

# On to linear algebra

1

Hopefully the first part of Chapter 6 is a review for you.

# On to linear algebra

2

We will make  
good use of  
numpy here

$$\begin{bmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \boxed{\underline{0} \quad \underline{0}}$$

<https://xkcd.com/184/>

Aside: Need to be careful about storing large matrices unless they are sparse (have lots of zeros). A 1000x1000 matrix has 1 million entries!

# On to linear algebra

We start with a reminder of Gaussian Elimination, a method for solving simultaneous systems of equations, which can be written in matrix form:

$$\mathbf{Ax} = \mathbf{b}$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

Goal: Convert matrix to reduced row echelon form, an upper-triangular form where leading entry in each non-zero row is a 1 and each column containing a leading 1 has zeros in all its other entries. All entries below the diagonal are zero

# Gaussian Elimination

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

## Possible row operations:

- 1) Swap rows
- 2) Multiply a row by a non-zero number
- 3) Add a multiple of one row to another

Once things are in the right form, we back-substitute to solve our system of equations

# Gaussian Elimination example

System of equations	Row operations	Augmented matrix
$2x + y - z = 8$ $-3x - y + 2z = -11$ $-2x + y + 2z = -3$		$\left[ \begin{array}{ccc c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array} \right]$
$2x + y - z = 8$ $\frac{1}{2}y + \frac{1}{2}z = 1$ $2y + z = 5$	$L_2 + \frac{3}{2}L_1 \rightarrow L_2$ $L_3 + L_1 \rightarrow L_3$	$\left[ \begin{array}{ccc c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array} \right]$
$2x + y - z = 8$ $\frac{1}{2}y + \frac{1}{2}z = 1$ $-z = 1$	$L_3 + -4L_2 \rightarrow L_3$	$\left[ \begin{array}{ccc c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array} \right]$
The matrix is now in echelon form (also called triangular form)		<a href="https://en.wikipedia.org/wiki/Gaussian_elimination">https://en.wikipedia.org/wiki/Gaussian_elimination</a>
$2x + y = 7$ $\frac{1}{2}y = \frac{3}{2}$ $-z = 1$	$L_2 + \frac{1}{2}L_3 \rightarrow L_2$ $L_1 - L_3 \rightarrow L_1$	$\left[ \begin{array}{ccc c} 2 & 1 & 0 & 7 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array} \right]$
$2x + y = 7$ $y = 3$ $z = -1$	$2L_2 \rightarrow L_2$ $-L_3 \rightarrow L_3$	$\left[ \begin{array}{ccc c} 2 & 1 & 0 & 7 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array} \right]$
$x = 2$ $y = 3$ $z = -1$	$L_1 - L_2 \rightarrow L_1$ $\frac{1}{2}L_1 \rightarrow L_1$	$\left[ \begin{array}{ccc c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array} \right]$

# Partial pivoting

6

Need to make sure that we don't divide by zero! This can occur if any of the rows has a zero as the first element at any point. We solve this with **partial pivoting**

- 1) Before performing the  $i$ th row operation, search for the element  $a_{ki}$  ( $k=i, \dots, n-1$ ) with the largest magnitude
- 2) If  $k \neq i$ , interchange rows  $i$  and  $k$  of the augmented matrix, and interchange  $x_i <---> x_k$
- 3) Continue

# LU factorization

LU = “Lower Upper”. If we want to solve

$$\mathbf{Ax} = \mathbf{b}$$

we rewrite  $\mathbf{A} = \mathbf{LU}$ , where  $\mathbf{L}$  is a lower triangular matrix and  $\mathbf{U}$  is an upper triangular matrix. Then:

$$\mathbf{LUx} = \mathbf{b}$$

and we can rewrite  $\mathbf{Ux} = \mathbf{y}$

so that  $\mathbf{Ly} = \mathbf{b}$

Since  $\mathbf{L}$  is in triangular form, this is easy to solve and then we solve again  $\mathbf{Ux} = \mathbf{y}$  ( $\mathbf{U}$  is also triangular!)

Why do we do this? If we change  $\mathbf{b} \rightarrow \mathbf{c}$ , then we can still solve the Gaussian elimination on  $\mathbf{A}$  without redoing it so that we save computation, and only need to perform the back-substitutions

# Using numpy

```

import numpy as np
import random as rand
A=np.zeros(25)
A=A.reshape(5,5)
v=np.zeros(5)
v=v.reshape(5,1)
for ix in range(5):
    v[ix][0]=rand.randint(0,10)
    for iy in range(5):
        A[ix,iy]=rand.randint(0,10)
print("A=\n",A)
print("v=\n",v)
x=np.linalg.solve(A,v)
print("Solving for x in Ax=v, x=\n",x)

```



```

A=
[[10.  6.  9.  7. 10.]
 [ 9.  5.  4. 10.  1.]
 [10.  4.  3.  0.  7.]
 [ 3.  0.  8.  3. 10.]
 [ 2.  6. 10.  8.  9.]]
v=
[[ 3.]
 [ 3.]
 [10.]
 [ 3.]
 [ 0.]]
Solving for x in Ax=v, x=
[[ -1.54765432]
 [  0.06765432]
 [-12.20740741]
 [  5.65876543]
 [  8.83259259]]

```

I leave it to you to check  
that the solution is right :)

# Tridiagonal matrices

A **tridiagonal matrix** is one that has entries only along the diagonal plus those above and below

$$\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Can imagine such matrices occurring in nearest-neighbor models or a set of masses on springs. Matrices like these are much easier (read: faster) to solve

# Tridiagonal matrices

$$\left( \begin{array}{ccccc} \frac{a_{00}}{a_{00}} & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{array} \right) \left( \begin{array}{c} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right) = \left( \begin{array}{c} \frac{b_0}{a_{00}} \\ b_1 \\ b_2 \\ x_3 \\ x_4 \end{array} \right)$$

$$\left( \begin{array}{ccccc} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{array} \right) \left( \begin{array}{c} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right) = \left( \begin{array}{c} \frac{b_0}{a_{00}} \\ b_1 \\ b_2 \\ x_3 \\ x_4 \end{array} \right)$$

# Tridiagonal matrices

$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ 0 & a_{11} - \frac{a_{10}a_{01}}{a_{00}} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ b_1 - \frac{a_{10}b_0}{a_{00}} \\ b_2 \\ x_3 \\ x_4 \end{pmatrix}$$

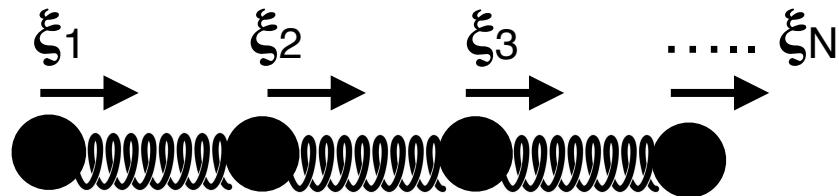
$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ 0 & 1 & \frac{a_{12}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ \frac{b_1 - \frac{a_{10}b_0}{a_{00}}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} \\ b_2 \\ x_3 \\ x_4 \end{pmatrix}$$

# Tridiagonal matrices

$$\left( \begin{array}{cccccc} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 & \\ 0 & 1 & \frac{a_{12}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} & 0 & 0 & \\ 0 & 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} & a_{23}a_{21} & 0 & \\ 0 & 0 & a_{32} & a_{33} & a_{34} & \\ 0 & 0 & 0 & a_{43} & a_{44} & \end{array} \right) \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ \frac{b_1 - \frac{a_{10}b_0}{a_{00}}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} \\ b_2 - a_{21} \frac{b_1 - \frac{a_{10}b_0}{a_{00}}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} \\ x_3 \\ x_4 \end{pmatrix}$$

Can see that if we start at the top, we can recursively solve this by subtracting each row only from the row underneath it and iterating. This is much faster! Let's look at Example 6.2 from the book together

# One-dimensional vibrations of N identical masses (Example 6.2)



$$m \frac{d^2 \xi_i}{dt^2} = k(\xi_{i+1} - \xi_i) + k(\xi_{i-1} - \xi_i) + F_i$$

Hooke's Law and  $F=ma$

$$m \frac{d^2 \xi_1}{dt^2} = k(\xi_2 - \xi_1) + F_1$$

First Spring

$$m \frac{d^2 \xi_N}{dt^2} = k(\xi_{N-1} - \xi_N) + F_N$$

Last Spring

## One-dimensional vibrations of N identical masses (Example 6.2)

Drive the system with  $F_1 = Ce^{i\omega t}$ , other  $F_i = 0$  so that  $\xi_i(t) = x_i e^{i\omega t}$ , where  $x_i$  is the amplitude of the  $i$ th mass

$$m \frac{d^2 \xi_i}{dt^2} = k(\xi_{i+1} - \xi_i) + k(\xi_{i-1} - \xi_i)$$
$$-m\omega^2 = k(x_{i+1} - x_i) + k(x_{i-1} - x_i)$$

True for all but first and last  
masses

# One-dimensional vibrations of N identical masses (Example 6.2)

Drive the system with  $F_1 = Ce^{i\omega t}$ , other  $F_i = 0$  so that  $\xi_i(t) = x_i e^{i\omega t}$ , where  $x_i$  is the amplitude of the  $i$ th mass

$$m \frac{d^2 \xi_1}{dt^2} = k(\xi_2 - \xi_1) + F_1$$

First  
mass

$$-m\omega^2 x_1 = k(x_2 - x_1) + C$$

$$m \frac{d^2 \xi_N}{dt^2} = k(\xi_{N-1} - \xi_N)$$

Last  
mass

$$-m\omega^2 x_N = k(x_{N-1} - x_N)$$

## Our three equations

$$-m\omega^2 x_1 = k(x_2 - x_1) + C$$

$$-m\omega^2 x_i = k(x_{i+1} - x_i) + k(x_{i-1} - x_i)$$

$$-m\omega^2 x_N = k(x_{N-1} - x_N)$$

$$(\alpha - k)x_1 - kx_2 = C$$

$$\alpha x_i - kx_{i-1} - kx_{i+1} = 0$$

$$(\alpha - k)x_N - kx_{N-1} = 0$$

with  $\alpha = 2k - m\omega^2$

These define a  
tridiagonal set of  
simultaneous  
equations!

# From the book's appendix and online tools

```
#####
#
# Function to solve a banded system of linear equations using
# Gaussian elimination and backsubstitution
#
# x = banded(A,v,up,down)
#
# This function returns the vector solution x of the equation A.x = v,
# where v is an array representing a vector of N elements, either real
# or complex, and A is an N by N banded matrix with "up" nonzero
# elements above the diagonal and "down" nonzero elements below the
# diagonal. The matrix is specified as a two-dimensional array of
# (1+up+down) by N elements with the diagonals of the original matrix
# along its rows, thus:
#
#   ( - - A02 A13 A24 ...
#   ( - A01 A12 A23 A34 ...
#   ( A00 A11 A22 A33 A44 ...
#   ( A10 A21 A32 A43 A54 ...
#   ( A20 A31 A42 A53 A64 ...
#
# Elements represented by dashes are ignored -- it doesn't matter what
# these elements contain. The size of the system is taken from the
# size of the vector v. If the matrix A is larger than NxN then the
# extras are ignored. If it is smaller, the program will produce an
# error.
#
# The function is compatible with version 2 and version 3 of Python.
#
# Written by Mark Newman <mejn@umich.edu>, September 4, 2011
# You may use, share, or modify this file freely
#
#####

```

```
from numpy import copy

def banded(Aa,va,up,down):

    # Copy the inputs and determine the size of the system
    A = copy(Aa)
    v = copy(va)
    N = len(v)

    # Gaussian elimination
    for m in range(N):

        # Normalization factor
        div = A[up,m]

        # Update the vector first
        v[m] /= div
        for k in range(1,down+1):
            if m+k<N:
                v[m+k] -= A[up+k,m]*v[m]

    # Now normalize the pivot row of A and subtract from lower ones
    for i in range(up):
        j = m + up - i
        if j<N:
            A[i,j] /= div
            for k in range(1,down+1):
                A[i+k,j] -= A[up+k,m]*A[i,j]

    # Backsubstitution
    for m in range(N-2,-1,-1):
        for i in range(up):
            j = m + up - i
            if j<N:
                v[m] -= A[i,j]*v[j]

    return v
```

