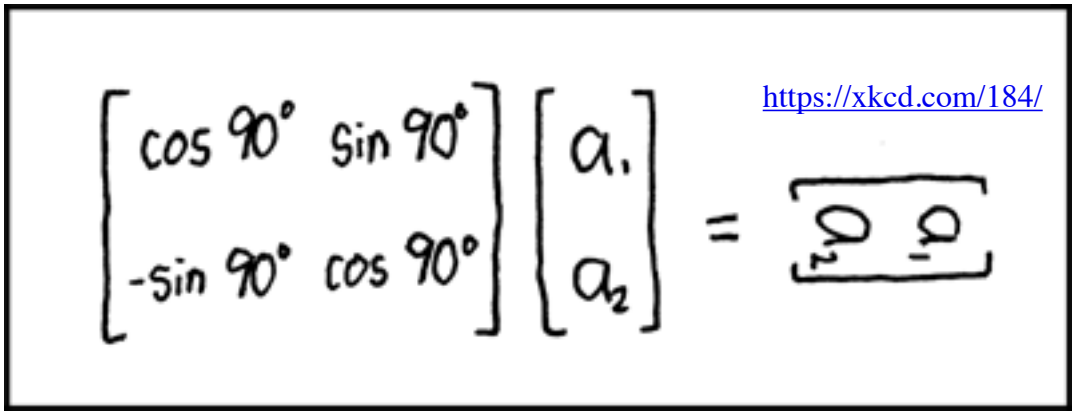
On to linear algebra

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

On to linear algebra

We will make good use of numpy here



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:

Ax = 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 matrix where the 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 (just swapping equations, it's OK!)
2) Multiply a row by a non-zero number (ex: if x=2, 2x=4!)
3) Add a multiple of one row to another (ex: if x=2 and y=3, x+y = 5)

Once things are in the right form, we back-substitute to solve our system of equations. The correct form is an upper triangular matrix. Let's think why that is

Gaussian Elimination example

System of equations	Row operations	Augmented matrix
$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$		$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$
$\begin{aligned} 2x + y - z &= 8 \\ \tfrac{1}{2}y + \tfrac{1}{2}z &= 1 \\ 2y + z &= 5 \end{aligned}$	$\begin{aligned} L_2 + \tfrac{3}{2}L_1 &\to L_2 \\ L_3 + L_1 &\to L_3 \end{aligned}$	$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right]$
$\begin{aligned} 2x + y - z &= 8 \\ \tfrac{1}{2}y + \tfrac{1}{2}z &= 1 \\ -z &= 1 \end{aligned}$	$L_3 + -4L_2 \to L_3$	$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right]$
The matrix is now in echelon form (also called triangular form)		
$\begin{aligned} 2x + y &= 7 \\ \tfrac{1}{2}y &= \tfrac{3}{2} \\ -z &= 1 \end{aligned}$	$\begin{aligned} L_2 + \tfrac{1}{2}L_3 &\to L_2 \\ L_1 - L_3 &\to L_1 \end{aligned}$	$\left[\begin{array}{ccc\|c} 2 & 1 & 0 & 7 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array}\right]$
$\begin{aligned} 2x + y &= 7 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} 2L_2 &\to L_2 \\ -L_3 &\to L_3 \end{aligned}$	$\left[\begin{array}{ccc\|c} 2 & 1 & 0 & 7 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$
$\begin{aligned} x &= 2 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} L_1 - L_2 &\to L_1 \\ \tfrac{1}{2}L_1 &\to L_1 \end{aligned}$	$\left[\begin{array}{ccc\|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$

https://en.wikipedia.org/wiki/Gaussian_elimination

Once we have an upper triangular form like this one, the full solution is trivial! The last row gives the last variable, then we use that in the row above it, giving the second to last variable, and so on … This is **backsubstitution**

