

Iteration and Iterables

Phys 281 – Class 05

Grant Wilson

Answers to Exercises

All Exercises have the same header info:

```
from matplotlib import pyplot as plt  
import csv  
import numpy as np  
plt.ion()
```

E4.1

#E4.1 - read in a bunch of normally distributed data and plot it

datafile = "E1.npz"

npz = np.load(datafile)

x1 = npz['x1']

x2 = npz['x2']

plt.plot(x1,'b',label='x1')

plt.plot(x2,'r',label='x2')

plt.xlabel('index')

plt.ylabel('x1 and x2')

plt.legend(loc='upper left')

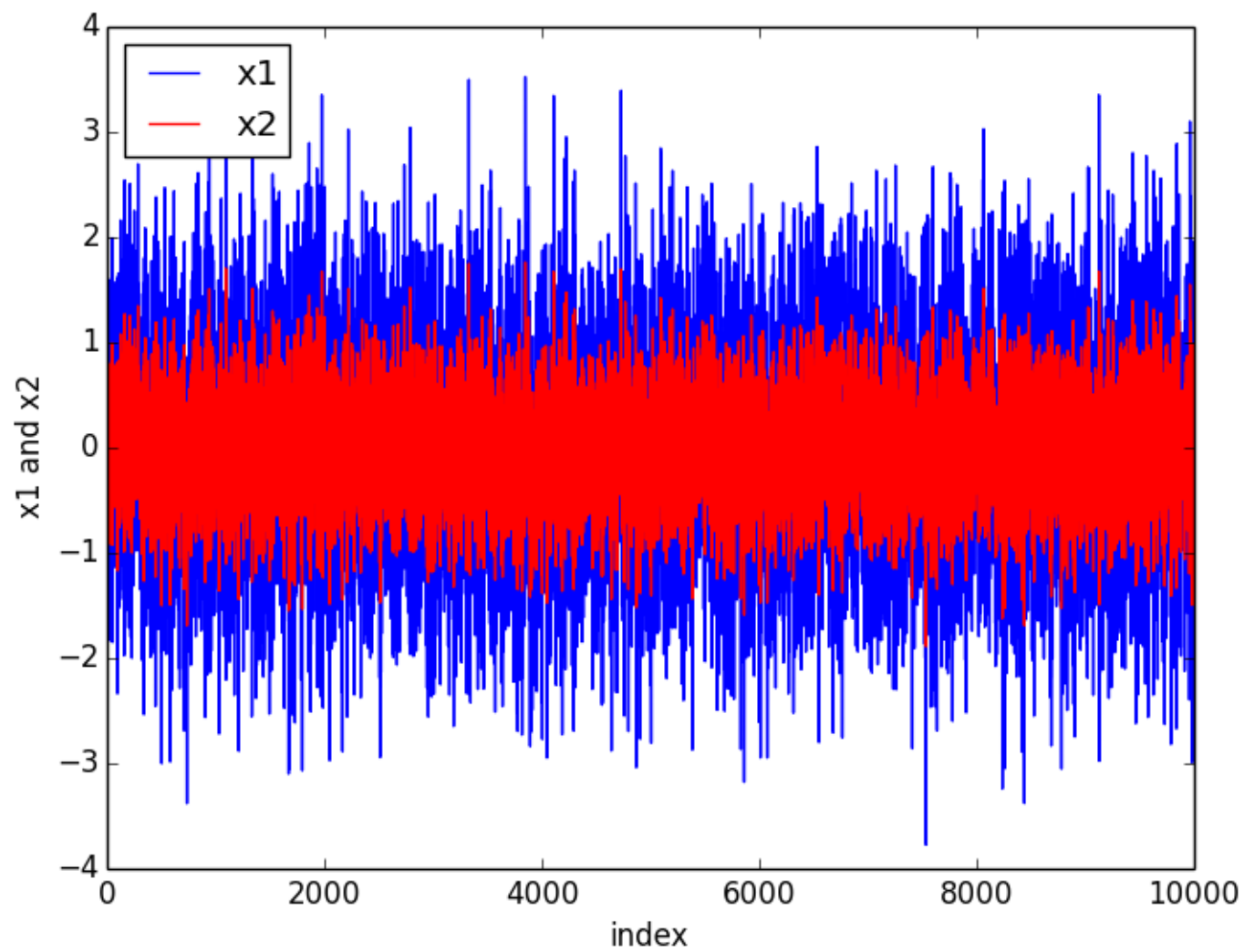
plt.show()

tmp = raw_input("Plot for E4.1: press Enter to continue.")

del tmp

plt.close()

plt.clf()



E4.2

#E4.2 - now make a histogram of the data

hist_x1, edges1 = np.histogram(x1,bins=40)

hist_x2, edges2 = np.histogram(x2,bins=40)

#set up the histogram bin parameters

width1 = 0.75(edges1[1]-edges1[0])*

width2 = 0.75(edges2[1]-edges2[0])*

center1 = (edges1[:len(edges1)-1]+edges1[1:])/2.

center2 = (edges2[:len(edges2)-1]+edges2[1:])/2.