# From the book's appendix and online tools

```

from numpy import empty,zeros
from pylab import plot,show

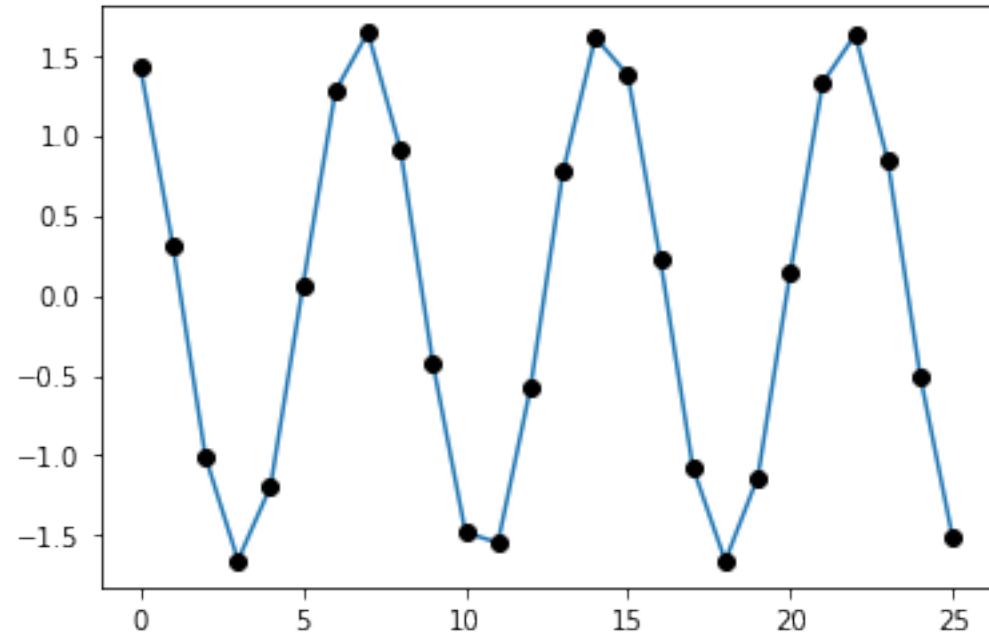
# Constants
N = 26
C = 1.0
m = 1.0
k = 6.0
omega = 2.0
alpha = 2*k-m*omega*omega

# Set up the intial values of the arrays
A = empty([3,N],float)
A[0,:] = -k
A[1,:] = alpha
A[2,:] = -k
A[1,0] = alpha - k
A[1,N-1] = alpha - k
v = zeros(N,float)
v[0] = C

# solve the equations
x = banded(A,v,1,1)

# Make a plot using both dots and lines
plot(x)
plot(x,"ko")
show()

```



# Inverting matrices, as in the book

```
[20] from numpy.linalg import inv
     import random as rand
     A=np.zeros(25)
     A=A.reshape(5,5)
     for ix in range(5):
         for iy in range(5):
             A[ix,iy]=rand.randint(0,10)
     invA = inv(A)
     print("A=\n",A)
     print("A inverse = \n",invA)
```



```
A=
[[ 6.  3.  1.  3.  1.]
 [ 0.  1.  3.  8.  7.]
 [ 2.  3.  8.  2.  8.]
 [ 3.  8.  7.  3.  9.]
 [ 0.  5.  9.  10.  3.]]
A inverse =
[[ 0.17435203 -0.01550952  0.08920579 -0.07484126 -0.03528677]
 [-0.03705631 -0.05162902 -0.17955657  0.19381701  0.03018632]
 [-0.00739044 -0.09456646  0.15800978 -0.09055897  0.07343604]
 [ 0.03445404  0.08171125 -0.06058083 -0.02852087  0.04496721]
 [-0.03091496  0.09737691  0.02716769  0.04371812 -0.08717602]]
```

# How to do DQ decomposition

The goal is to rewrite the matrix as the product of an orthogonal matrix (Q) and an upper-triangular matrix (R).

Let's work through the QR algorithm as explained in Example 6.8. Imagine we have an NxN vector **A**, and we describe it as a set of N column vectors **a<sub>0</sub>**, **a<sub>1</sub>**, ... **a<sub>N-1</sub>**

$$A = \left( \begin{array}{c|c|c|c} | & | & | & \dots \\ a_0 & a_1 & a_2 & \dots \\ | & | & | & \dots \end{array} \right) \quad \text{Define new vectors } \mathbf{u}_i$$

$$\mathbf{u}_0 = \mathbf{a}_0, \mathbf{q}_0 = \frac{\mathbf{u}_0}{|\mathbf{u}_0|}$$

$$\mathbf{u}_1 = \mathbf{a}_1 - (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0, \mathbf{q}_1 = \frac{\mathbf{u}_1}{|\mathbf{u}_1|}$$

$$\mathbf{u}_2 = \mathbf{a}_2 - (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 - (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1, \mathbf{q}_2 = \frac{\mathbf{u}_2}{|\mathbf{u}_2|}$$

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)\mathbf{q}_j, \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|}$$

# How to do DQ decomposition

$$A = \begin{pmatrix} & & & \dots \\ | & | & | & \\ a_0 & a_1 & a_2 & \dots \\ | & | & | & \dots \end{pmatrix}$$

$$\begin{aligned} u_0 &= a_0, \quad q_0 = \frac{u_0}{|u_0|} \\ u_1 &= a_1 - (q_0 \cdot a_1)q_0, \quad q_1 = \frac{u_1}{|u_1|} \\ u_2 &= a_2 - (q_0 \cdot a_2)q_0 - (q_1 \cdot a_2)q_1, \quad q_2 = \frac{u_2}{|u_2|} \\ u_i &= a_i - \sum_{j=0}^{i-1} (q_j \cdot a_i)q_j, \quad q_i = \frac{u_i}{|u_i|} \end{aligned}$$

Want to prove  
orthonormality, ie:

$$q_i \cdot q_j = 1(i == j), 0(i \neq j)$$

Suppose that for some value of i the  
vectors  $q_k$  are orthonormal

$$0 \leq k < i$$

$$q_i \cdot q_k = \frac{u_i}{|u_i|} \cdot q_k = \frac{1}{|u_i|} \left[ a_i - \sum_{j=0}^{i-1} (q_j \cdot a_i)q_j \right] \cdot q_k$$

# How to do DQ decomposition

Suppose that for some value of  $i$  the vectors  $\mathbf{q}_k$  are orthonormal

$$0 \leq k < i$$

$$\mathbf{q}_i \cdot \mathbf{q}_k = \frac{\mathbf{u}_i}{|\mathbf{u}_i|} \cdot \mathbf{q}_k = \frac{1}{|\mathbf{u}_i|} \left[ \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i) \mathbf{q}_j \right] \cdot \mathbf{q}_k$$

$$\mathbf{q}_i \cdot \mathbf{q}_k = \frac{1}{|\mathbf{u}_i|} \left[ \mathbf{a}_i \cdot \mathbf{q}_k - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)(\mathbf{q}_j \cdot \mathbf{q}_k) \right]$$


ZERO unless  $j=k$ , in which case that term is 1!

$$\mathbf{q}_i \cdot \mathbf{q}_k = \frac{1}{|\mathbf{u}_i|} [\mathbf{a}_i \cdot \mathbf{q}_k - (\mathbf{q}_k \cdot \mathbf{a}_i)(\mathbf{q}_k \cdot \mathbf{q}_k)]$$

# How to do DQ decomposition

Suppose that for some value of  $i$  the vectors  $\mathbf{q}_k$  are orthonormal

$$\mathbf{q}_i \cdot \mathbf{q}_k = \frac{1}{|\mathbf{u}_i|} [ \mathbf{a}_i \cdot \mathbf{q}_k - (\mathbf{q}_k \cdot \mathbf{a}_i)(\mathbf{q}_k \cdot \mathbf{q}_k) ] \quad 0 \leq k < i$$

$$\mathbf{q}_i \cdot \mathbf{q}_k = \frac{1}{|\mathbf{u}_i|} [ \mathbf{a}_i \cdot \mathbf{q}_k - (\mathbf{q}_k \cdot \mathbf{a}_i) ]$$

$$\mathbf{q}_i \cdot \mathbf{q}_k = 0$$

$$\mathbf{q}_i \cdot \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|} \cdot \frac{\mathbf{u}_i}{|\mathbf{u}_i|} = \frac{|\mathbf{u}_i|^2}{|\mathbf{u}_i|^2} = 1$$

# How to do DQ decomposition

Suppose that for some value of  $i$  the vectors  $\mathbf{q}_k$  are orthonormal

$$0 \leq k < i$$

$$\mathbf{q}_i \cdot \mathbf{q}_k = 0$$

$$\mathbf{q}_i \cdot \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|} \cdot \frac{\mathbf{u}_i}{|\mathbf{u}_i|} = \frac{|\mathbf{u}_i|^2}{|\mathbf{u}_i|^2} = 1$$

So if the  $(i-1)$  eigenvectors are orthonormal, so are the first  $i$  eigenvectors. But we know that  $\{\mathbf{q}_0\}$  is orthonormal as a set, so we can use induction to prove that all of them are!

# How to do DQ decomposition

$$A = \begin{pmatrix} & & & \cdots \\ | & | & | & \cdots \\ a_0 & a_1 & a_2 & \dots \\ | & | & | & \dots \end{pmatrix}$$

$$u_0 = a_0, q_0 = \frac{u_0}{|u_0|}$$

$$u_1 = a_1 - (q_0 \cdot a_1)q_0, q_1 = \frac{u_1}{|u_1|}$$

$$u_2 = a_2 - (q_0 \cdot a_2)q_0 - (q_1 \cdot a_2)q_1, q_2 = \frac{u_2}{|u_2|}$$

$$u_i = a_i - \sum_{j=0}^{i-1} (q_j \cdot a_i)q_j, q_i = \frac{u_i}{|u_i|}$$

Rewriting things:

$$a_0 = |u_0|q_0$$

$$a_1 = |u_1|q_1 + (q_0 \cdot a_1)q_0$$

$$a_2 = |u_2|q_2 + (q_0 \cdot a_2)q_0 + (q_1 \cdot a_2)q_1$$

Let's multiply it out and check this

**A**

**Q**

**R**

$$A = \begin{pmatrix} & & & \cdots \\ | & | & | & \cdots \\ a_0 & a_1 & a_2 & \dots \\ | & | & | & \dots \end{pmatrix} = \begin{pmatrix} & & & \cdots \\ | & | & | & \cdots \\ q_0 & q_1 & q_2 & \dots \\ | & | & | & \dots \end{pmatrix} \begin{pmatrix} |u_0| & q_0 \cdot a_1 & q_0 \cdot a_2 & \cdots \\ 0 & |u_1| & q_1 \cdot a_2 & \cdots \\ 0 & 0 & |u_2| & \cdots \end{pmatrix}$$

# Exercise 6.8

# Exercise 6.8 QR Algorithm

```
from numpy import sqrt, array, empty, zeros, dot, copy, matmul
from numpy import identity, absolute, max
```

# Magnitude of a vector

```
def mag(v):
    return sqrt(dot(v,v))
```

# Function to calculate QR decomposition

```
def QR(A):
```

```
    U = empty([N,N],float)
```

```
    Q = empty([N,N],float)
```

```
    R = zeros([N,N],float)
```

```
    for m in range(N):
```

```
        U[:,m] = A[:,m]
```

```
        for i in range(m):
```

```
            R[i,m] = dot(Q[:,i],A[:,m])
```

```
            U[:,m] -= R[i,m]*Q[:,i]
```

```
        R[m,m] = mag(U[:,m])
```

```
        Q[:,m] = U[:,m]/R[m,m]
```

```
    return Q,R
```

```
[[ 2.1000000e+01  7.36067983e-07  2.00505365e-14 -4.44738475e-14]
 [ 7.36067998e-07 -8.00000000e+00  5.99038894e-08 -4.92772067e-15]
 [ 1.76373367e-14  5.99038974e-08 -3.00000000e+00 -2.06494012e-08]
 [ 5.69063112e-23  9.58659066e-16 -2.06493983e-08  1.00000000e+00]]
[[ 2.1000000e+01  1.93217851e-06  1.21048159e-13 -4.44738470e-14]
 [ 1.93217849e-06 -8.00000000e+00  1.59743718e-07  1.35556481e-14]
 [ 1.23461357e-13  1.59743727e-07 -3.00000000e+00 -6.19481921e-08]
 [ 1.19503254e-21  7.66927253e-15 -6.19481950e-08  1.00000000e+00]]
[[ 0.43151698 -0.38357064 -0.77459666 -0.25819889]
 [ 0.38357063  0.43151698 -0.25819889  0.77459667]
 [ 0.62330228  0.52740965  0.25819889 -0.51639778]
 [ 0.52740965 -0.62330227  0.51639779  0.25819889]]
```

# Main program

```
A = array( [[1,4,8,4],
            [4,2,3,7],
            [8,3,6,9],
            [4,7,9,2]],float)
```

```
N = len(A)
```

```
epsilon = 1e-6
```

```
V = identity(N,float) # start with identity matrix
delta = 1.0
```

```
Q = empty([N,N],float)
```

```
R = zeros([N,N],float)
```

```
while delta > epsilon:
```

# Perform a step of the QR alg

```
Q,R = QR(A)
```

```
A = dot(R,Q)
```

```
V = dot(V,Q)
```

# Find the largest off-diagonal element

```
Ac = copy(A)
```

```
for i in range(N): Ac[i,i] = 0.0
```

```
delta = max(absolute(Ac))
```

```
print(A)
```

```
print(matmul(Q,R))
```

```
print(V)
```

# Eigenvalues and eigenvectors, as in the book

```
A=np.array([[1,2,3],[4,5,6],[7,8,9.]])  
x,v = np.linalg.eigh(A)  
print("A=",A)  
print(x)  
print(v)  
size=len(A)  
for i in range(size):  
    print("eigenvalue = ",x[i]," and eigenvector = \n",v[:,i].reshape(size,1))
```

```
A= [[1. 2. 3.]  
 [4. 5. 6.]  
 [7. 8. 9.]]  
[-3.15746784 -0.67276795 18.8302358 ]  
[[-0.80238891 0.43402538 -0.40962667]  
 [-0.16812656 -0.82296167 -0.54264865]  
 [ 0.57263033 0.36654613 -0.73330651]]  
eigenvalue = -3.1574678406080765 and eigenvector =  
[[-0.80238891]  
 [-0.16812656]  
 [ 0.57263033]]  
eigenvalue = -0.6727679548987614 and eigenvector =  
[[ 0.43402538]  
 [-0.82296167]  
 [ 0.36654613]]  
eigenvalue = 18.830235795506834 and eigenvector =  
[[-0.40962667]  
 [-0.54264865]  
 [-0.73330651]]
```

Of course, we typically use the numpy or other such libraries!

