Numerical Experiments

Chapter 3 stuff

Finding the integer cube root

- How would you go about solving this?

- How can a computer help?

Finding the integer cube root

```
>>> def find_int_cube_root (x):
        ans = 0
        while ans**3 < abs(x):
                ans = ans + 1
        if ans**3 != abs(x):
                return x, 'is not a perfect cube'
        else:
                if x < 0:
                        ans = -ans
                return ans
```

Finding the integer cube root

```
>>> def find_int_cube_root(x):
        ans = 0
        while ans**3 < abs(x):
            ans = ans + 1
        if ans**3 != abs(x):
            return x, 'is not a perfect cube'
        else:
            if x < 0:
                ans = -ans
            return ans
```

For which integers does this terminate?

Finding the integer cube root

```
>>> def find_int_cube_root (x):
        ans = 0
        while ans**3 < abs(x):
                ans = ans + 1
        if ans**3 != abs(x):
                return x, 'is not a perfect cube'
        else:
                if x < 0:
                        ans = -ans
                return ans
```

For which integers does this terminate?

All of them!!

Finding the integer cube root

```
>>> def find_int_cube_root (x):
        ans = 0
        while ans**3 < abs(x):
            ans = ans + 1
        if ans**3 != abs(x):
            return x, ‘is not a perfect cube’
        else:
            if x < 0:
                ans = -ans
            return ans
```

Why?

Finding the integer cube root

```
>>> def find_int_cube_root (x):
        ans = 0
        while ans**3 < abs(x):
            ans = ans + 1
        if ans**3 != abs(x):
            return x, 'is not a perfect cube'
        else:
            if x < 0:
                ans = -ans
            return ans
```

Why?

Because of the decrementing function:

abs(x)-ans**3

gets smaller after each iteration.

Finding the integer cube root

```
>>> def find_int_cube_root (x):
        ans = 0
        while ans**3 < abs(x):
            print(abs(x)-ans**3)
            ans = ans + 1
        if ans**3 != abs(x):
            return x, 'is not a perfect cube'
        else:
            if x < 0:
                ans = -ans
            return ans
```

Why?

Because of the decrementing function:

abs(x)-ans**3

gets smaller after each iteration.

Exhaustive enumeration

Exhaustive enumeration

Guess and check:

You go through all possibilities until you get the right answer or you exhaust the list of possibilities.

Exhaustive enumeration

Guess and check:

You go through all possibilities until you get the right answer or you exhaust the list of possibilities.

What if we want to know all perfect cubes between 1 million and 2 million?

Exhaustive enumeration

Guess and check:

You go through all possibilities until you get the right answer or you exhaust the list of possibilities.

What if we want to know all perfect cubes between 1 million and 2 million?

```
>>> for i in range(1000001,2000001):
        if type(find_int_cube_root(i)) is int:
            print i
```

Exhaustive Approximation

- We have seen how to find integer cube roots. Sometimes we would like to find approximations of cube roots.
- For example, the cube root of 5 $\sqrt[3]{5} \approx 1.709976\ldots.$
- How would you go about finding this value. (or a value close to this.)
- Would using the speed and power of a computer help?

Exhaustive Approximation

- How about we start at x = 0 and increment by a small value, say 0.1
- Then we check if x**3 < input.

Exhaustive Approximation

- How about we start at x = 0 and increment by a small value, say 0.1
- Then we check if x**3 < input.
- >>>def cube_root_approx(inp):

```
x = 0
while x**3 < inp:
    x = x+ 0.1
return x
```

Using float numbers

- set x = 0.1
- Decimal numbers are called float.
- What does python do when you have a float?

Using float numbers

- set x = 0.1
- Python uses binary and you cannot express 1/10 exactly in binary. (just like you can't express 1/3 exactly in decimal.)
- 1/3 = 0.3333333…. in decimal
- 1/10 = 0.0001100110011001100…..=0.10000000000000000555111….

Using float numbers

- set x = 0.1
- Python uses binary and you cannot express 1/10 exactly in binary. (just like you can’t express 1/3 exactly in decimal.)

