 2.1b: same axes as (a); tighter tol $= 1 \times 10^{-3}$ makes the orange path pack more segments around the fixed point r.
- Figure 2.1c: x-axis is the iteration index n, y-axis is the iterate x_n; shows the sequence of iterates as damped oscillation for tol $= 5 \times 10^{-2}$, with the dashed line marking the final x_n.
- Figure 2.1d: same axes as (c); tighter run tol $= 1 \times 10^{-3}$ settles visibly onto $r \approx 0.739$.

Remark 2.2 (Convergence not Guaranteed). [rem:fixed-point-iteration-convergence] The convergence of fixed point iteration is not guaranteed.

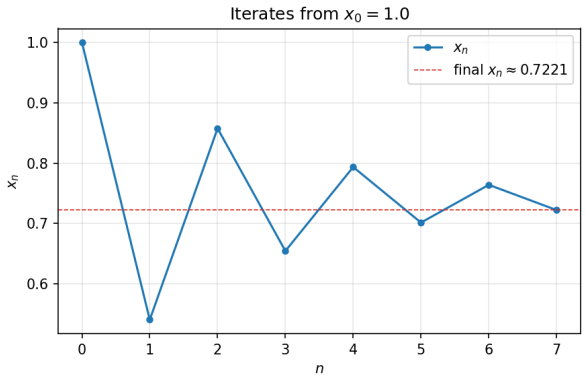
Example 2.2 (Divergent Case: $x - 1.5\cos x = 0$)**.** [eg:divergent-fpi] Same procedure as example 2.1, but with $g(x) = 1.5\cos x$ and $x_0 = 1$. The iterates do not converge to a fixed point; they



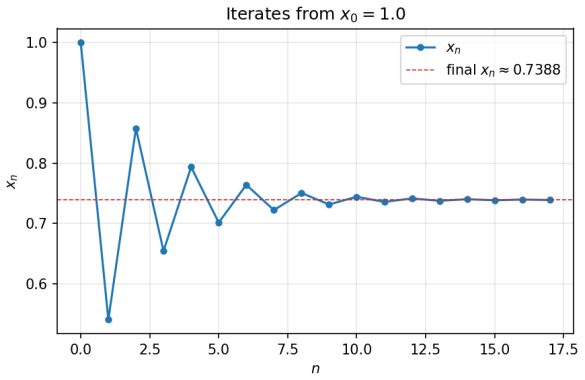
(a) Trace Iteration, tol $= 5 \times 10^{-2}$.



(b) Trace Iteration, tol $= 1 \times 10^{-3}$.



(c) x_n vs. n, tol $= 5 \times 10^{-2}$.



(d) x_n vs. n, tol $= 1 \times 10^{-3}$.

Figure 2.1: Convergent case: fixed point iteration on $x - \cos x = 0$ from $x_0 = 1$.

approach an alternating pair near 0.123 and 1.489. Both plotted runs reach $N = 100$ without satisfying the stopping test

Remark 2.3 (Advantage and Disadvantage of Fixed Point Iteration Method). [rem:fpi-evaluation]

1. Advantage: It is easy to implement.
2. Disadvantage: The convergence is not guaranteed.

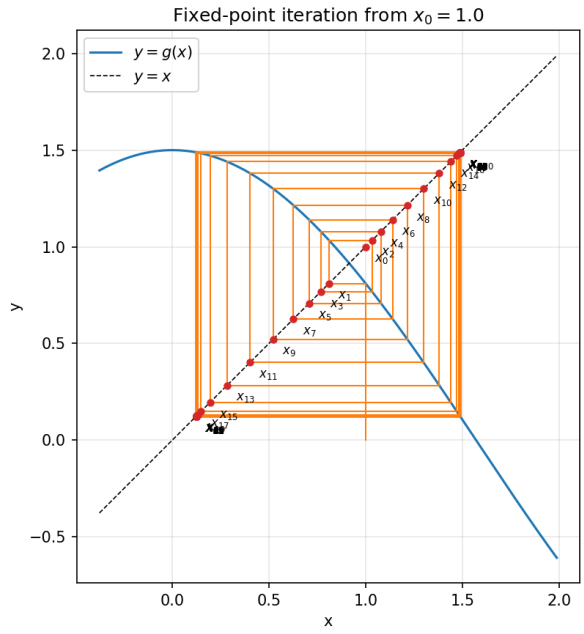
Usually used for *good initial guess* and *good* $g(x)$ *function*.