# Nonlinear equations

Consider the equation  
 $x=4-e^{0.5x}+e^{0.4x}$

What is the solution to this? No analytic solution.

Relaxation method: We guess a value and then plug it in and then iterate and try to get convergence on an answer

It's converging... slowly. Let's try and add more iterations

```
[2] from math import exp
x=5 ## guess a guess!
for k in range(40):
    x=4-exp(0.5*x)+exp(0.4*x)
    print(x)

⇒ -0.7934378617728228
4.055534552419392
1.4671841743770724
3.715809477043053
2.010522319728514
3.5023069797459816
2.2976994178947336
3.352420163397929
2.4774829414284874
3.2426414615958707
2.5987437850508344
3.160803050978196
2.683782981887578
3.0993657751234003
2.7447605678816256
3.0531474625599433
2.7890740540985193
3.0183819369685114
2.821548955858161
2.9922572894874393
2.845477746880245
2.9726504416168322
2.863173287603618
2.9579532158473465
2.8762915098745165
2.946948002138102
2.8860330259574583
2.9387146570520355
2.893275709680601
2.9325594161794712
2.89866515888904
2.9279603349076195
```

# Nonlinear equations

```
[6] from math import exp
x=5 ## guess a guess!
def func(x):
    return 4-exp(0.5*x)+exp(0.4*x)

n=1000
for k in range(n):
    x=func(x)
    if (k>980): print(x)

print ("At the end x = ",x, "and func = ", func(x))
```

At the end x = 2.

# Nonlinear equations

```
[18] from math import exp,sin,cos
x=5 ## guess a guess!
def func(x):
    return exp(sin(x))+cos(x)

n=1000
for k in range(n):
    x=func(x)
    if (k>980): print(x)

print ("At the end x = ",x, "and func = ", func(x))
```

```
↳ 2.4216195085101226
1.1817396931849844
2.9018581769719125
0.29660382444181976
2.2958046227904654
1.450672394014024
2.8185989821906654
0.425271828760605
2.4216195085101226
1.1817396931849844
2.9018581769719125
0.29660382444181976
2.2958046227904654
1.450672394014024
2.8185989821906654
0.425271828760605
2.4216195085101226
1.1817396931849844
2.9018581769719125
```

```
At the end x =  2.9018581769719125 and func =  0.29660382444181976
```

Hmmm that is clearly not working! Why not?

# Nonlinear equations

$x^*$  is a solution and let's Taylor expand to find  $x'$  (new value) after an iteration of the method

$$\begin{aligned}x' = f(x) &= f(x^*) + (x - x^*)f'(x) + \dots = x^* + (x - x^*)f'(x^*) + \dots \\(x' - x^*) &= (x - x^*)f'(x^*)\end{aligned}$$

Tells us that the distance to the true solution will get smaller and smaller if the absolute value of the derivative at the solution is  $< 1$ !

# Nonlinear equations

```
[24] from math import exp,sin,cos,asin,log
      x=0.1 ## guess a guess!
      def func(x):
          return exp(1-x**3)

      n=1000
      for k in range(n):
          x=func(x)
          if (k>980): print(x)

      print ("At the end x = ",x, "and func = ", func(x))
```

## Also not working

# Nonlinear equations

```
[30] from math import exp,sin,cos,asin,log,pow
x=2 ## guess a guess!
def func(x):
    return pow(1-log(x),1/3.)
n=200
for k in range(n):
    x=func(x)
    if (k > 190): print(x)

print ("At the end x = ",x, "and func = ", func(x))
```

Sometimes a different form will have better slope and convergence

```
↳ 1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
At the end x = 1.0 and func = 1.0
```

$x=e^{(1-x^3)}$  didn't converge.  
But let's rearrange:

$$x=e(1-x^3)$$

$$\ln(x) = 1-x^3$$

$$x^3 = 1-\ln(x)$$

$$x = (1-\ln(x))^{1/3}$$

Quick check:  $1=1^{1/(1-1^3)}$ , yes!

# Another relaxation method example

### Exercise 6.10, solve  $x = 1 - e^{-cx}$  for unknown  $c$

```
from math import exp
from numpy import arange
from pylab import plot,xlabel,ylabel,show

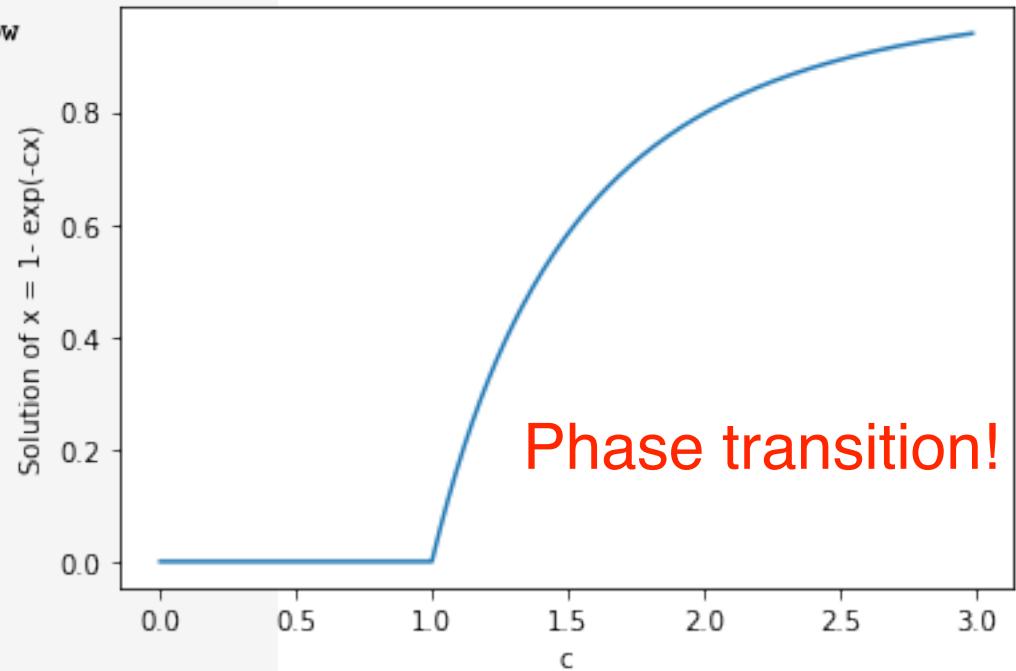
target = 1e-6 # Target accuracy

def f(x,c):
    return 1-exp(-c*x)

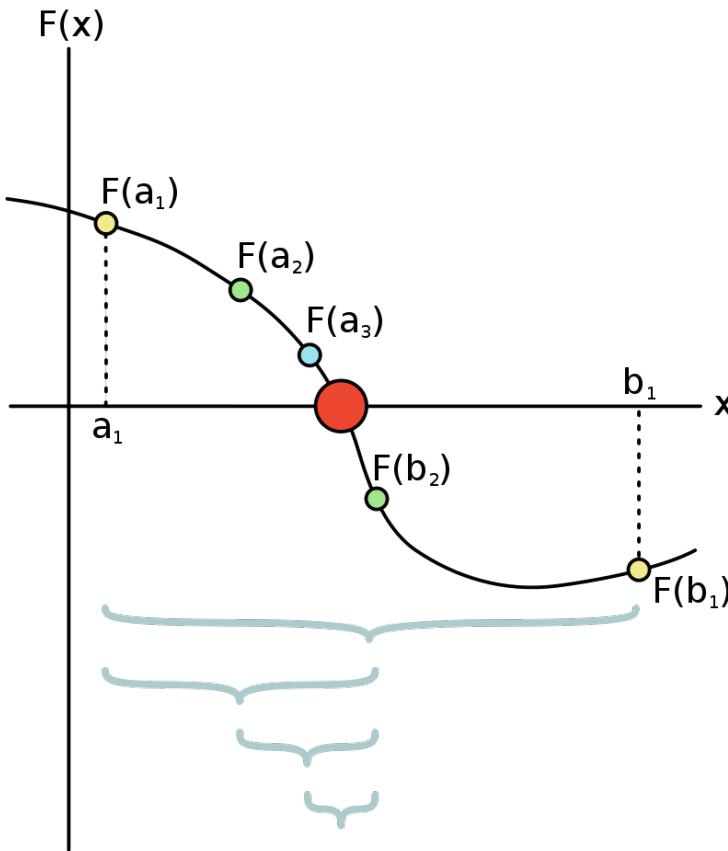
def fp(x,c):
    return c*exp(-c*x)

cpoints=arange(0,3,0.01)
xpoints=[]
for c in cponts:
    x = 1.0
    epsilon = 1.0
    while epsilon > target:
        xp = f(x,c)
        derivative = fp(xp,c)
        if (derivative < 1e-10): derivative = 1e-10
        epsilon = abs((xp-x)/(1-1/derivative))
        x = xp
    xpoints.append(x)

plot(cpoints,xpoints)
xlabel("c")
ylabel("Solution of x = 1- exp(-cx)")
show()
```



# Bisection method



[https://en.wikipedia.org/wiki/Bisection\\_method](https://en.wikipedia.org/wiki/Bisection_method)

For  $F(x) = 0$ , if we can find  $a_1$  and  $b_1$  such that  $F(a_1)$  and  $F(b_1)$  have opposite sign, then we know there must be a solution  $F(x) = 0$  for some

$a_1 < b_1$  ! We choose a point halfway between the two and use its sign to decide which direction to use, and iterate

Doesn't work for even-order polynomials or roots falling in the same place

# Bisection method on our old friend

```
[13] from math import sin,exp,cos,log
    def f(x):
        return exp(1-x**3)-x ## x=exp(1-x**3)

    a=-5
    b=5
    midpoint=0.5*(a+b)
    eval=f(midpoint)
    if (f(a)*f(b) > 0): print("No good, initial endpoints have the same sign")
    nmax=10000 ## break if we get stuck
    n=0
    epsilon=1e-8 ### how close we want to get

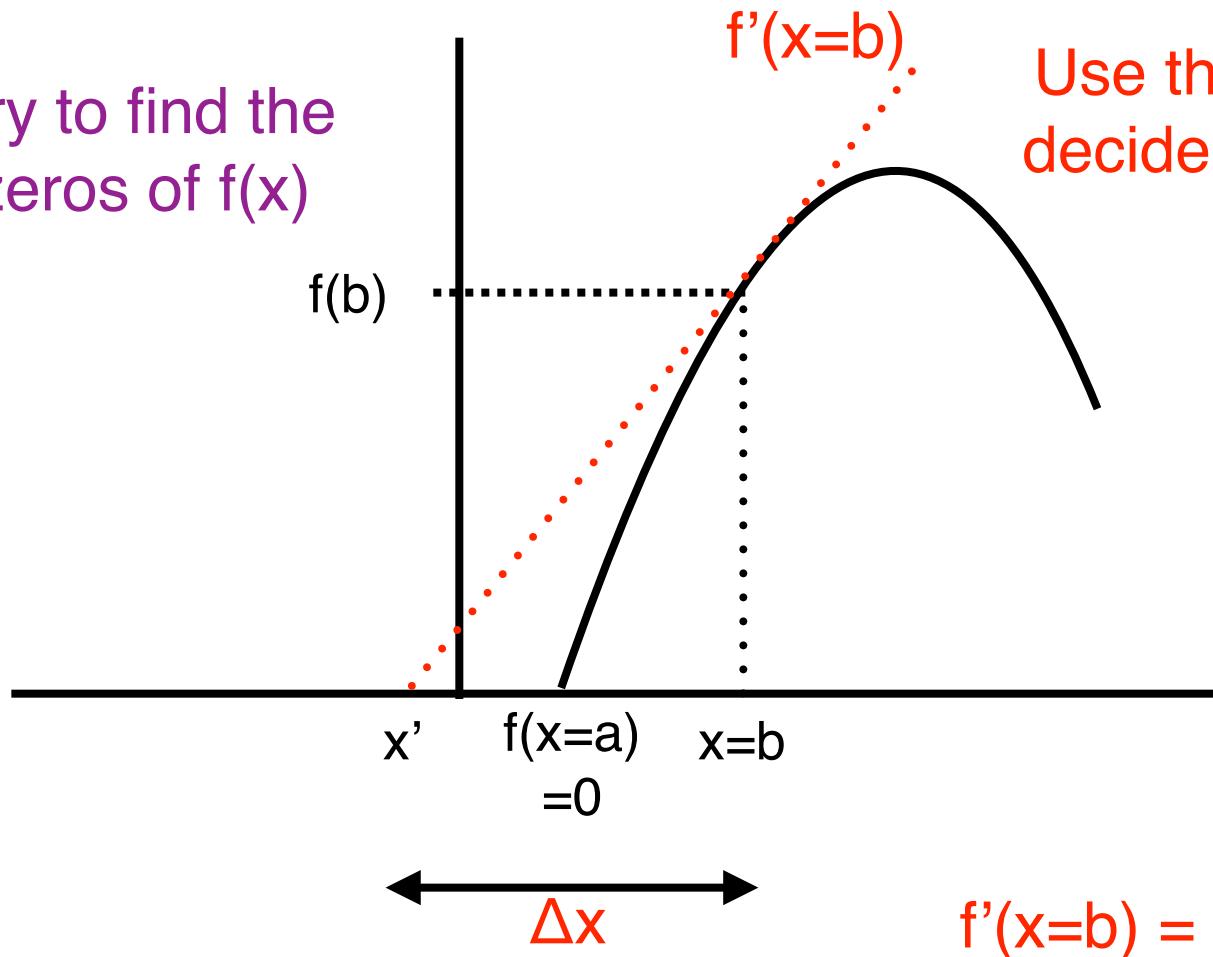
    while((abs(b-a) > epsilon) and n < nmax):
        if ((f(midpoint) * f(a)) > 0): a = midpoint
        else: b = midpoint
        midpoint=0.5*(a+b)
        n = n+1
    print("It took", n, "iterations", " and midpoint = ", midpoint)
```

No transformation needed, Just need to put the function in terms of  $f(x) = 0$  and this worked!

