Some Approximation Techniques in Python

Numerical Search Methods and Their Applications

CS 101 - Fall 2025

Course Overview

So, What are we talking about today?

- To understand exhaustive enumeration and iterative numerical techniques for approximation
- To study and understand how algorithms use approximations to find square and cube roots
- Compare efficiency of different approximation methods, and some of their limitations
- To explore primality testing using enumeration
- Implement practical Python solutions (code)

Part1: Exhaustive Method

Simple Exhaustive Square Roots

```
# Exhaustive square root
def simple_square_root(x):
    # find a root of a perfect root
    ans = 0
    while ans **2 < abs(x):
        ans += 1
    if ans**2 != abs(x):
        print(f"x = {x} is not perfect square root...")
    else:
        if x<0:
            ans =-ans
        print(f"Square root of {x} is {ans}.")
```

We count and then square the result to compare to the absolute value of the number to check.

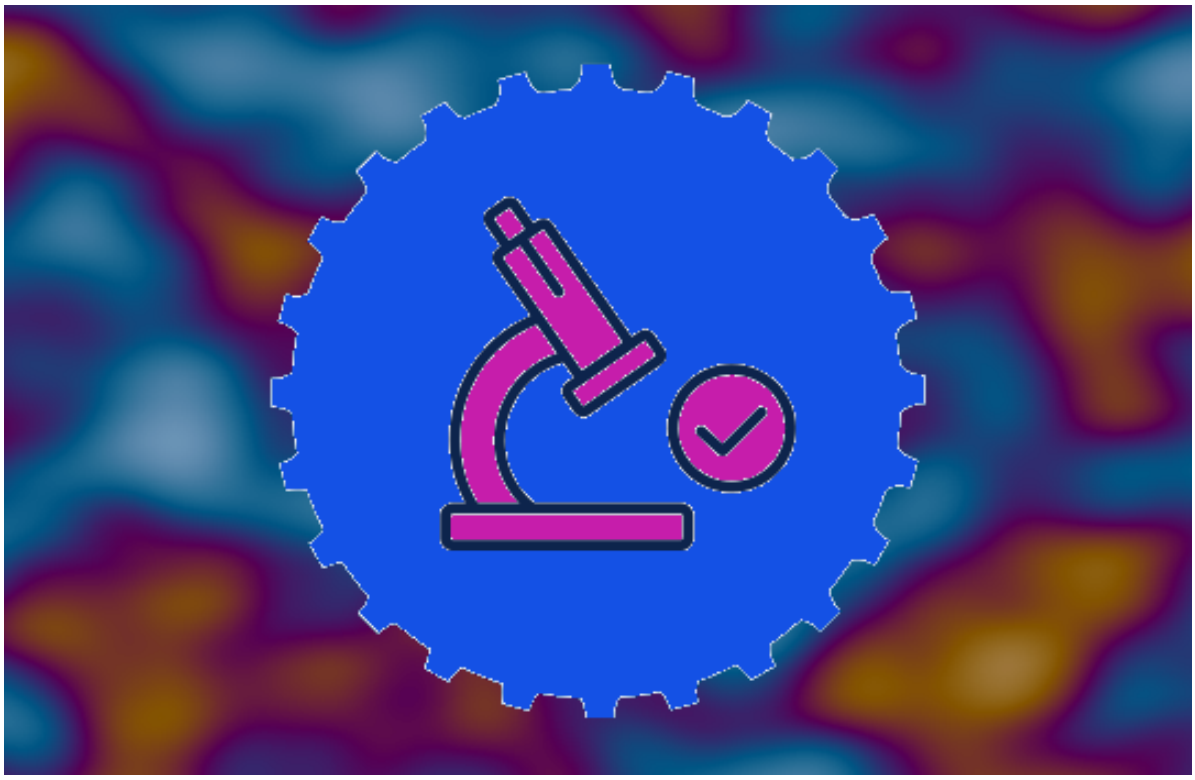
Testing the Prototype

```
simple_square_root(25) # find the square root of a square.
```

```
Square root of 25 is 5.
```

```
simple_square_root(26) # not a square ...
```

```
x = 26 is not perfect square root...
```



`simple_square_root(x)`

Count, square and check

Step-by-Step Process:

1. Start with `ans = 0`
2. Check if `ans`^{2} $<$ |x|
3. If true, increment `ans` by 1
4. Repeat until `ans`^{2} $\geq$ |x|
5. Test if `ans`^{2} exactly equals |x|

Key Logic: Exhaustive search: Tests every integer sequentially.