- So python may give you a crazy looking number.
- You can use the round(number, number of digits) function to round the number.

Exhaustive Approximation

- How long does this take?

- If I start at 0 and increment by 0.1 then it will take
- $10 * \sqrt[3]{n}$ many steps.

- Is there a better way?

Exhaustive Approximation

- What if we make a guess and check to see if the guess is greater or less than the target value.

- to find the cube root of n, what may be a good first guess?

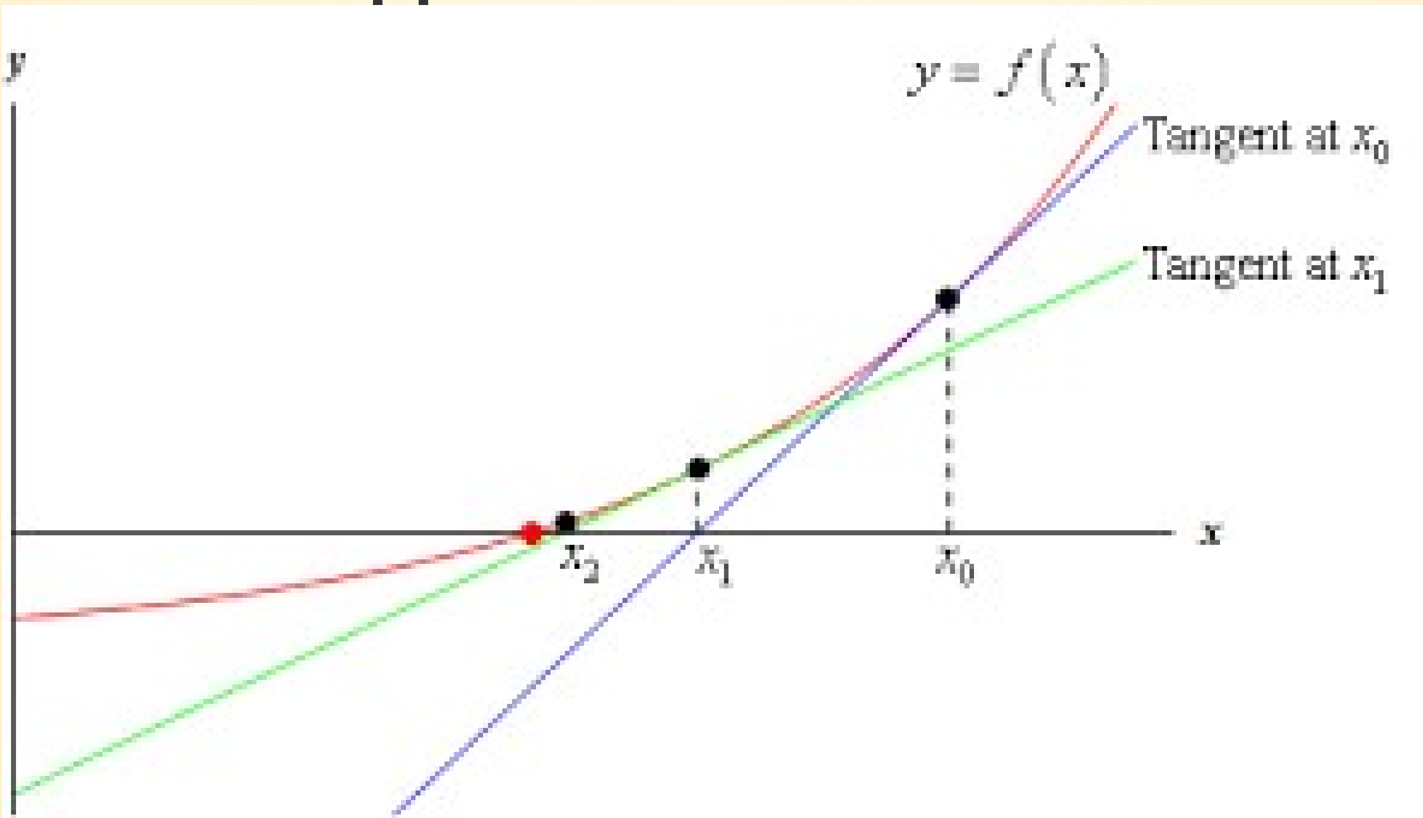
Exhaustive Approximation

- Start with n/2.
- Calculate (n/2)**3. If it is greater than n then we know that the 0<ans<n/2 and we can try again with n/4
- If it is less then we know that n/2<ans<n and we can try again with 3n/4