 It took 30 iterations and midpoint = 1.0000000009313226

# Newton's method

Try to find the zeros of  $f(x)$



Use the slope at  $f(x)$  to decide what new guess to make

New guess:  

$$x' = x - \Delta x = x - f(x)/f'(x)$$

# How fast do we converge on the solution?

Let  $f(x^*)$  be a solution such that  $f(x^*) = 0$ .

Then Taylor expand about where our current position,  $x$

$$f(x^*) = f(x) + (x^* - x)f'(x) + \frac{1}{2}(x^* - x)^2 f''(x) + \dots$$

$$0 = f(x) + (x^* - x)f'(x) + \frac{1}{2}(x^* - x)^2 f''(x) + \dots$$

$$x^* = [x - \frac{f(x)}{f'(x)}] - \frac{1}{2}(x^* - x)^2 \frac{f''(x)}{f'(x)} + \dots$$

$$x^* = x' - \frac{1}{2}(x^* - x)^2 \frac{f''(x)}{f'(x)} + \dots$$

# How fast do we converge on the solution?

$$x^* = x' - \frac{1}{2}(x^* - x)^2 \frac{f''(x)}{f'(x)} + \dots$$

Define errors on our estimates such that

$x^* = x + \epsilon = x' + \epsilon'$  ( $x$  is our current estimate,  $x'$  is our next estimate)

$$x^* = x' - \frac{1}{2}\epsilon^2 \frac{f''(x)}{f'(x)} + \dots$$

$$\epsilon' = -\frac{\epsilon^2}{2} \frac{f''(x)}{f'(x)} + \dots$$

Error on our next estimate is the square of the current error. This means it can converge very quickly!

# How fast do we converge on the solution?

$$\epsilon' = -\frac{\epsilon^2}{2} \frac{f''(x)}{f'(x)} + \dots$$

Error on our next estimate is the square of the current error. This means it can converge very quickly!

Current error is  $\epsilon$ , after one iteration it is  $\epsilon^2$ , after two iterations it is  $\epsilon^4$ , after three iterations  $\epsilon^8$ . After  $N$  iterations it will be  $\epsilon^{2^N}$ . As long as  $f''$  and  $f'$  are not changing wildly, we can find the solution for small  $N$ . In practice we just stop after  $x'$  and  $x$  are very close together

Be careful that  $f'$  isn't small (if it's zero we're in trouble!) or  $f''$  isn't too big

# Example

```
[10] def f(x):
    return 123*x***6 - 456*x***5 + 789*x***4 - 1012*x***3 + 3456*x***2 - 7890*x - 123456

def fp(x):
    return 6*123*x***5 - 5*456*x***4 + 4*789*x***3 - 3*1012*x***2 + 2*3456*x - 7890

accuracy = 1e-12
delta = 1.0
x = 0.0
N=0
while abs(delta) > accuracy and N < maxN:
    delta = f(x)/fp(x)
    x -= delta
    N=N+1
print("It took", N, "iterations to find x = ", x, "with f(x) = ", f(x))
```

VERY QUICK!

 It took 16 iterations to find  $x = -2.3262553592304425$  with  $f(x) = -1.4551915228366852e-11$

```
[12] def f(x):
    return 123*x***6 - 456*x***5 + 789*x***4 - 1012*x***3 + 3456*x***2 - 7890*x - 123456

def fp(x):
    return 6*123*x***5 - 5*456*x***4 + 4*789*x***3 - 3*1012*x***2 + 2*3456*x - 7890

accuracy = 1e-12
delta = 1.0
x = 1e10
N=0
while abs(delta) > accuracy:
    delta = f(x)/fp(x)
    x -= delta
    N=N+1
print("It took", N, "iterations to find x = ", x, "with f(x) = ", f(x))
```

Started super far away and took only  
126 iterations, but note that we found a  
different solution. Be careful!

 It took 126 iterations to find  $x = 3.74107809105797$  with  $f(x) = -2.9103830456733704e-11$

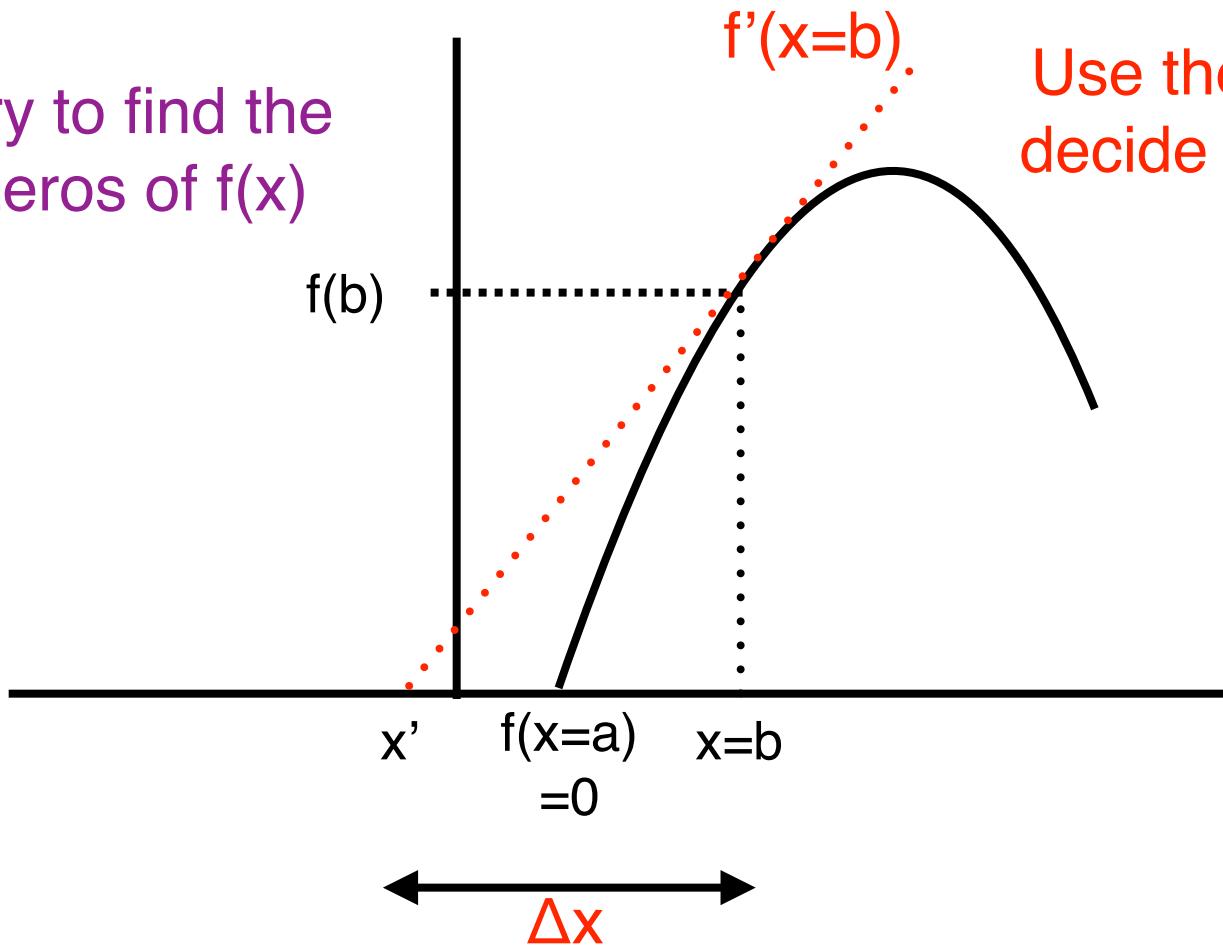
## How do we use Newton's method?

If we know how to calculate  $f'(x)$ , then we can easily use Newton's method as in the last slide. And sometimes we will have an analytical formula for this. Often-times, we won't, though, but we can just use our previously found estimate of the derivative at a point:

$$f'(x_2) \sim \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

# Newton's method for two or more variables

Try to find the zeros of  $f(x)$



New guess:  
 $x' = x - \Delta x = x - f(x)/f'(x)$

Use the slope at  $f(x)$  to decide what new guess to make

This is in 1D, but what if we have not position  $x$  but a vector  $\mathbf{x} = (x_1, x_2, \dots, x_N)$

# Newton's method for two or more variables

We have not position  $x$  but  
a vector  $\mathbf{x} = (x_1, x_2, \dots, x_N)$   
and  $N$  equations:

$$f_1(\mathbf{x}) = 0$$

$$f_2(\mathbf{x}) = 0$$

...

$$f_N(\mathbf{x}) = 0$$

Write the  $N$  equations as a  
vector, too, solve for  $\mathbf{x}^*$   
where  $\mathbf{f}(\mathbf{x}^*) = 0$

Need to solve system of  
equations!

$$\mathbf{f}(\mathbf{x}^*) = \mathbf{f}(\mathbf{x}) + \mathbf{J} \cdot (\mathbf{x}^* - \mathbf{x}) + \dots$$

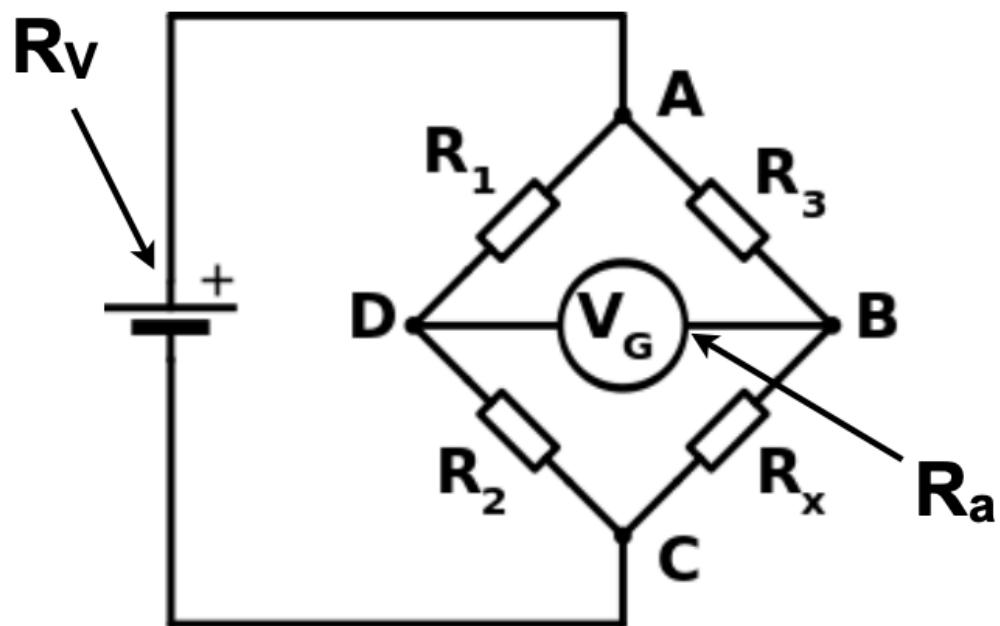
with  $\mathbf{J}$  the Jacobian matrix,