#make the histogram plot using plt.bar()

plt.bar(center1, hist_x1, align='center', width=width1, label='x1')

plt.bar(center2, hist_x2, align='center', width=width2, color='y', label='x2')

plt.xlabel('x1 or x2 value')

plt.ylabel('# of samples')

plt.legend(loc = 'upper left')

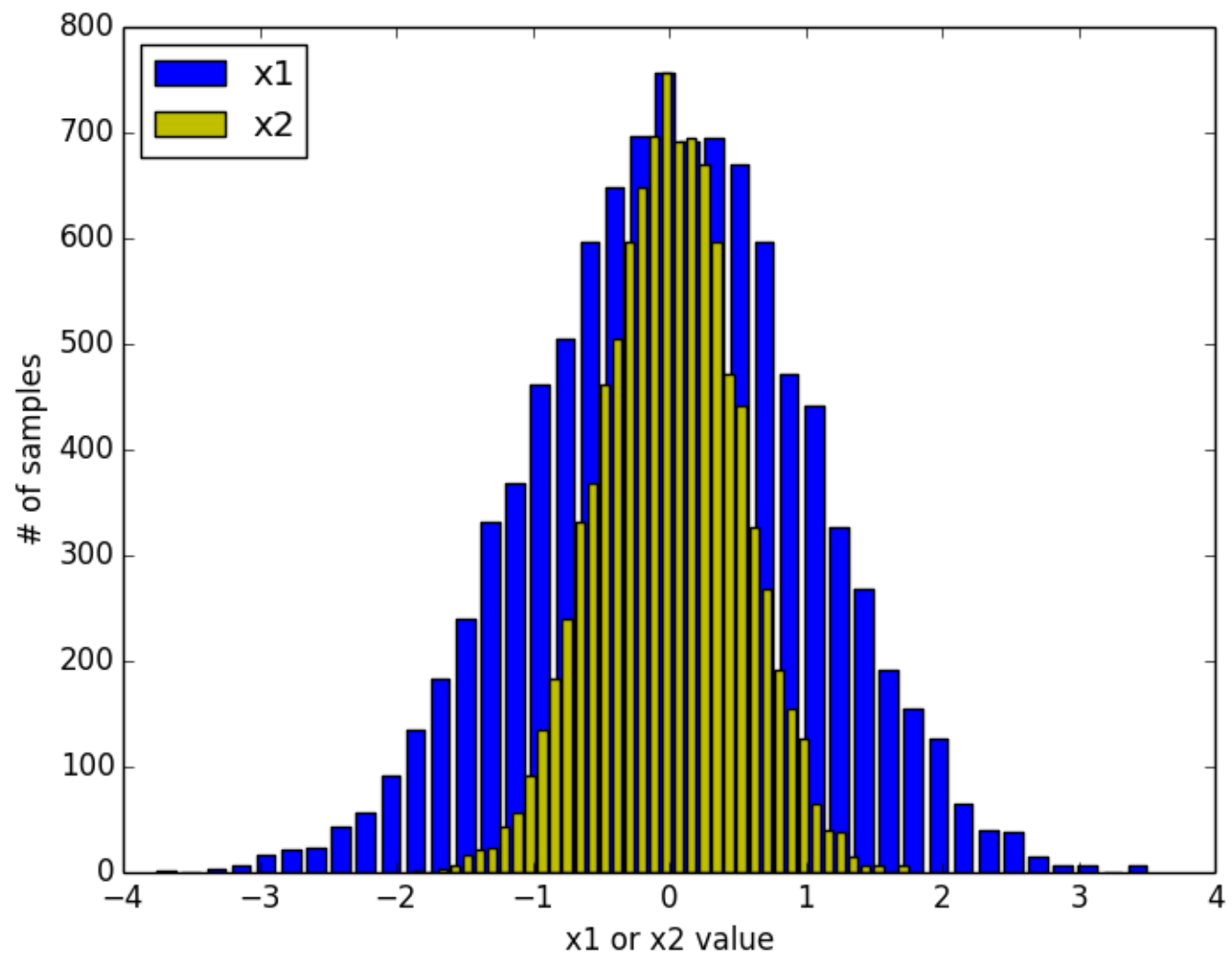
plt.show()

tmp = raw_input("Plot for E4.2: press Enter to continue.")

del tmp

plt.close()

plt.clf()



E4.3

#E4.3 - extract data from a csv file

datafile = 'my_first.csv'

#make storage for the values as lists

#since we don't know the length

xlist = []

zlist = []

#use the with-as construct to open

#the file so that we know it will

#be closed properly even if there is an

#error

with open(datafile) as csvfile:

csvreader = csv.reader(csvfile)

#loop through the rows

for row in csvreader:

xlist.append(row[3])

zlist.append(row[5])

#get rid of the leading column name

#strings

xlist.remove(xlist[0])

zlist.remove(zlist[0])

#now pack these into numpy arrays

x = np.empty(len(xlist),dtype='float')

z = np.empty(len(zlist),dtype='float')

for i in range(len(xlist)):

x[i] = float(xlist[i])

z[i] = float(zlist[i])