3 Bisector Method

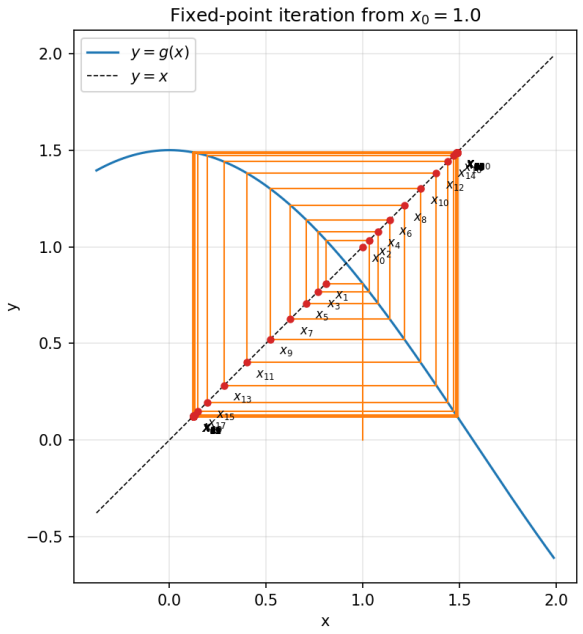
Definition 3.1 (Bisector Method)**.** [def:bisector-method] The idea of *bisector method* is when we know that a root r is in the interval $[a, b]$, then we can find the midpoint of the range and evaluate which part of the interval contains the root. We can then repeat this process until we find a sufficiently accurate approximation of the root.

Convention 3.1 (Procedure of Bisector Method)**.** [conv:bisector-method-procedure] Let f be continuous on $[a_0, b_0]$, where $a_0 < b_0$

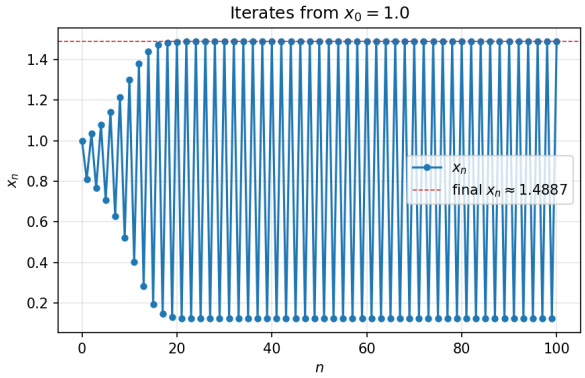
1. If either endpoint is an exact root, return it. Otherwise require $f(a_0) \cdot f(b_0) < 0$
2. Choose $N : \mathbb{N}$ halvings and initialize $a = a_0$, $b = b_0$



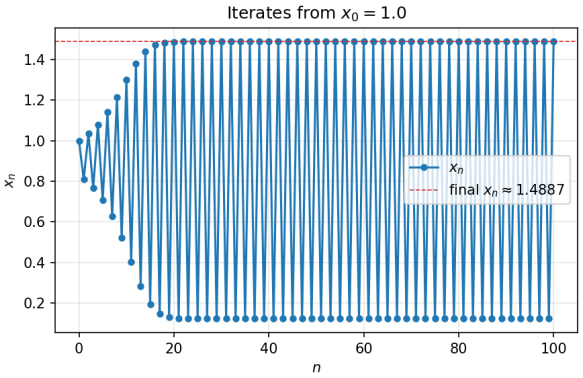
(a) Trace Iteration, tol $= 5 \times 10^{-2}$.



(b) Trace Iteration, tol $= 1 \times 10^{-3}$.



(c) x_n vs. n, tol $= 5 \times 10^{-2}$.



(d) x_n vs. n, tol $= 1 \times 10^{-3}$.

Figure 2.2: Divergent case: fixed point iteration on $x - 1.5\cos x = 0$ from $x_0 = 1$.

3. For each halving, compute $c = (a + b)/2$. If $f(c) = 0$, return c
4. Otherwise, if $f(a) \cdot f(c) < 0$, set $b = c$; if not, set $a = c$
5. Repeat until N halvings are complete, then return the midpoint $(a+b)/2$ of the final interval

Alternatively, stop once half the current interval width is at most the requested absolute error, and return its midpoint

Pf. Bisector Method Convergence. Continuity and the intermediate value theorem; real-analysis **TODO:** Use the intermediate value theorem and nested bisection intervals to prove convergence □