$$\mathbf{J} \cdot \Delta \mathbf{x} = \mathbf{f}(\mathbf{x})$$

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

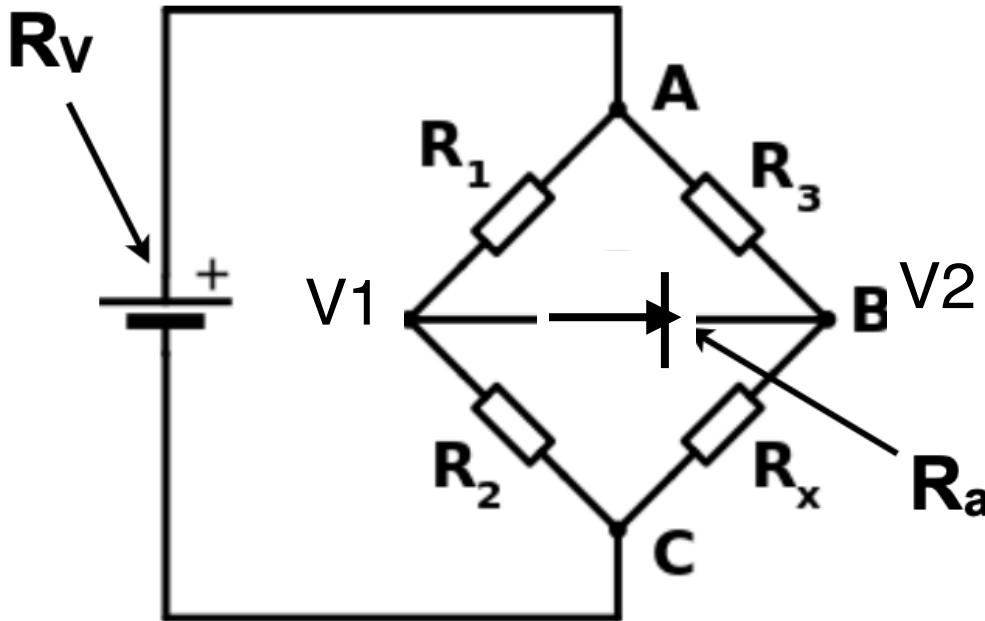
# Wheatstone bridge (from Sal)

-[http://en.wikipedia.org/wiki/Wheatstone\\_bridge](http://en.wikipedia.org/wiki/Wheatstone_bridge)



Solve for  $R_x$  given  $R_1$ ,  $R_2$ ,  $R_3$ ,  $i$  and  $V_G$ . Adjust  $R_2$  until  $V_G$  is zero, tells us  $R_x$

## Exercise 6.17



$V_g$  is replaced now with  
a diode!  $I=I_0(e^{V/V_T}-1)$

Junction at  $V_1$

$$\frac{V_1 - V_+}{R_1} + \frac{V_1}{R_2} + I_0(e^{(V_1 - V_2)/V_T} - 1) = 0$$

Junction at  $V_2$

$$\frac{V_2 - V_+}{R_3} + \frac{V_2}{R_4} + I_0(e^{(V_1 - V_2)/V_T} - 1) = 0$$

# Exercise 6.17

## Junction at V1

$$\frac{V_1 - V_+}{R_1} + \frac{V_1}{R_2} + I_0(e^{(V_1 - V_2)/V_T} - 1) = 0$$

## Junction at V2

$$\frac{V_2 - V_+}{R_3} + \frac{V_2}{R_4} + I_0(e^{(V_1 - V_2)/V_T} - 1) = 0$$

```
# Exercise 6.17 using Newton's method
from math import sqrt,exp
R1 = 1000.0
R2 = 4000.0
R3 = 3000.0
R4 = 2000.0
Vp = 5.0
VT = 0.05
I0 = 3.0e-9
target = 1.0e-8

# initial guesse
V1 = 1.0
V2 = 1.0

# Main loop
error = 1.0
while error > target:
    # Calculate matrix elements and determinant
    x = exp((V1-V2)/VT)*I0/VT
    a = 1/R1 + 1/R2 + x
    b = -x
    c = -x
    d = 1/R3 + 1/R4 + x
    det = a*d - b*c

    # Calculate inverse
    ia = d/det
    id = a/det
    ib = -b/det
    ic = -c/det

    # Calculate vector elements
    y = I0*(exp(((V1-V2)/VT)-1))
    z1 = (V1-Vp)/R1 + V1/R2 + y
    z2 = (V2-Vp)/R3 + V2/R4 - y

    # Calculate delta x
    deltaV1 = ia*z1 + ib*z2
    deltaV2 = ic*z1 + id*z2

    # Calculate the error
    error = sqrt(deltaV1**2+deltaV2**2)

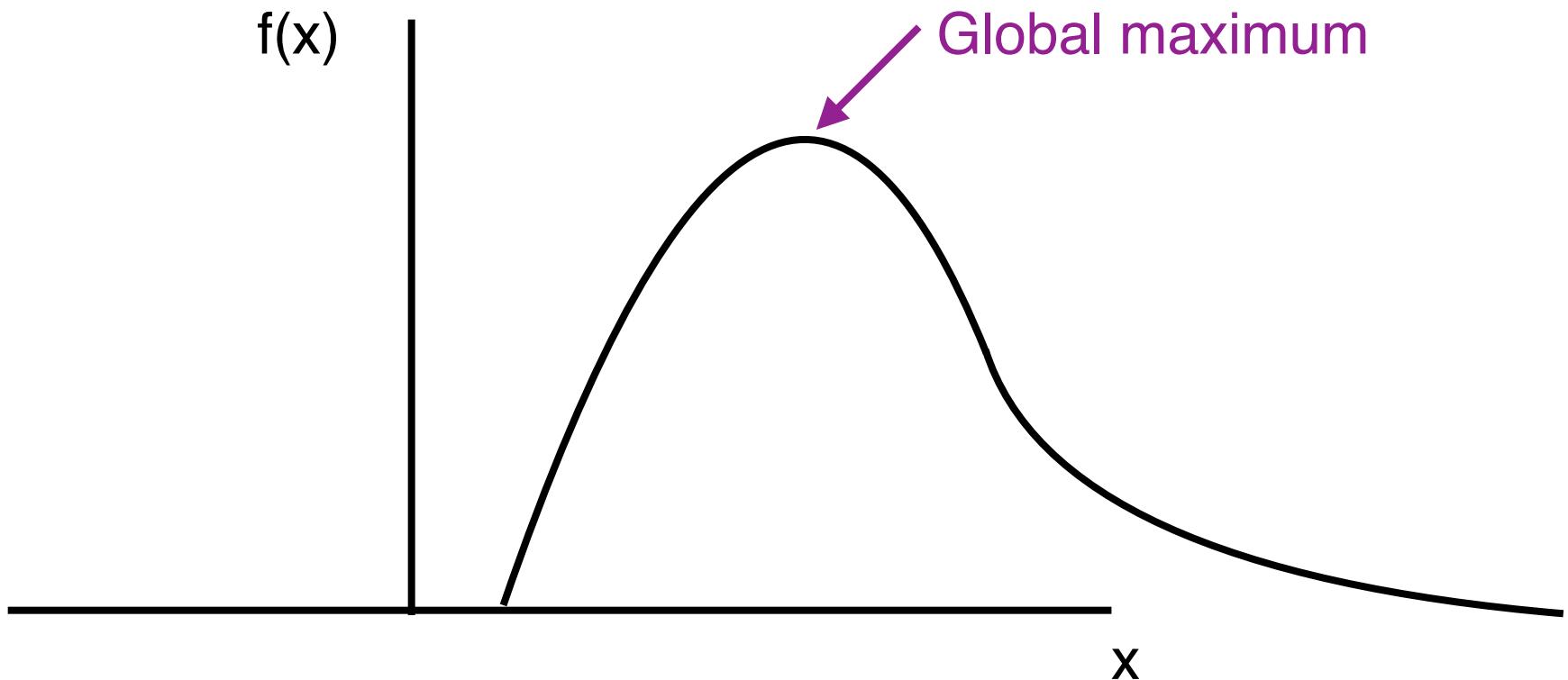
    # Calculate new voltages
    V1 -= deltaV1
    V2 -= deltaV2

# Print results
print(V1,V2)
```



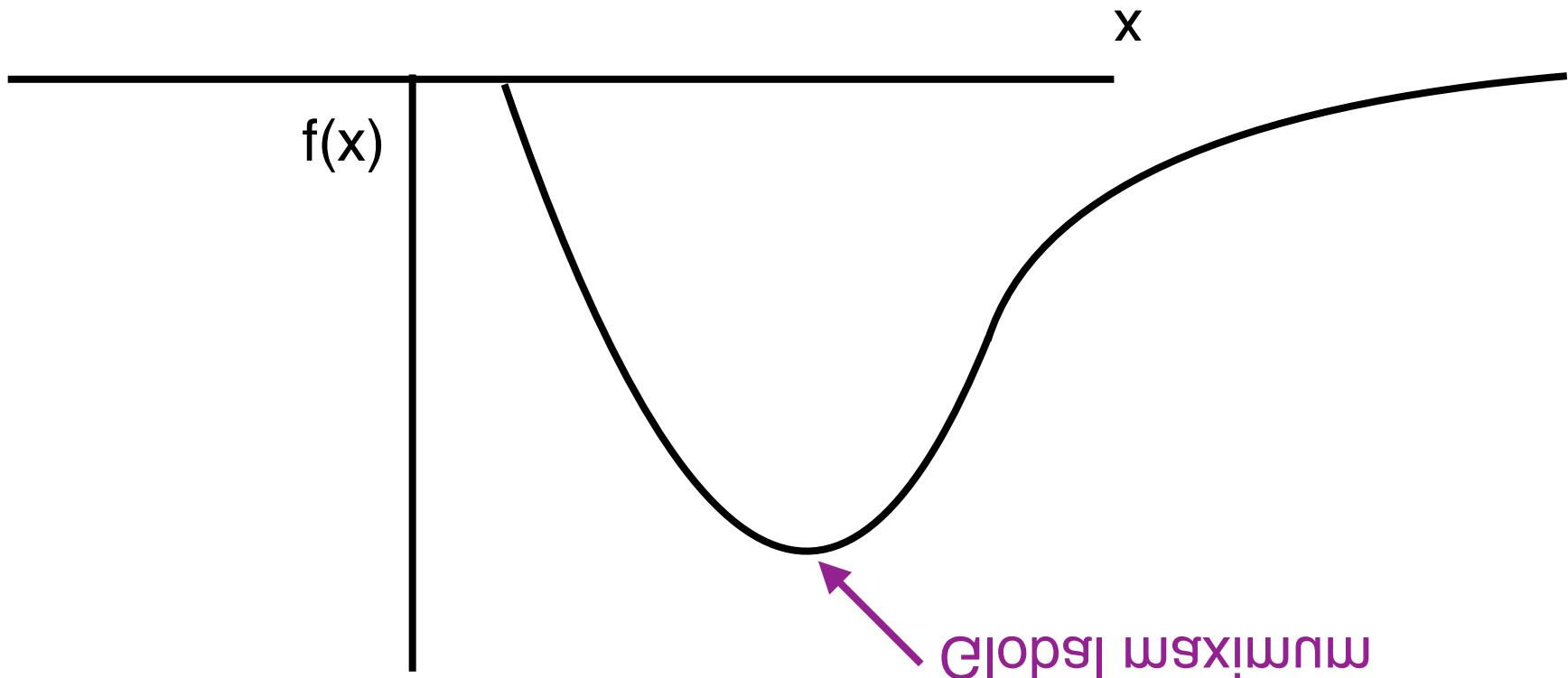
3.466244495626595 2.8006332565601078

# Minimization and maximization problems



Important question to ask:  
How to find the maximum  
value of  $f(x)$ ?