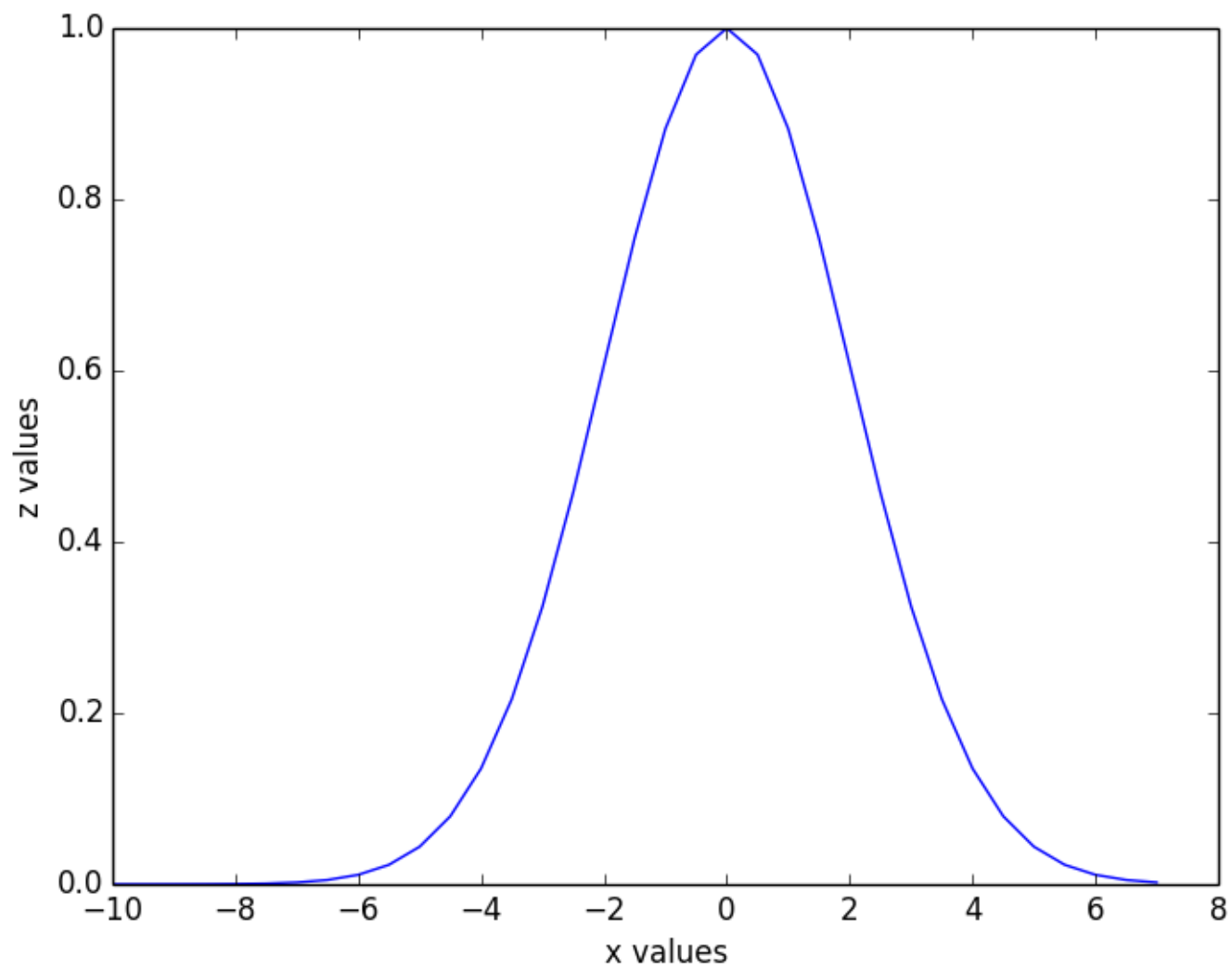
#finally, the plots

plt.plot(x,z)

plt.xlabel('x values')

plt.ylabel('z values')

plt.show()



Iteration and Iterables

Phys 281 – Class 05

Grant Wilson

“The purpose of computing is insight, not numbers.”

- Richard Hamming

Definitions

There is a large class of numerical algorithms that repeat simple (or complex) operations in order to converge to a solution.

– Iteration

- Definition of a function or sequence through repetition of a process.
- Solution technique involving successively better approximations.

– Recursion

- A special case of iteration wherein a function or sequence is defined by calling itself.

– Convergence

- stop changing, come to a stable value under repeated iterations

General Setup for Iteration

1. Create any needed storage
2. Initialize all your constants
3. Perform your loop
4. Check your results

Exercise: Computing a Sum

- Write a script using iteration to compute the sum of the numbers from 1 to 10
 1. Create any needed storage
 2. Initialize any constants
 3. Perform your loop
 4. Check your results

$$\sum_{i=1}^{10} i$$

Computing a Sum

```
import numpy as np

#create some storage for the answer
#and initialize it.
sum = 0.

#this is the iteration
for i in np.arange(1,11):
    sum = sum + i

#check the answer by doing it another way
vec = np.arange(1,11)
print vec.sum()
print sum
```

Example #1 – The Fibonacci Sequence

- A famous number sequence derived from the simple recursion relation:
 - $F_n = F_{n-1} + F_{n-2}$
- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
- Features:
 - Natural mechanical phenomena (petals on flowers)
 - F_n/F_{n-1} approaches the Golden Ratio as $n \rightarrow \infty$