If `ans`^{2} == |x| then the root is found. Cool!

Binary outcome: Either finds exact root or reports failure

Example: For x = 25,

tests: 0^2, 1^2, 2^2, 3^2, 4^2, 5^2 = 25

Be careful!

Example: For x = 26,

tests: 0^2, 1^2, 2^2, 3^2, 4^2, 5^2, 6^2 != 26

(We just passed the correct value!!)

Simple Exhaustive Cube Roots (with same method)

```
# Exhaustive cube root
def simple_cube_root(x):
    # find a root of a perfect root
    ans = 0
    while ans **3 < abs(x):
        ans += 1
    if ans**3 != abs(x):
        print(f"x = {x} is not perfect cube root...")
    else:
        if x<0:
            ans =-ans
        print(f"Cube root of {x} is {ans}.")
```

Testing the Prototype

```
simple_cube_root(8) # has a cube root
```

```
Cube root of 8 is 2.
```

```
simple_cube_root(7) # does not have a cube ...
```

```
x = 7 is not perfect cube root...
```

`simple_cube_root(x)`

Count, cube and check

Step-by-Step Process:

(Same as before)

1. Start with `ans = 0`
2. Check if `ans^3` < |x|
3. If true, increment `ans` by 1
4. Repeat until `ans^3` |x|
5. Test if `ans^3` exactly equals |x|

Key Logic: Exhaustive search: Tests every integer sequentially.

If `ans^3` == |x| then the root is found. Nifty!

Binary outcome: Either finds exact root or reports failure

Example: For x = 8,

tests: 0^3, 1^3, 2^3 = 8

Be careful!

For x = 9, tests: 0^3, 1^3, 2^3, 3^3 != 9

(We just passed the correct value!!)

How to Generalize This?

Exhaustive nth root

```
def simple_n_root(x, n):
    # find a root of a perfect root
    ans = 0
    while ans **n < abs(x):
        ans += 1
    if ans**n != abs(x):
        print(f"x = {x} is not perfect {n} root...")
    else:
        if x<0:
            ans =-ans
        print(f"{n} root of {x} is {ans}.")
```

Testing the Prototype

```
simple_n_root(8,2) # find the square root
```

```
x = 8 is not perfect 2 root...
```

```
simple_n_root(17,2) # not a square ...
```

```
x = 17 is not perfect 2 root...
```

```
simple_n_root(9,3) # not a cube ...
```

```
x = 9 is not perfect 3 root...
```

Complicated Exhaustive Square Roots

Add print statements to see steps.

```
def exhaustive_sqrt(x, epsilon=0.01):
    """
    Find square root using exhaustive enumeration
    """
    step = epsilon
    num_guesses = 0
    ans = 0.0

    print(f"Finding square root of {x}")

    while abs(ans**2 - x) >= epsilon and ans*ans <= x:
        ans += step
        num_guesses += 1

    print(f"Number of guesses: {num_guesses}")

    if abs(ans**2 - x) >= epsilon:
        print(f"Failed to find square root of {x}")
        return None
    else:
        print(f"Square root of {x} is approximately {ans}")
        return ans
```

Testing the Prototype

```
exhaustive_sqrt(25) # find the square root of a square.
```

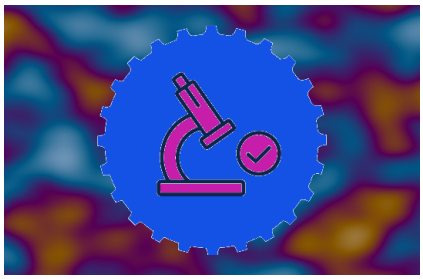
```
Finding square root of 25
Number of guesses: 500
Square root of 25 is approximately 4.999999999999938

4.999999999999938
```

```
exhaustive_sqrt(26) # Not a square ...
```

```
Finding square root of 26
Number of guesses: 510
Square root of 26 is approximately 5.099999999999936

5.099999999999936
```



How This Code Works:

Algorithm Steps: 1. Start with `ans = 0.0` 2. Increment by `epsilon` each iteration 3. Check if `ans`^{2} is close enough to `x` 4. Stop when within tolerance or exceeded target

Key Variables: - `step`: How much to increment each guess - `num_guesses`: Performance counter - `ans`: Current approximation

Loop Condition Explained: - `abs(ans**2 - x) >= epsilon`: Not accurate enough yet - `ans*ans <= x`: Haven't exceeded target (prevents infinite loop)