Remark 3.1 (Error Analysis of Bisector Method). `[rem:bisector-method-error]` Unless an exact root is found earlier, after k halvings the retained interval $[a_k, b_k]$ contains a root and has width

$$b_k - a_k = \frac{b_0 - a_0}{2^k}$$

Its midpoint $c_k = (a_k + b_k)/2$ therefore satisfies, for a root $r : [a_k, b_k]$,

$$|c_k - r| \leq \frac{b_k - a_k}{2} = \frac{b_0 - a_0}{2^{k+1}}$$

Returning an endpoint instead gives the weaker bound $b_k - a_k$. Choose k so that the appropriate bound is at most the requested tolerance; the width is an error bound, not the exact error

4 Newton-Raphson Method

Definition 4.1 (Newton-Raphson Method)**.** [def:newton-raphson-method] The idea of *Newton-Raphson method* is to use the slope of the function at a point to find the root of the function

Remark 4.1 (Intuition of Newton-Raphson Method). [rem:newton-raphson-intuition] At the current guess x_n, approximate f by its tangent line:

$$y = f'(x_n)\,(x - x_n) + f(x_n).$$

The next guess x_{n+1} is where this tangent line crosses zero. Setting $y = 0$ and solving for x:

$$0 = f'(x_n)\,(x - x_n) + f(x_n) \quad \Longrightarrow \quad x = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This gives the update rule used in step 4 below.

Convention 4.1 (Procedure of Newton-Raphson Method)**.** [conv:newton-raphson-method-procedure]

1. Choose an initial guess x_0, a tolerance $\text{tol} > 0$, and an iteration limit N
2. If $f(x_n) = 0$, return x_n
3. Evaluate $f'(x_n)$; if it is zero away from a root, stop with an error
4. Update:
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
5. If $f(x_{n+1}) = 0$ or $|x_{n+1} - x_n| < \text{tol}$, return x_{n+1}
6. Repeat from step 2; if N updates are exhausted, stop with an error

Reject non-finite function values, derivatives and iterates

Remark 4.2 (Newton-Raphson Error Analysis). [rem:newton-raphson-method-error] Let $f(r) = 0$, $f'(r) \neq 0$, and $e_n = x_n - r$. Assume twice continuous differentiability near r, real-analysis. A Taylor expansion with remainder through the linear term gives

$$0 = f(x_n) - f'(x_n) \cdot e_n + \frac{f''(\xi_n)}{2} \cdot e_n^2$$

for some ξ_n between x_n and r. Hence

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)} = \frac{f''(\xi_n)}{2f'(x_n)} \cdot e_n^2 \approx \frac{f''(r)}{2f'(r)} \cdot e_n^2$$

This is local quadratic convergence when the initial guess is sufficiently close to r. The stopping test $|x_{n+1} - x_n| < \text{tol}$ alone does not certify $|x_{n+1} - r| < \text{tol}$; also inspect the residual $|f(x_{n+1})|$

For valid finite inputs, the core iteration is:

```
def solve_newton_raphson(
    f,                      # function whose root we seek
    df,                     # derivative of f
    x0: float,              # initial guess
    tol: float = 1e-5,      # stop when |x_new - x| < tol
    max_iter: int = 100,    # safety cap on updates
) -> float:
    # Assume valid inputs and finite function values and iterates.
    x, fx = x0, f(x0)
```

```
    for i in range(max_iter):
        if fx == 0:
            return x
        dfx = df(x)
        if dfx == 0:
            raise ValueError("Zero derivative away from a root.")
        x_new = x - fx / dfx
        f_new = f(x_new)
        print(f"Iteration {i+1}: x = {x_new}, f(x) = {f_new}")
        if f_new == 0 or abs(x_new - x) < tol:
            print(f"Converged after {i+1} updates.")
            return x_new
        x, fx = x_new, f_new
    raise ValueError("Root not found within max_iter.")
```

The full implementation in `lec01/newton_raphson.py` also validates inputs and records the iteration history used by the plots

Example 4.1 (Newton-Raphson on $x^3 = \sin x$)**.** [eg:newton-a1-positive-root] *Source:* AP30019 Assignment 1, Question 2 (equation only). Apply the Newton-Raphson procedure to $x^3 - \sin x = 0$, starting from $x_0 = 1$ to target the positive root

$$\frac{d}{dx}(x^3 - \sin x) = 3x^2 - \cos x, \qquad x_{n+1} = x_n - \frac{x_n^3 - \sin x_n}{3x_n^2 - \cos x_n}$$