Fibonacci Sequence

Iterative Approach

- Write a simple script to fill a 20 element array *f* with the Fibonacci sequence. You will need to initialize the first two elements of *f* to get the sequence going.

```
import numpy as np
```

```
f = np.zeros(20)
```

```
f[0] = 0
```

```
f[1] = 1
```

```
...
```

```
print f
```


Fibonacci Sequence

Iterative Approach

- Write a simple script to fill a 20 element array *f* with the Fibonacci sequence. You will need to initialize the first two elements of *f* to get the sequence going.

```
import numpy as np
```

```
f = np.zeros(20)
```


```
f[0] = 0
```

```
f[1] = 1
```

```
...
```

```
print f
```

Use a for or while loop to fill in the missing code.



Fibonacci Sequence

Iterative Approach

```
import numpy as np

#setup
npts = 20
f = np.zeros(npts)
f[0] = 0
f[1] = 1
index = np.arange(2,npts)

#the iteration
for i in index:
    f[i] = f[i-1]+f[i-2]

print f
```

Fibonacci Sequence

Recursive Approach

A recursive function is a function that calls itself.

```
import numpy as np

#a recursive function to compute the
# n-th element of the fibonacci sequence
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Fibonacci Sequence

Recursive Approach

A recursive function is a function that calls itself.

```
import numpy as np

#a recursive function to compute the
# n-th element of the fibonacci sequence
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

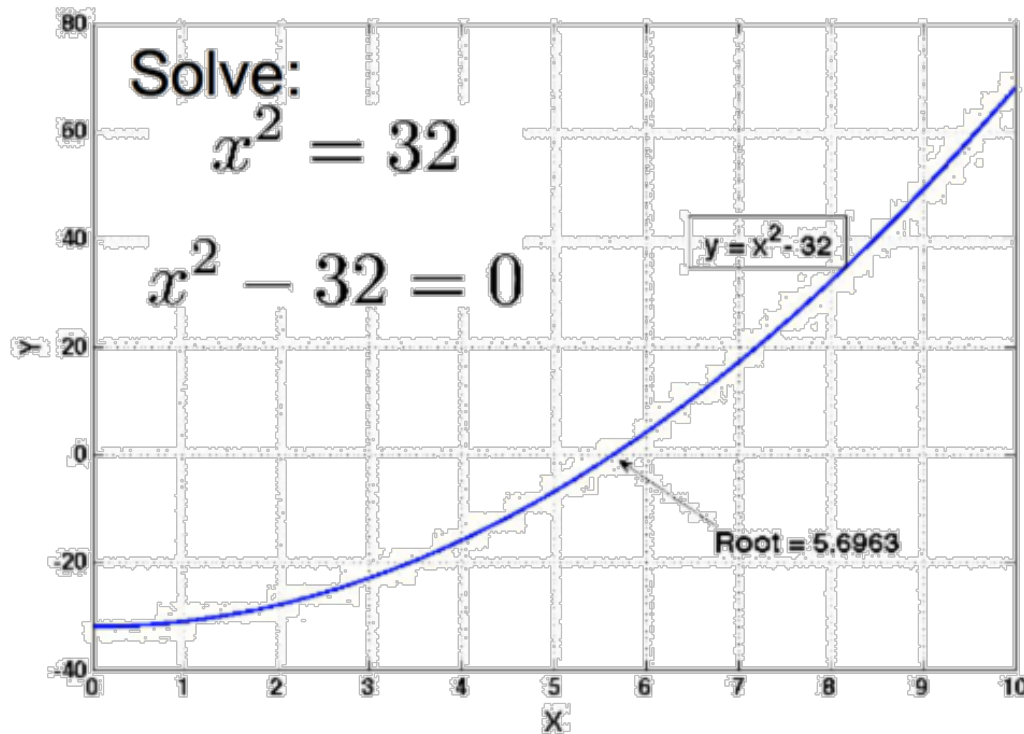
Try this out with $n=2$
and mentally trace the
steps that the code goes
through. Now do the
same with $n=4$.

Example #2 – Find the square root of 32

- What are some strategies we could use?

Example #2 – Find the square root of 32

Our Approach – root finding:



An Aside on Root Finding

Almost all equations can be written in the form

$$f(x) = 0 \quad - \text{ in one dimension} \quad (1)$$

or

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad - \text{ in } N \text{ dimensions} \quad (2)$$

Solving 1-d equations of the form (1) is equivalent to finding the “roots” of the function $f(x)$. Having the machinery in hand to do this is very powerful.