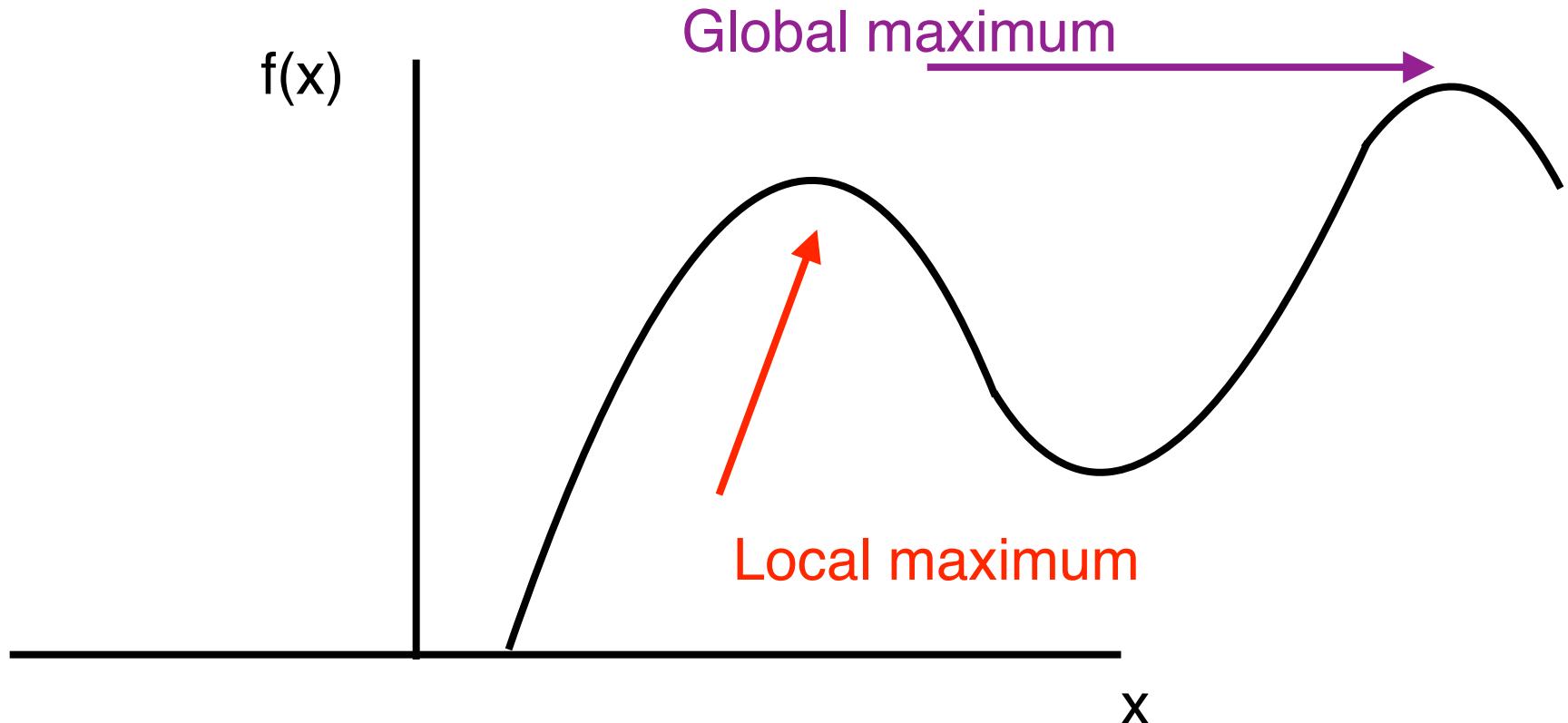
# Minimization and maximization problems



Global maximum  
Global minimum

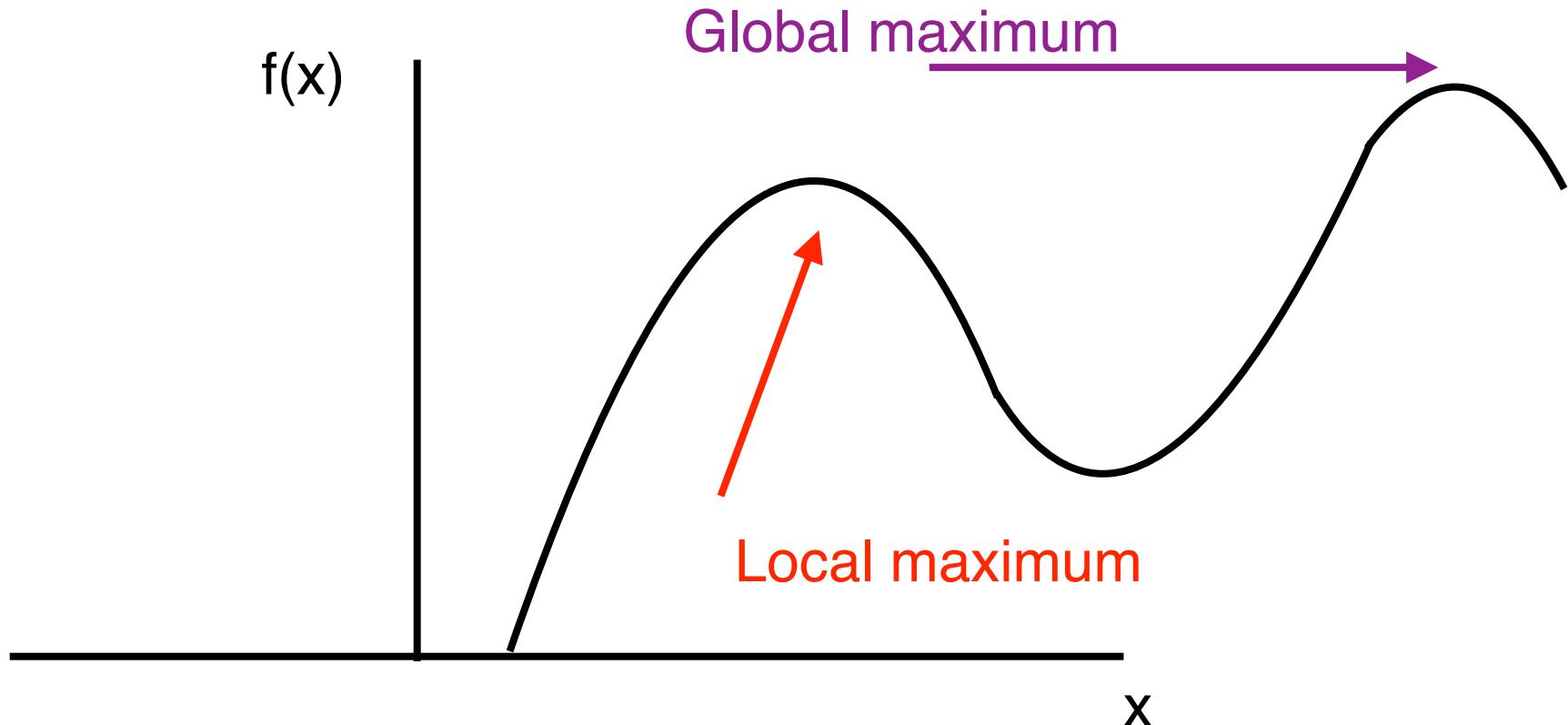
Finding the minimum and  
finding the maximum is  
equivalent (just flip by  
multiplying with -1)!

# Minimization and maximization problems



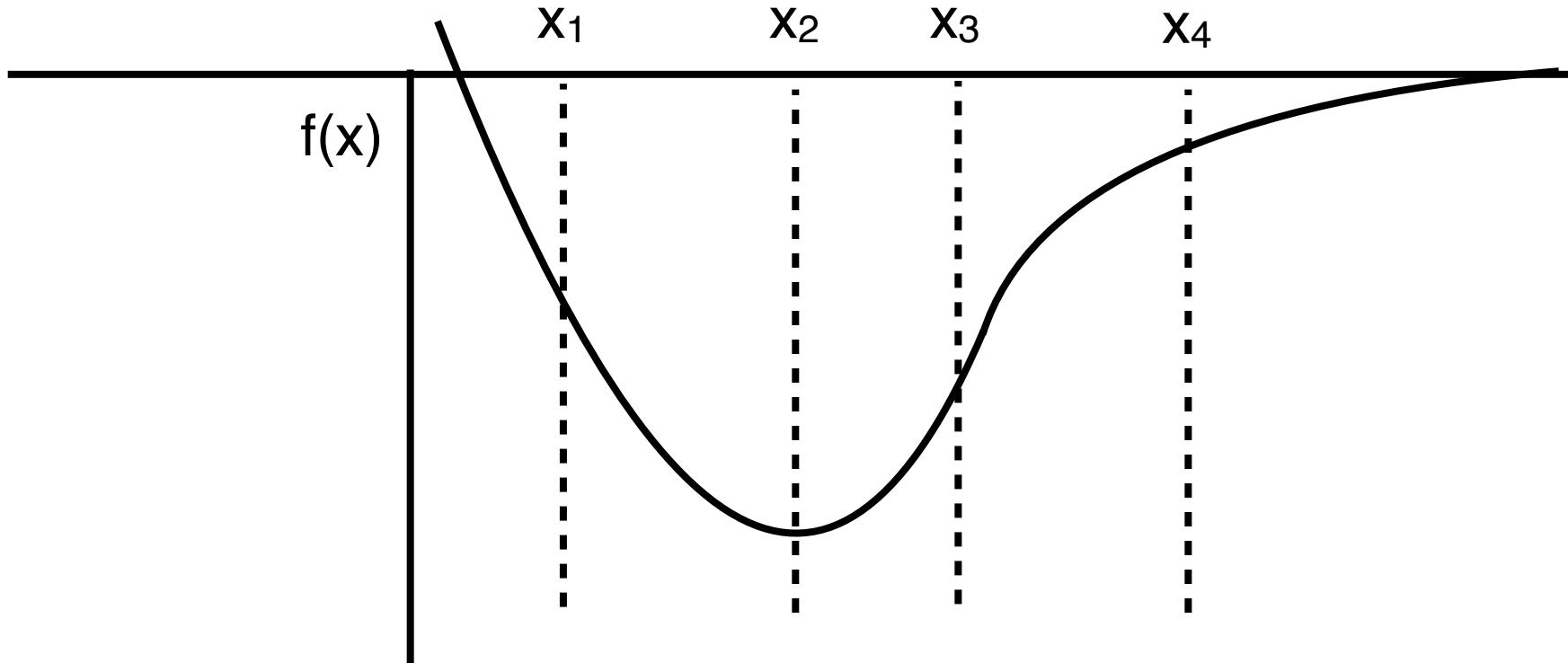
Note that we may be interested in the global maximum or a local maximum, and sometimes we may not know which we have found!

# Minimization and maximization problems



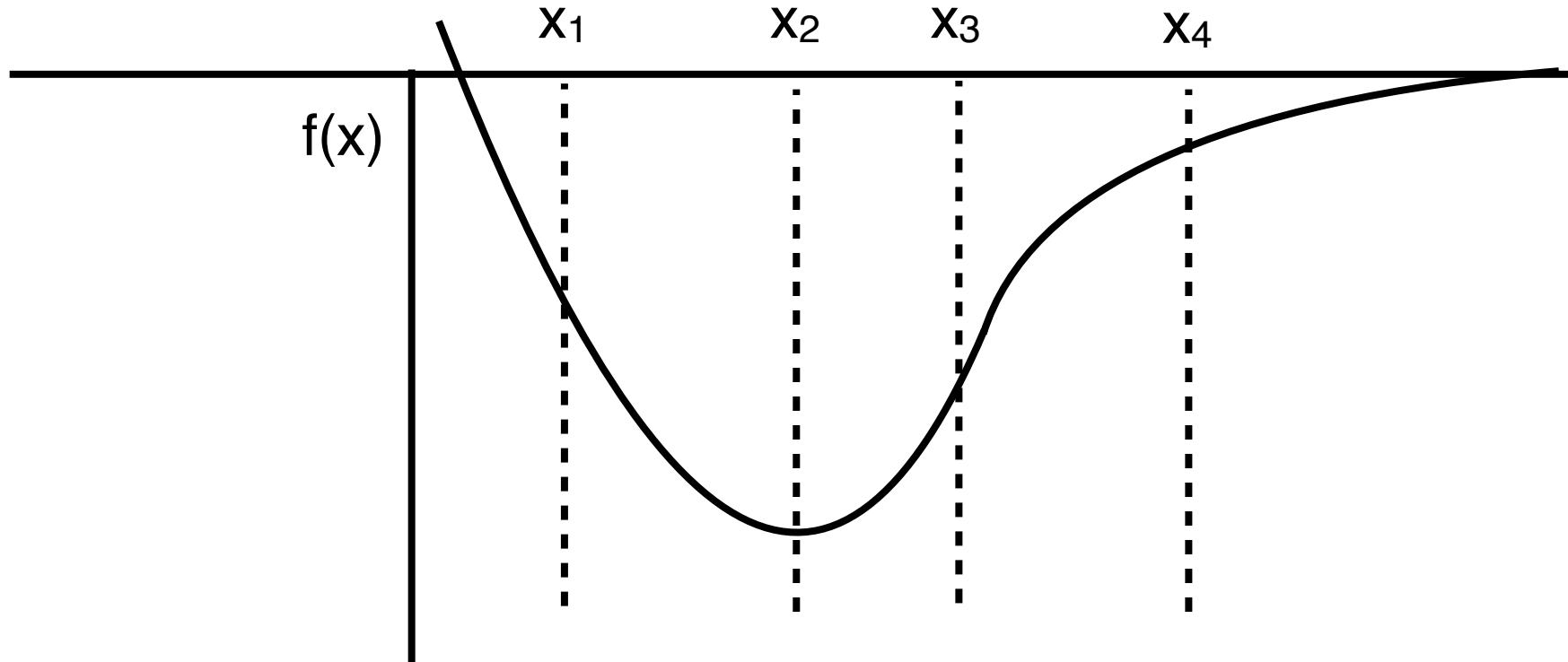
One thing that  
**differentiates** maxima/  
minima from all other points  
is that the derivative at that  
point is zero