The equation also has the root 0; the initial guess matters

```
from math import cos, sin

from newton_raphson import solve_newton_raphson

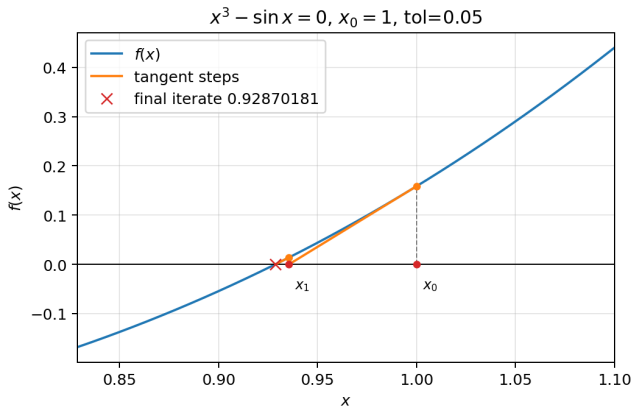

# Solve x**3 = sin(x), targeting the positive root from x0 = 1.
def f(x):
    return x**3 - sin(x)


def df(x):
    return 3 * x**2 - cos(x)

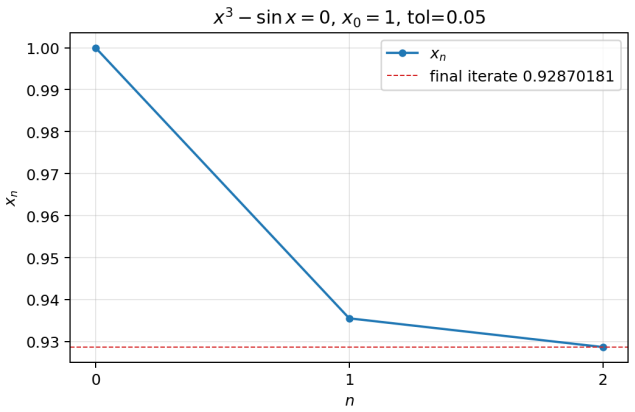

for tol in (5e-2, 1e-8):
    history = []
    root = solve_newton_raphson(f, df, x0=1.0, tol=tol, history=history)
    print(f"tol={tol:.0e}, x0=1, root={root:.12f}, "
          f"iterations={len(history) - 1}, residual={abs(f(root)):.3e}")
    print("iterates:", history)
```

tol	updates	final x	$\lvert f(x)\rvert$
5×10^{-2}	2	0.928701808804	1.501×10^{-4}
10^{-8}	4	0.928626308732	3.331×10^{-16}

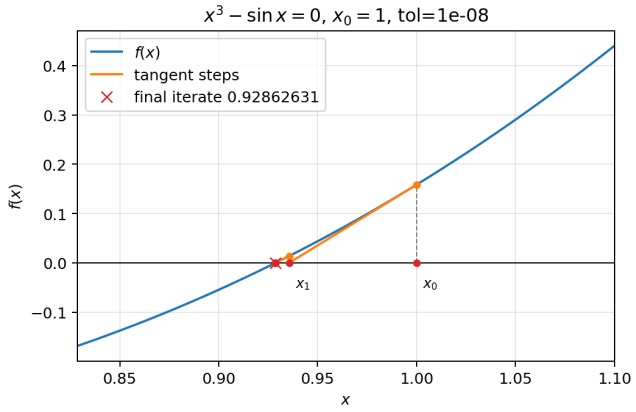
The tangent plots show the update geometrically; the sequence plots show x_n approaching the positive root. Run `python3 lec01/plot_newton_raphson_demo_1.py` to regenerate the figures



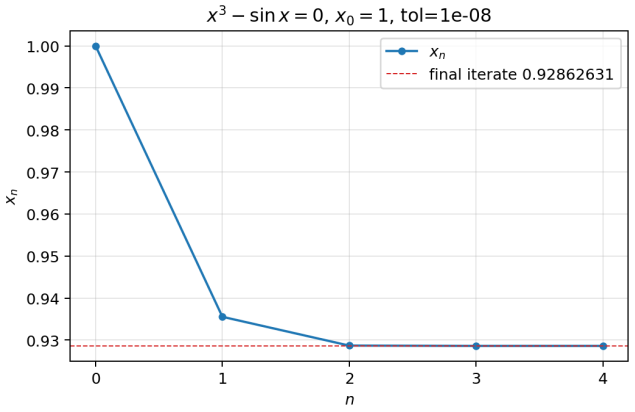
(a) Tangent steps, tol $= 5 \times 10^{-2}$



(b) Iterates, tol $= 5 \times 10^{-2}$



(c) Tangent steps, tol $= 10^{-8}$



(d) Iterates, tol $= 10^{-8}$

Figure 4.1: Newton-Raphson applied to $x^3 - \sin x = 0$ from $x_0 = 1$