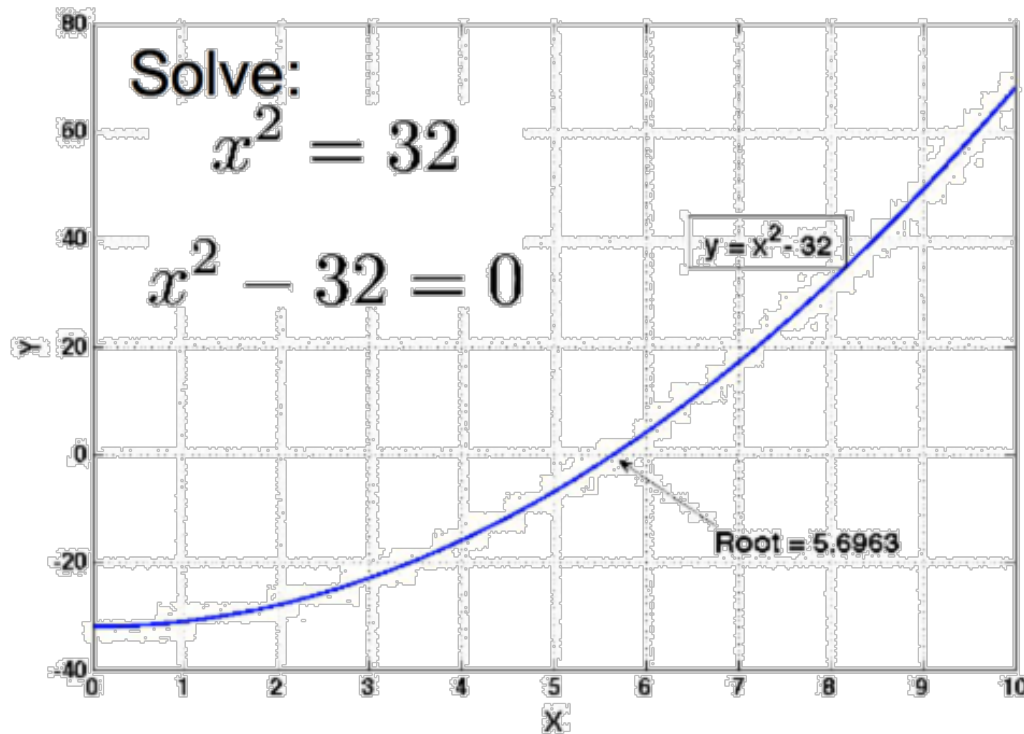
Note: We'll come back to solving simultaneous equations of the form (2) later in the semester.

One-dimensional Root Finding

- Strategy
 1. get an idea of what the function looks like by plotting it.
 2. bracket the root you are looking for
 3. use some iterative approach to zero-in on the root to your tolerance.

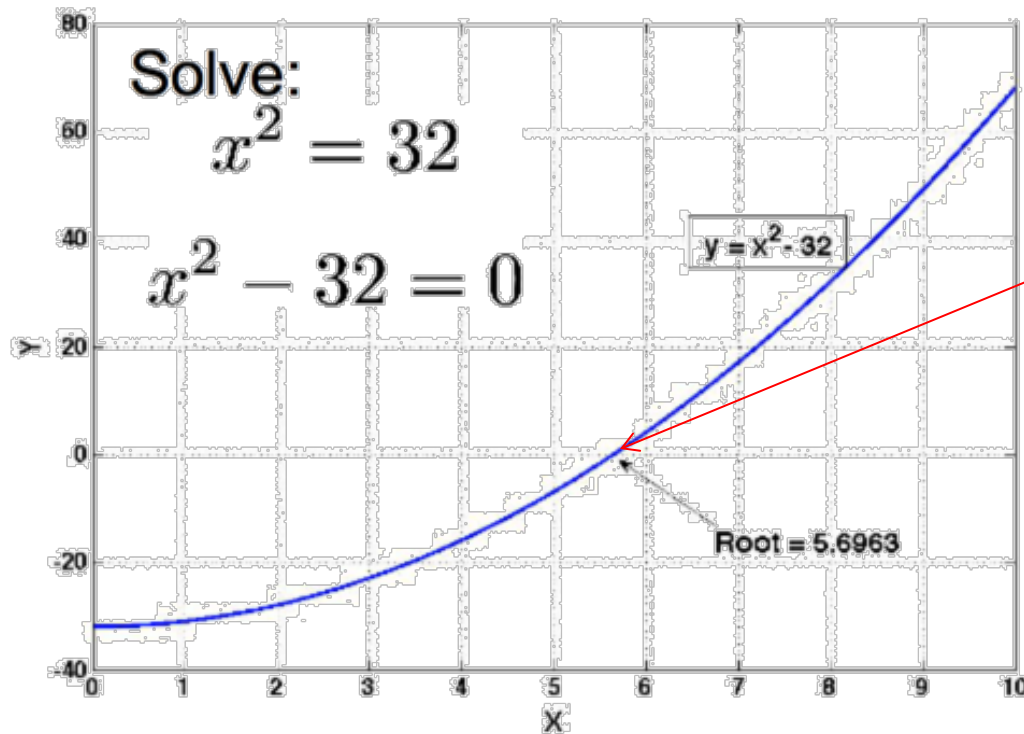
Example #2 – Find the square root of 32

Our Approach – root finding:



Example #2 – Find the square root of 32

Our Approach – root finding:



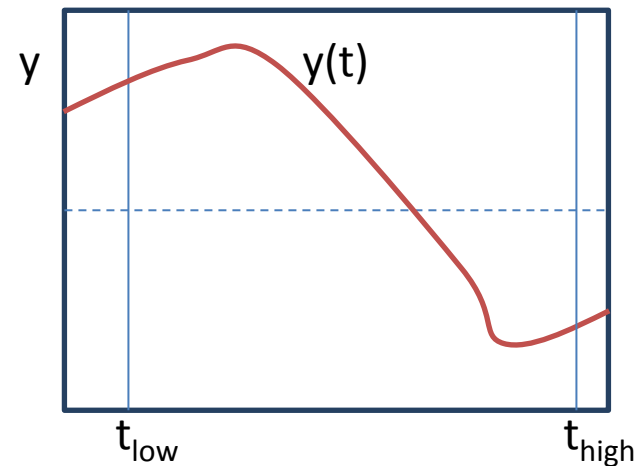
Define a function that is zero at our point of interest.