## Golden ratio search



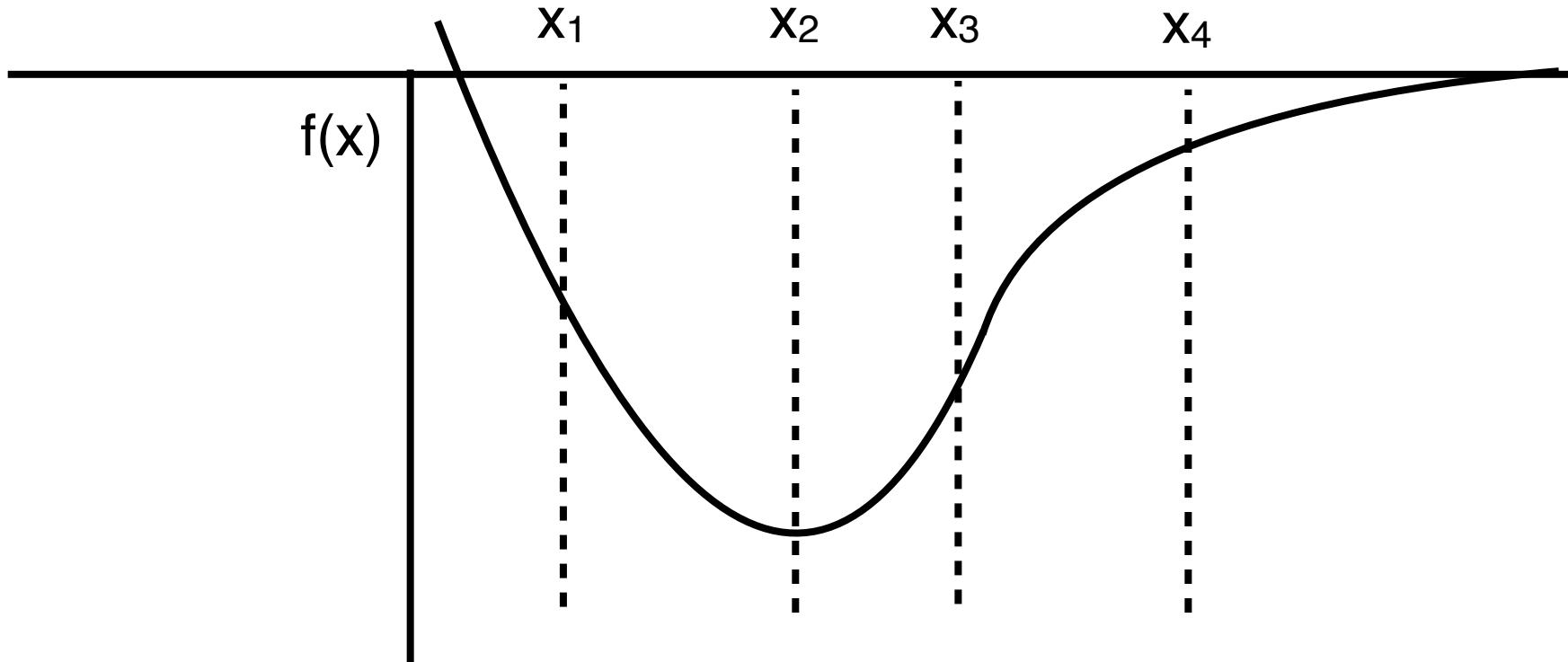
Want to find the (local) minimum. We choose 4 points to start. Pick  $x_1$  and  $x_4$  such that we expect to find a minimum between them

## Golden ratio search



We check that at least  $f(x_2)$  or  $f(x_3)$  is less than  $f(x_1)$  and  $f(x_4)$ . In that case there must be a minimum between  $x_1$  and  $x_4$  since the function goes down and then back up

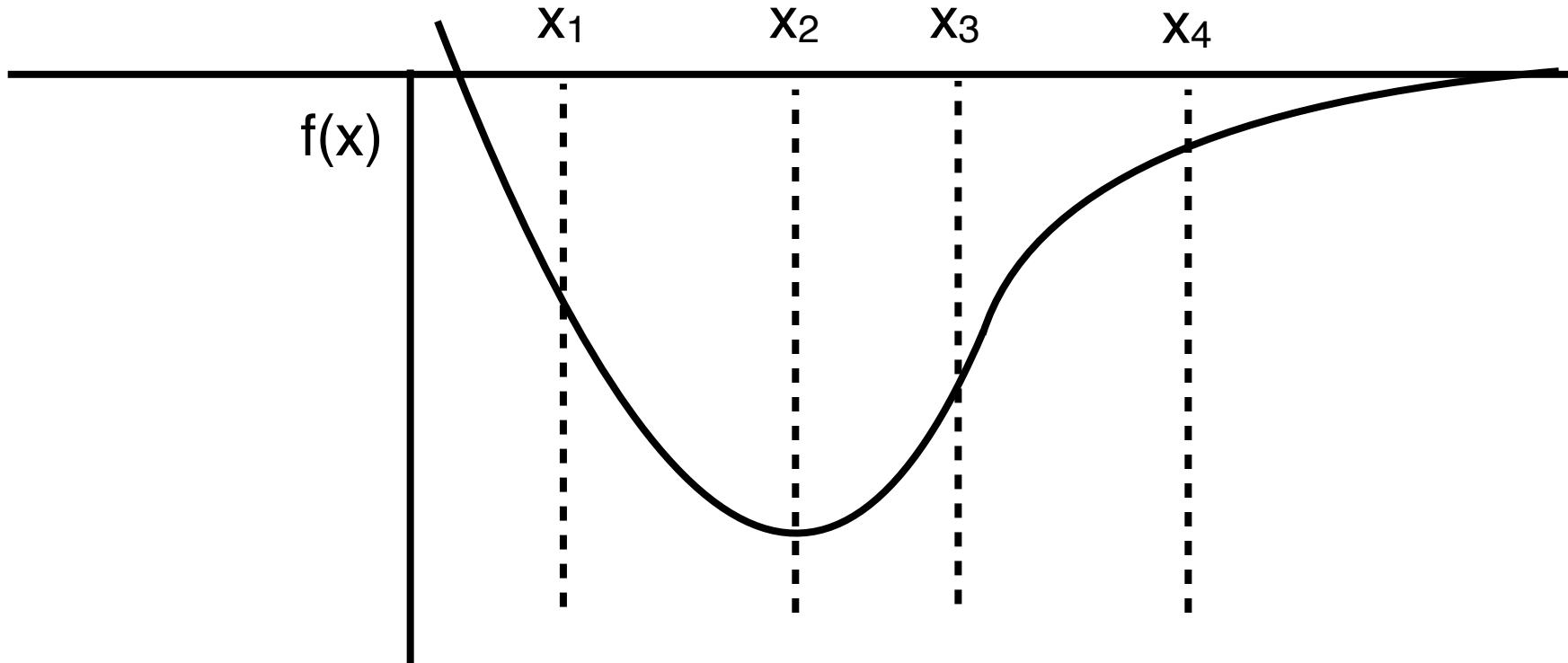
## Golden ratio search



If  $f(x_2)$  is smaller than  $f(x_3)$  then we know that the minimum is between  $x_1$  and  $x_3$ , because it still goes down and up between those points.

Otherwise the function minimum must be between  $x_2$  and  $x_4$  since it goes down and up in that range instead

## Golden ratio search

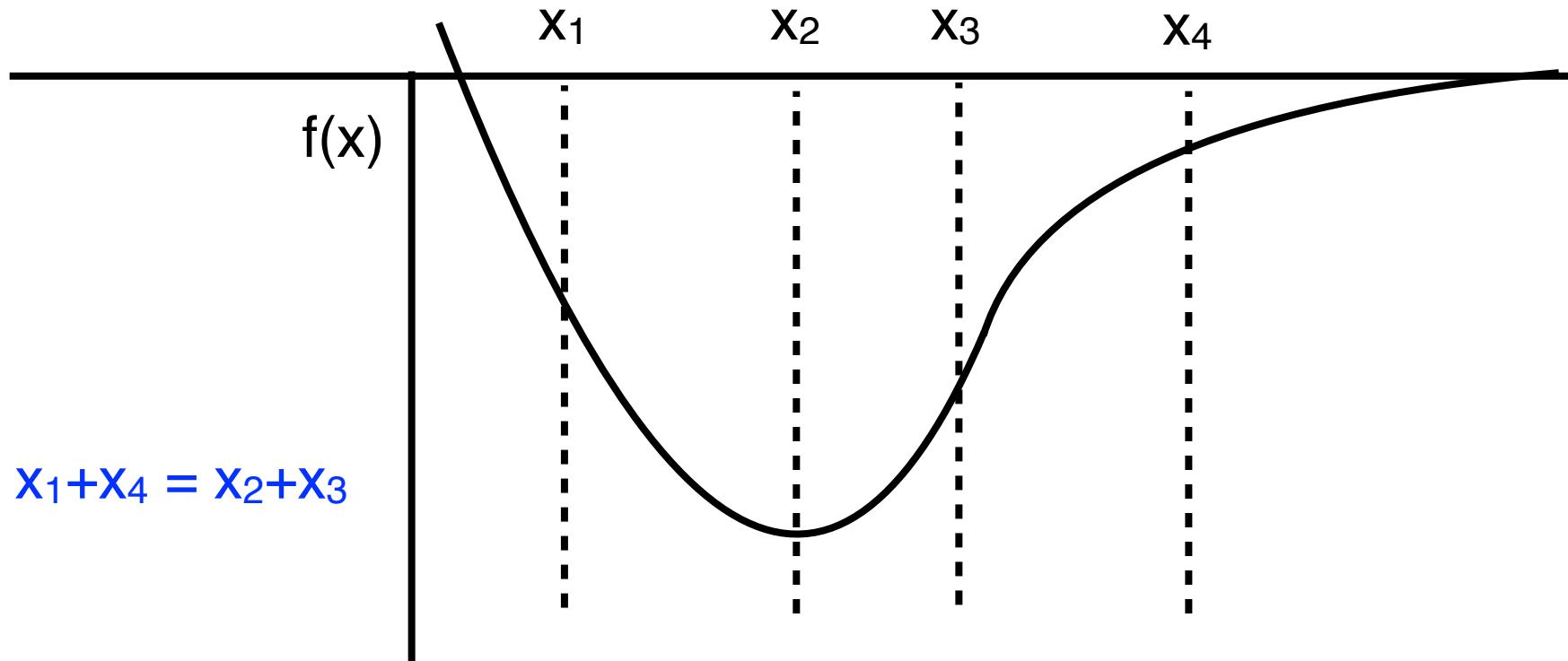


We try to put  $x_2$  and  $x_3$  the same distance from the midpoint of  $x_1$  and  $x_4$  (we don't know any better which side to prefer). Midpoint is  $(x_1+x_4)/2$ , so:

$$(x_1+x_4)/2 - x_2 = x_3 - (x_1+x_4)/2$$

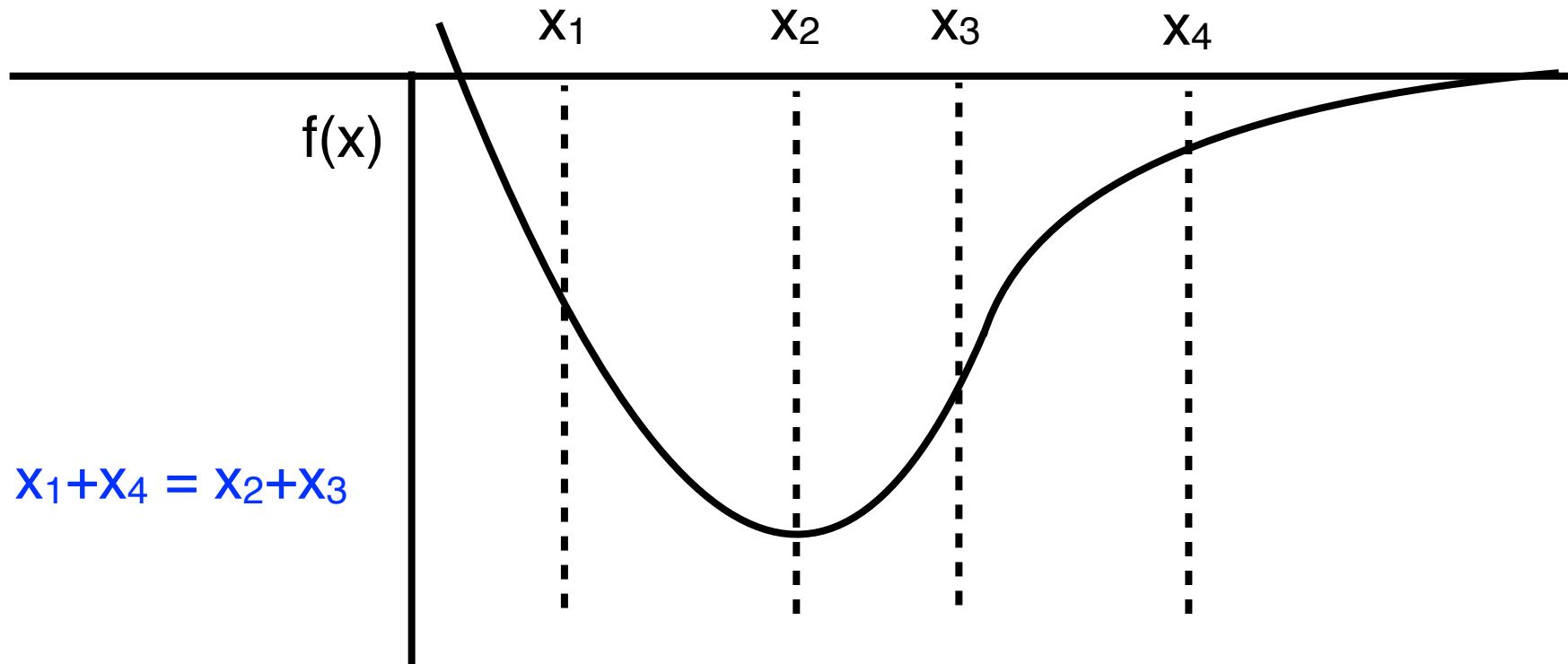
$$x_1+x_4 = x_2+x_3$$

## Golden ratio search



How much do we narrow down the range in each iteration? We start with a range  $x_4-x_1$  and we end up with an interval  $x_3-x_1$  (or  $x_4-x_2$ , which is the same size from above). So the ratio is  $z=(x_4-x_1)/(x_3-x_1)$ . Using the above this is  $(x_2+x_3-x_1-x_1)/(x_3-x_1) = (x_2-x_1)/(x_3-x_1)+1$