Why It Works: - Systematically tests every possible value - Guaranteed to find solution if it exists - Simple but inefficient for large numbers

---

The Fundamental Limitation

These functions use "exhaustive enumeration" over integers:

- **Perfect squares/cubes:** Have integer roots (4, 9, 16, 25…)
- **Non-perfect squares/cubes:** Have irrational/decimal roots

Key Insight: The algorithm design assumes the answer is an integer!

```
# This works:
simple_square_root (25) = 5
simple_square_root(27) = 3

# (only exact integers)
```

```
# This fails:
simple_square_root(26) = 5.099...
simple_cube_root(26) = 2.962...

# (not integers, no results found
```

Part2: Approximation

What if I need exact numbers for roots of a non-perfect value?

My number is …

- Not a square
- Not a cube
- Not an n^{th} value of anything!

How to find exact values of *any* number I want?!

What We Need Instead

For approximating non-perfect roots, we need:

1. **Decimal precision** (not just integers)
2. **Tolerance/epsilon** (how close is "close enough?")
3. **Different search strategies:**
    - Increment by small decimals (0.01, 0.001…)
    - Newton's method

Next: We'll explore these *approximation* techniques!

So, What is *Approximation*?

Key Concepts:

- Finding "good enough" solutions
- Trading precision for efficiency
- Iterative refinement
- Stopping criteria (stop the approximation by setting precision)

Like, Why Approximation?

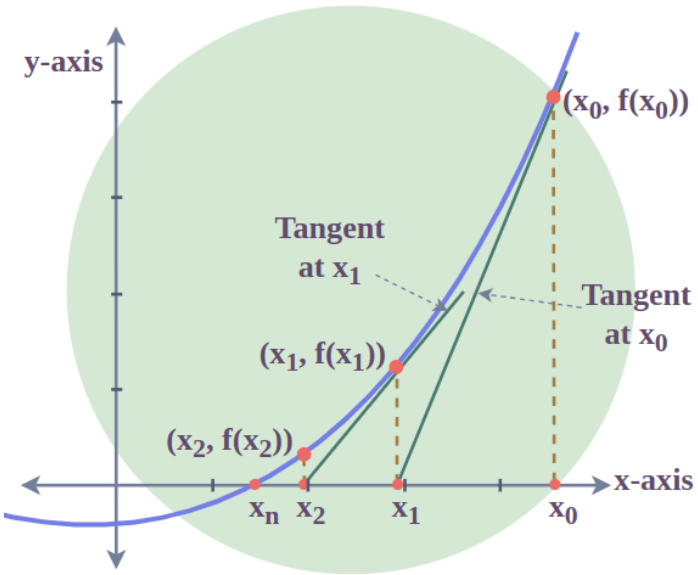
- Exact solutions may not exist (not a perfect number)
- Real-world applications (like how your computer does this root-finding!)

Issac Newton

Whoops, here's the right slide.

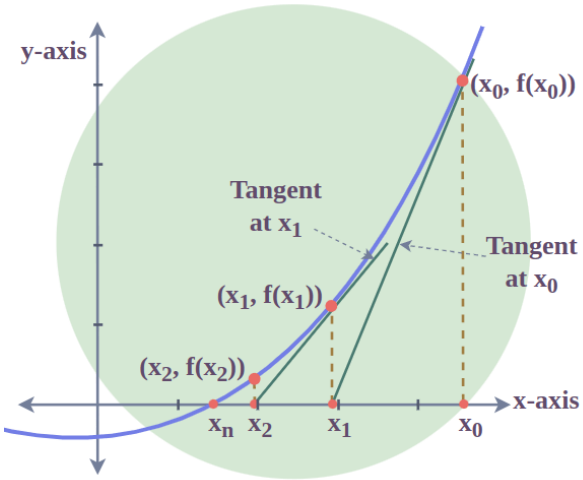
(Issac Newton)

Square Root Approximation with Newton's Method



```
def newtons_sqrt(n:float, guess:float = 1.0) -> float:
    while abs(n - guess*guess) > .0001:
        print(f"n = {n}, guess = {guess}")
        print(f"   abs(n - guess*guess) = {abs(n - guess*guess)}")
        guess = guess - (guess*guess - n)/(2*guess)
        print(f"   guess = guess - (guess*guess - n)/(2*guess) = {guess}\n")

    return guess
```