A place where a function is 0 is called a “root” of that function.

Iterative Searches for Roots

Method 1: Bisection Algorithm

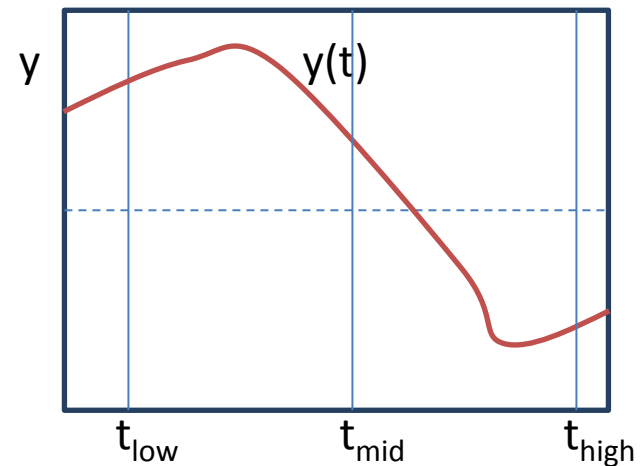
- Find value of t where $y(t)=0$
- if $y(t_{\text{low}})$ and $y(t_{\text{high}})$ have opposite signs then a root exists in the interval.
- Compute the midpoint of the interval $t_{\text{mid}} = (t_{\text{high}} + t_{\text{low}})/2$. and determine which half has the root.
- Repeat the search until the interval has narrowed to give an answer that is “close enough”



Iterative Searches for Roots

Bisector Algorithm

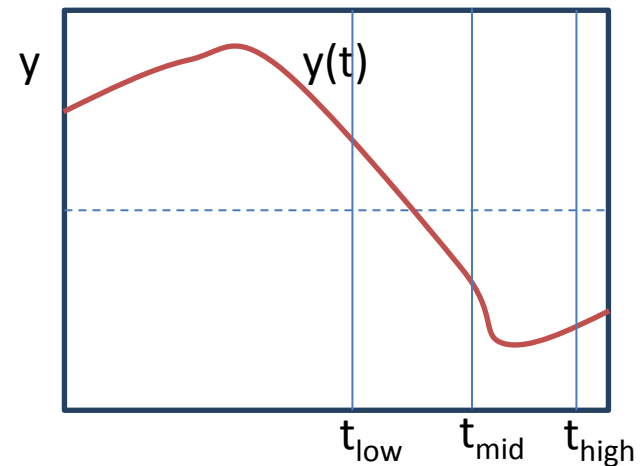
- Find value of t where $y(t)=0$
- if $y(t_{\text{low}})$ and $y(t_{\text{high}})$ have opposite signs then a root exists in the interval.
- Compute the midpoint of the interval $t_{\text{mid}} = (t_{\text{high}} + t_{\text{low}})/2$. and determine which half has the root.
- Repeat the search until the interval has narrowed to give an answer that is “close enough”



Iterative Searches for Roots

Bisector Algorithm

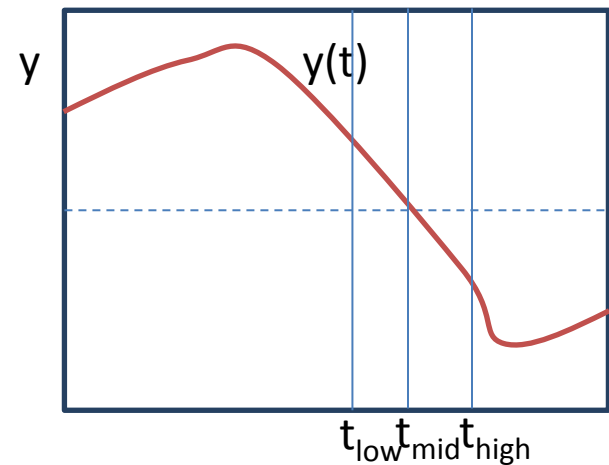
- Find value of t where $y(t)=0$
- if $y(t_{\text{low}})$ and $y(t_{\text{high}})$ have opposite signs then a root exists in the interval.
- Compute the midpoint of the interval $t_{\text{mid}} = (t_{\text{high}} + t_{\text{low}})/2$. and determine which half has the root.
- Repeat the search until the interval has narrowed to give an answer that is “close enough”



Iterative Searches for Roots

Bisector Algorithm

- Find value of t where $y(t)=0$
- if $y(t_{\text{low}})$ and $y(t_{\text{high}})$ have opposite signs then a root exists in the interval.
- Compute the midpoint of the interval $t_{\text{mid}} = (t_{\text{high}} + t_{\text{low}})/2$. and determine which half has the root.
- Repeat the search until the interval has narrowed to give an answer that is “close enough”