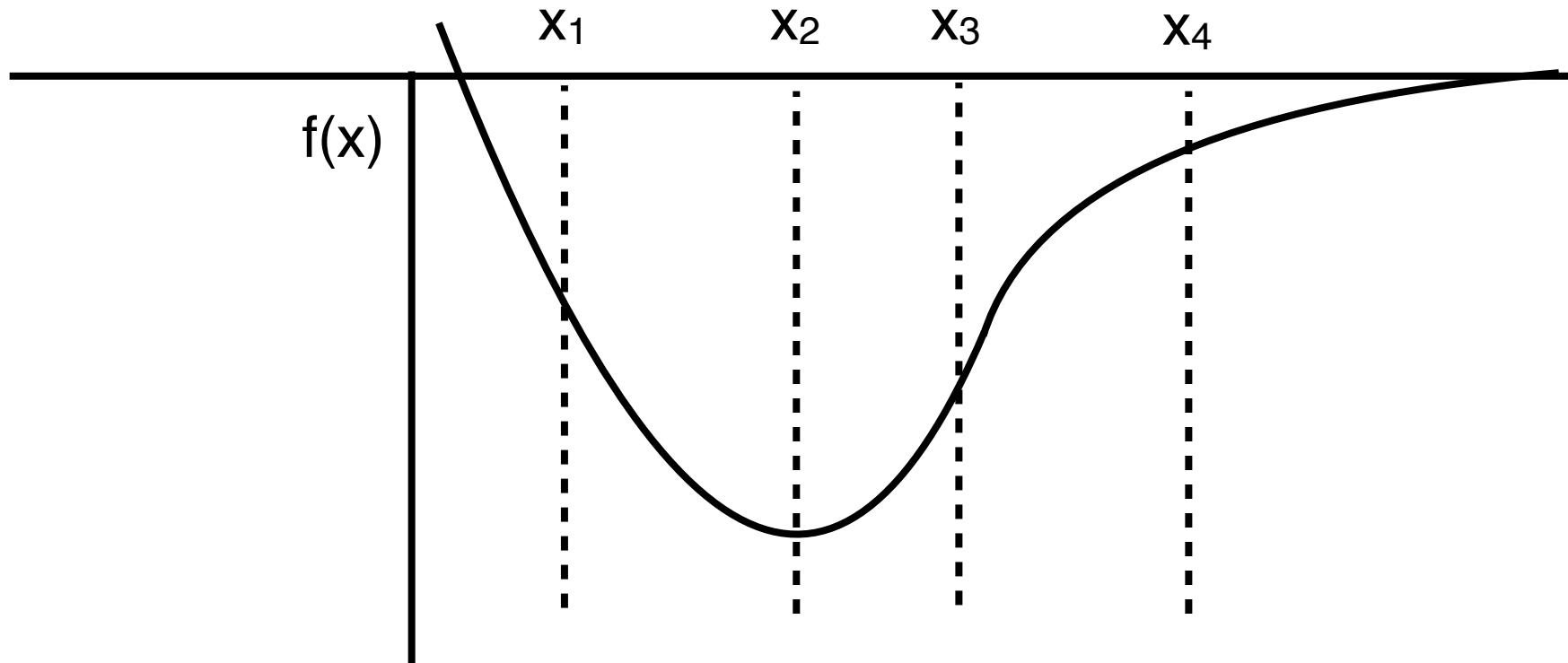
# Golden ratio search



So we zoom in by a ratio of  $z = (x_2-x_1)/(x_3-x_1)+1$   
 In our next iteration,  $z = (x_3-x_1)/(x_2-x_1)$ . But if we want  
 to keep “zooming in” on the answer at the same rate,  
 these are equal:

$$z = (x_2-x_1)/(x_3-x_1)+1 = (x_3-x_1)/(x_2-x_1)$$

## Golden ratio search



$$z = (x_2 - x_1) / (x_3 - x_1) + 1 = (x_3 - x_1) / (x_2 - x_1)$$

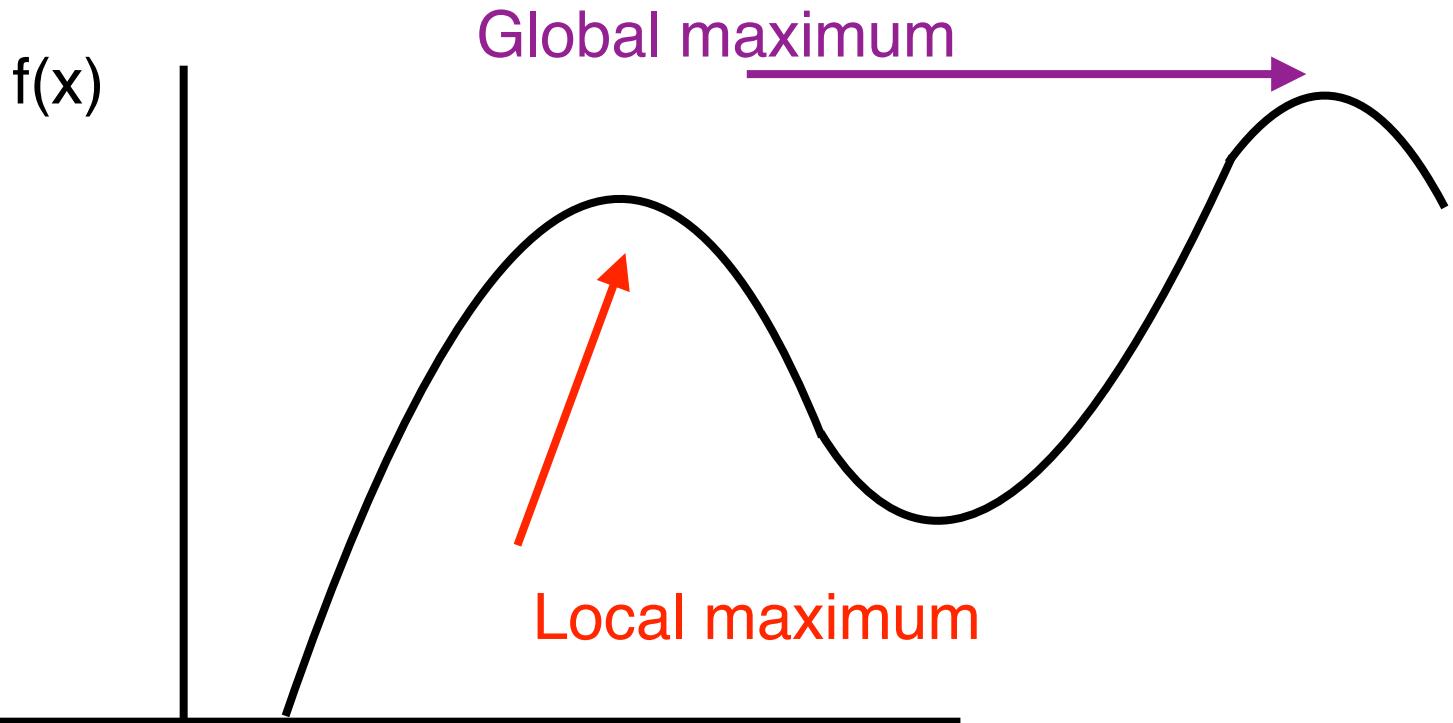
$$1/z + 1 = z, \text{ or } 1 + z = z^2$$

$z^2 - z - 1 = 0$ ,  $z = [1 + \sqrt{5}] / 2 = 1.618$ , the golden ratio.

We don't have to use this to set points, it's just optimal to do so. And we can use it to find new smaller windows until our window size is small enough that we have our answer

# Gauss-Newton Method and Gradient Descent

59



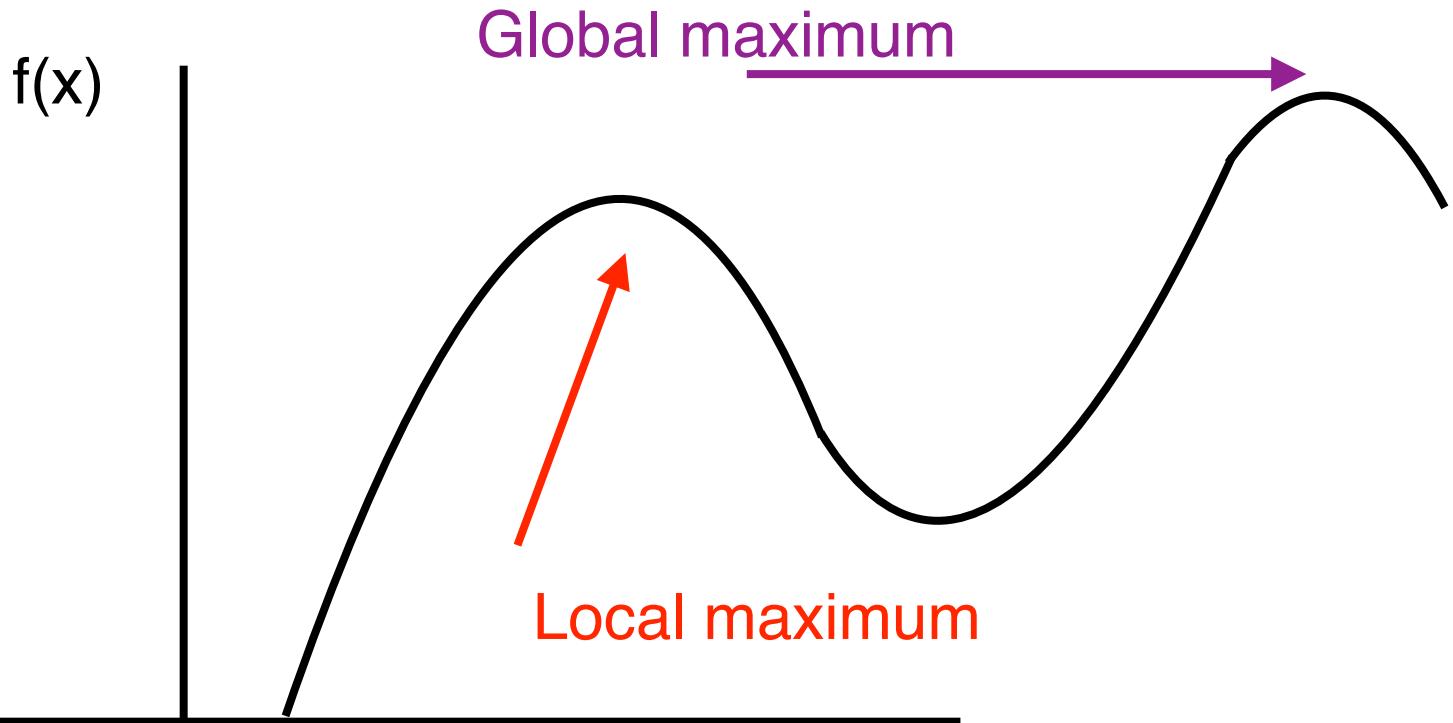
Can find a minimum or a maximum by locating a point where  $f'(x) = 0$ . But we know how to solve  $g(x) = f'(x) = 0$  using the Newton's method! Start with one guess and then

$$x' = x - \Delta x = x - g(x)/g'(x) = x - f'(x)/f''(x)$$

**Challenge: what is  $f''(x)$ ?** Solution: Call it a constant, ie  $\gamma$  so

$$x' = x - \gamma f'(x)$$

# Choice of sign in Gradient Descent



$x' = x - \gamma f'(x)$ , if  $\gamma$  is positive we move toward the minimum of a function (let's check this above), if  $\gamma$  is negative we move toward the maximum of a function (let's check in the picture above). If  $\gamma$  is too small in absolute value, convergence is slow, if it's too big we may overshoot or miss the min/max value. And if we can't calculate  $f'(x)$  we can estimate it as we have in the past

# Homework #4

From textbook: 6.7, 6.13, 6.18. Also, for 510 students,  
problem 6.9