With debug print statements ...

```
# Test the function
my_num = 25
result = newtons_sqrt(my_num)
print(f"Verification:\n {result}^2 = {result**2} is {result**2 == result**2}")
```

```
n = 25, guess = 1.0
   abs(n - guess*guess) = 24.0
   guess = guess - (guess*guess - n)/(2*guess) = 13.0

n = 25, guess = 13.0
   abs(n - guess*guess) = 144.0
   guess = guess - (guess*guess - n)/(2*guess) = 7.461538461538462

n = 25, guess = 7.461538461538462
   abs(n - guess*guess) = 30.674556213017752
   guess = guess - (guess*guess - n)/(2*guess) = 5.406026962727994

n = 25, guess = 5.406026962727994
   abs(n - guess*guess) = 4.22512752174206
   guess = guess - (guess*guess - n)/(2*guess) = 5.015247601944898

n = 25, guess = 5.015247601944898
```

```
   abs(n - guess*guess) = 0.15270850881405096
   guess = guess - (guess*guess - n)/(2*guess) = 5.000023178253949

n = 25, guess = 5.000023178253949
   abs(n - guess*guess) = 0.00023178307672111487
   guess = guess - (guess*guess - n)/(2*guess) = 5.000000000053723

Verification:
 5.000000000053723^2 = 25.00000000053723 is True
```

So, how does this thing work?

- Each approximation of x is from Equation: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
  - We approach the actual value with each iteration
- The $\frac{f(x_n)}{f'(x_n)}$ term allows for us to get closer to the actual number with each approximation.
  - x_0 to x_1 to x_2 to, ..., to x_n

Algorithm Steps: 1. Start with `ans = 0.0` 2. Increment by `epsilon` each iteration 3. Check if `ans`^{2} is close enough to `x` 4. Stop when within tolerance or exceeded target

Key Variables: - `step`: How much to increment each guess - `num_guesses`: Performance counter - `ans`: Current approximation

Loop Condition Explained: - `abs(ans**2 - x) >= epsilon`: Not accurate enough yet - `ans*ans <= x`: Haven't exceeded target (prevents infinite loop)

Why It Works: - Systematically tests every possible value - Guaranteed to find solution if it exists - Simple but inefficient for large numbers

---

Cube Roots

```python
# cube root
def newtons_cube_root(n:float, guess:float = 1.0) -> float:
    while abs(n - guess*guess*guess) > .0001:
        print(f"n = {n}, guess = {guess}")
        print(f"   abs(n - guess^3) = {abs(n - guess*guess*guess)}")
        guess = guess - (guess*guess*guess - n)/(3*(guess*guess))
        print(f"   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = {guess}\n")
    return guess
```

Testing the Prototype

```python
newtons_cube_root(8)
```

```
n = 8, guess = 1.0
   abs(n - guess^3) = 7.0
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 3.3333333333333335

n = 8, guess = 3.3333333333333335
   abs(n - guess^3) = 29.037037037037045
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 2.462222222222222

n = 8, guess = 2.462222222222222
   abs(n - guess^3) = 6.92731645541838
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 2.081341247671579

n = 8, guess = 2.081341247671579
   abs(n - guess^3) = 1.0163315496105625
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 2.003137499141287

n = 8, guess = 2.003137499141287
   abs(n - guess^3) = 0.03770908398584538
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 2.000004911675504

2.000004911675504
```

Testing the Prototype

```
newtons_cube_root(27)
```

```
n = 27, guess = 1.0
   abs(n - guess^3) = 26.0
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 9.666666666666666

n = 27, guess = 9.666666666666666
   abs(n - guess^3) = 876.2962962962961
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 6.540758356453956

n = 27, guess = 6.540758356453956
   abs(n - guess^3) = 252.82358364070478
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 4.570876778578707

n = 27, guess = 4.570876778578707
   abs(n - guess^3) = 68.49893783892396
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 3.4780192333867963

n = 27, guess = 3.4780192333867963
   abs(n - guess^3) = 15.07226932492673
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 3.0626891086275365

n = 27, guess = 3.0626891086275365
   abs(n - guess^3) = 1.7282216154620045
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 3.001274406506175

n = 27, guess = 3.001274406506175
   abs(n - guess^3) = 0.03442359474399126
   guess = guess - (guess*guess*guess - n)/(3*(guess*guess)) = 3.0000005410641766

3.0000005410641766
```

Challenge!

Can you modify the `newtons_cube_root()` algorithm to find the forth root `newtons_cube_root()`


THINK