Bisector Method – sqrt(32)

```
#Bisector method example, goal is to calculate sqrt(32)
```

```
import numpy as np
```

```
#define the function to find the root of
```

```
def y(t):
```

```
    return t**2 - 32.
```

```
#the initial range to search over and the convergence criteria
```

```
epsilon = 1.e-10
```

```
thi = 32.
```

```
tlow = 0.
```

```
ylow = y(tlow)
```

```
yhi = y(thi)
```

```
#we don't know when to stop so use a while loop
```

```
while (abs(thi-tlow) > 2*epsilon):
```

```
    tmid = (thi+tlow)/2.
```

```
    ymid = y(tmid)
```

```
    if(ymid * ylow > 0):
```

```
        tlow = tmid
```

```
    else:
```

```
        thi = tmid
```

```
print "Root is: ", (thi+tlow)/2.
```

```
print "Actual sqrt(32) is: ", np.sqrt(32.)
```

Bisector Method – sqrt(32)

```
#Bisector method example, goal is to calculate sqrt(32)
```

```
import numpy as np
```

```
#define the function to find the root of
```

```
def y(t):
```

```
    return t**2 - 32.
```

```
#the initial range to search over and the convergence criteria
```

```
epsilon = 1.e-10
```

```
thi = 32.
```

```
tlow = 0.
```

```
ylow = y(tlow)
```

```
yhi = y(thi)
```

```
#we don't know when to stop so use a while loop
```

```
while (abs(thi-tlow) > 2*epsilon):
```

```
    tmid = (thi+tlow)/2.
```

```
    ymid = y(tmid)
```

```
    if(ymid * ylow > 0):
```

```
        tlow = tmid
```

```
    else:
```

```
        thi = tmid
```

```
print "Root is: ", (thi+tlow)/2.
```

```
print "Actual sqrt(32) is: ", np.sqrt(32.)
```

While loop checks the range each loop and quits when the range is within the tolerance.

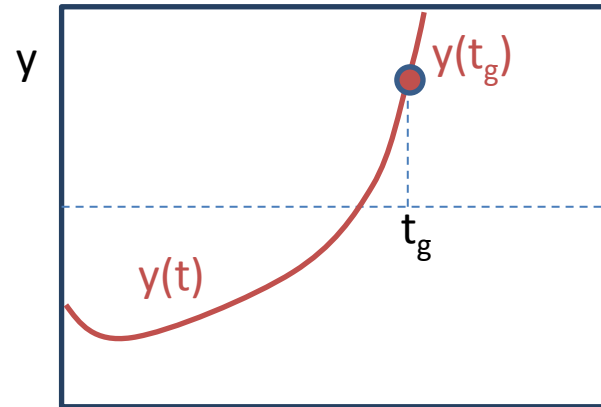
This is only true if both y_{mid} and y_{low} are positive, that is, on the same side of the root. In this case, use the other half of the search area.

Iterative Searches for Roots

Method 2: Newton Raphson Method

If you can calculate first derivatives of your function, you can get to the root faster using the Newton Raphson Method.

1. Choose a starting point, t_g

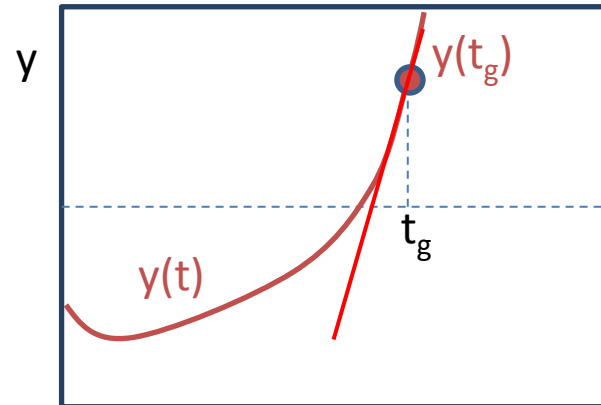


Iterative Searches for Roots

Method 2: Newton Raphson Method

If you can calculate first derivatives of your function, you can get to the root faster using the Newton Raphson Method.

2. calculate $y'(t_g)$ – that is, the slope of $y(t)$ at t_g . Extend this line to $y=0$ to find our updated guess for the root.



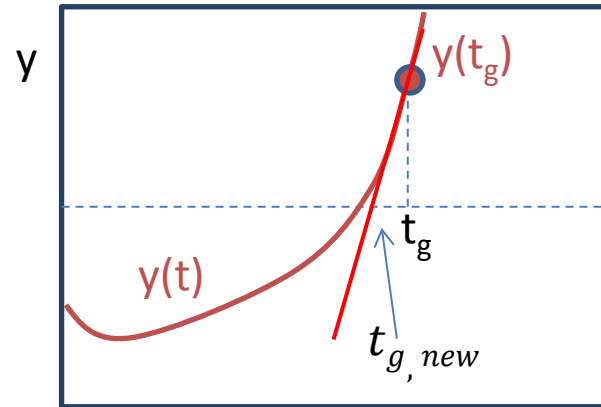
Iterative Searches for Roots

Method 2: Newton Raphson Method

If you can calculate first derivatives of your function, you can get to the root faster using the Newton Raphson Method.