Exhaustive Approximation

```
def cube_root_approx_2(n,epsilon):
    numGuesses = 0
    low = 0.0
    high = max(1.0,n)
    ans = (low + high)/2.0
    while abs(ans**3 - n)> epsilon:
        print low,high,ans
        numGuesses += 1
        if ans**3<n:
            low = ans
        else:
            high = ans
        ans = (low + high)/2.0
    return ans
```

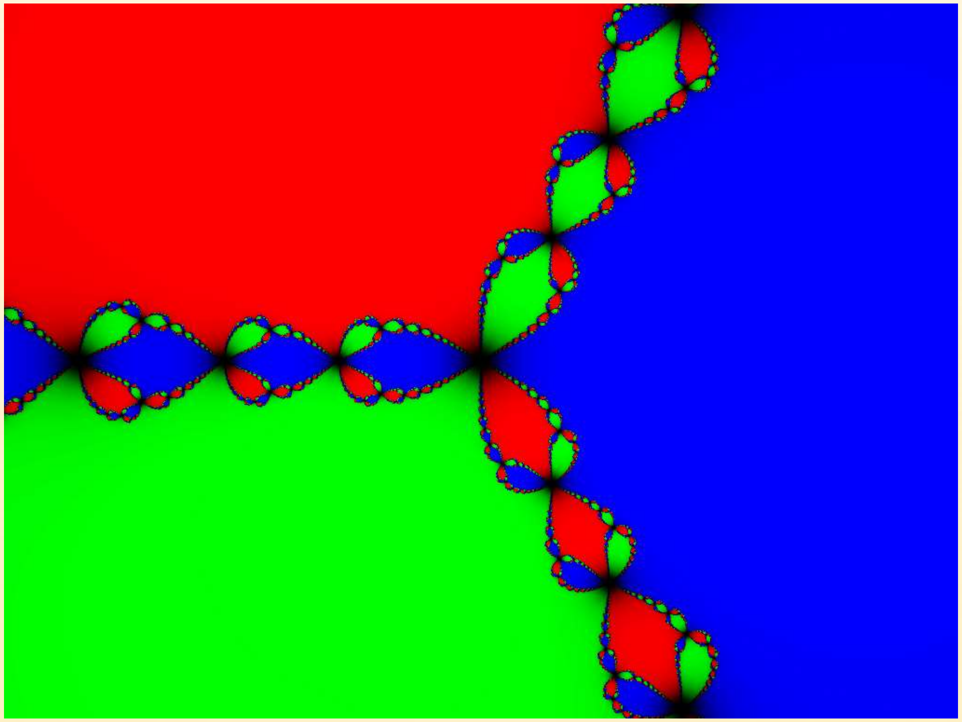
Exhaustive Approximation



Newton’s method (f is a function of z)

- def newton_method(f,guess,epsilon):
- z = Symbol('z')
- k = guess*1.0
- while abs(f.subs(z,k).evalf())>epsilon:
- fprime = f.diff(z).evalf()
- k = (k - 1.0*f.subs(z,k)/fprime.subs(z,k)).evalf()
- return k

Newton's method (f is a function of z)



Exhaustive enumeration

```
>>> def no_doub_ones(binarylist):
        c = 0
        n = len(binarylist)
        for b in binarylist:
                L = len(b)
                j = 0
                while j < L-1:
                        if b[j]=='1' and b[j+1]=='1':
                                c = c+1
                                j = L-1
                        else:
                                j = j+1
        return n-c
```

```
>>> def list_of_bin(n):
        X = []
        for i in range(2**n-1):
                X.append(bin(i)[2:])
        return X
```