2. Calculate $y'(t_g)$
3. Update t_g as follows:

$$t_{g,new} = t_g - \frac{y(t_g)}{y'(t_g)}$$



Iterative Searches for Roots

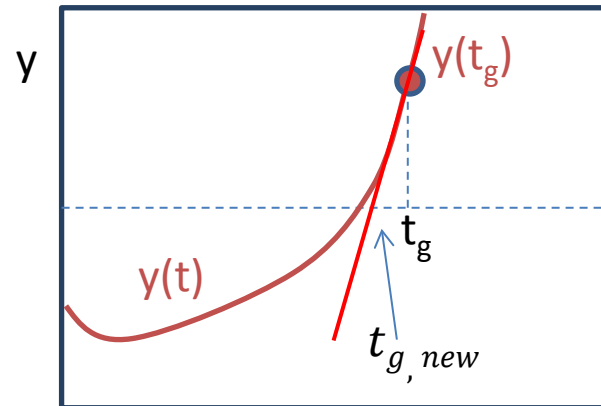
Method 2: Newton Raphson Method

If you can calculate first derivatives of your function, you can get to the root faster using the Newton Raphson Method.

2. Calculate $y'(t_g)$
3. Update t_g as follows:

$$t_{g, new} = t_g - \frac{y(t_g)}{y'(t_g)}$$

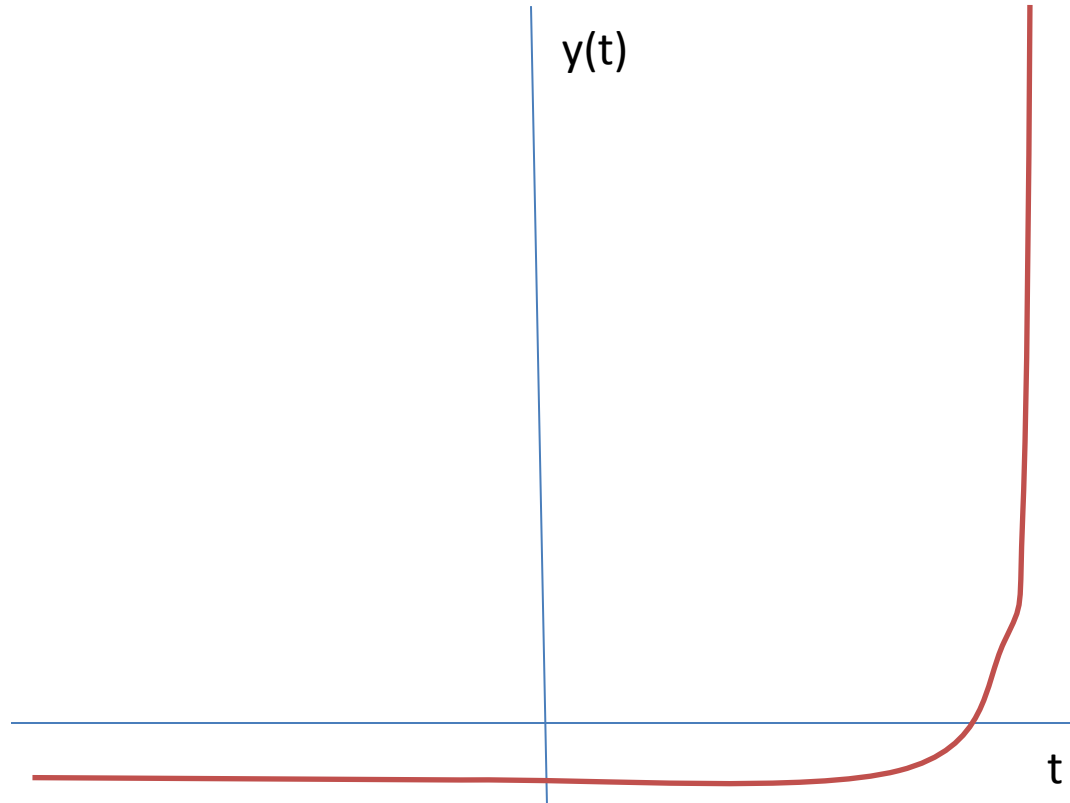
4. Repeat this process until successive values of t_g are sufficiently small.



Notes on Root Finding

- The bisection method is sure-fire if you can bracket a root and if the function is not too poorly behaved.
- The Newton Raphson method is much faster but there are some pathological functions that can cause it to explore areas way outside of your bracketing range.

Try thinking through Newton Raphson
with this function



Notes on Root Finding

- The bisection method is sure-fire if you can bracket a root and if the function is not too poorly behaved.
- The Newton Raphson method is much faster but there are some pathological functions that can cause it to explore areas way outside of your bracketing range.
- For a solid method that uses bisection (and a few other approaches that don't require derivatives) try looking at "Brent's method" which is available in the `scipy.optimize` library as `brentq`.

Exercise

- Implement your own Newton Raphson method to calculate the square root of 32.
- Implement your own NR or bisection to solve the equation

$$x - \sin(x) = 0.05$$