Let's check Example 6.1


```
from numpy import array,empty
A = array([[2,1,4,1],
           [3,4,-1,-1],
           [1,-4,1,5],
           [2,-2,1,3]],float)
v = array([-4,3,9,7],float)
N = len(v)

### Solve Ax = v, ie find the unknown column vector x
### Use Gaussian elimination!

for m in range(N):
  # Divide by the diagonal element first to set it to 1
  div = A[m,m] ### if this is zero, we will break things!
  A[m,:] /= div ### divide everything in the row of this array
  v[m] /= div ### divide v

  ### We want to get zeros below this diagonal element, to do this we:
  ### 1) Multiply the row we just looked at above (which starts with 1 now)
  ###    by the value of the first element in this row below it
  ### 2) Subtract that value from this new row below it, giving us zero
  for i in range(m+1,N):
    mult = A[i,m] ### for each row i below the one above, column is still the same column as above
    A[i,:] -= mult*A[m,:] ### subtract from this row the multiplicative factor above
    v[i] -= mult*v[m] ### do the same for v so equations are consistent

### Done! Now backsubstitute
x = empty(N,float)
for m in range(N-1, -1, -1): ### go in reverse order
  x[m] = v[m] ### we have x now, it's just equal to v
  for i in range (m+1, N): ### we have to update the remaining pieces
    x[m] -= A[m,i]*x[i]

print(x)
```

```
[ 2. -1. -2.  1.]
```

Let's go over this in detail together

Partial pivoting

Can try to write an automatic program to do this, but…. Need to make sure that we don't divide by zero! This can occur in Gaussian elimination 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 ith row operation, search for the element a_{ki} (k=i,...n-1) with the largest magnitude
2) If k != i, interchange rows i and k of the augmented matrix, and interchange $x_i \leftrightarrow x_k$
3) Continue

Essentially, divide by the largest remaining value in Gaussian elimination, ensuring that we divide by a value as far from zero as possible

Partial pivoting implementation (Ex 6.2)


```
from numpy import array,empty

### Solve Ax = v, ie find the unknown column vector x
def partialPivoter(A,v):
  print("A=",A)
  print("v =",v)

  N = len(v)
  for m in range(N):

    # Before any Gaussian elimination find the row with the largest pivot element
    pivot = m
    largest = A[m,m]
    for i in range(m+1,N):
      if abs(A[i,m]) > largest:
        largest = abs(A[i,m])
        pivot = i

    ## we found the pivot, so now swap rows, do it entry-by-entry across columns
    for i in range(N):
      A[m,i],A[pivot,i] = A[pivot,i],A[m,i]

    # and swap the row in v too!
    v[m],v[pivot] = v[pivot],v[m]

    # Now run Gaussian elimination!
    # Divide by the diagonal element first to set it to 1
    div = A[m,m] ### if this is zero, we will break things!
    A[m,:] /= div ### divide everything in the row of this array
    v[m] /= div ### divide v

    ### We want to get zeros below this diagonal element, to do this we:
    ### 1) Multiply the row we just looked at above (which starts with 1 now)
    ###    by the value of the first element in this row below it
    ### 2) Subtract that value from this new row below it, giving us zero
    for i in range(m+1,N):
      mult = A[i,m] ### for each row i below the one above, column is still the same column as above
      A[i,:] -= mult*A[m,:] ### subtract from this row the multiplicative factor above
      v[i] -= mult*v[m] ### do the same for v so equations are consistent

  ### Done! Now backsubstitute
  x = empty(N,float)
  for m in range(N-1, -1, -1): ### go in reverse order
    x[m] = v[m] ### we have x now, it's just equal to v
    for i in range (m+1, N): ### we have to update the remaining pieces
      x[m] -= A[m,i]*x[i]
  print("Ax = v has solution x = ",x)
  return x

A = array([[2,1,4,1],
           [3,4,-1,-1],
           [1,-4,1,5],
           [2,-2,1,3]],float)
v = array([-4,3,9,7],float)

x = partialPivoter(A,v)

A = array([[0,1,4,1],
           [3,4,-1,-1],
           [1,-4,1,5],
           [2,-2,1,3]],float)
v = array([-4,3,9,7],float)
x = partialPivoter(A,v)
```

```
A= [[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
v = [-4.  3.  9.  7.]
Ax = v has solution x =  [ 2. -1. -2.  1.]
A= [[ 0.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
v = [-4.  3.  9.  7.]
Ax = v has solution x =  [ 1.61904762 -0.42857143 -1.23809524  1.38095238]
```

Second example would have failed without pivoting

LU factorization

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

Ax = b

we rewrite **A = LU**, where **L** is a lower triangular matrix and **U** is an upper triangular matrix. Then:

LUx = b

and we can rewrite **Ux = y**

so that **Ly=b**

Since L is in triangular form, this is easy to solve and then we solve again **Ux=y** (**U** is also triangular!)

Why do we do this? If we change **b→c**, then we can still solve the Gaussian elimination on **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

$$\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

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

$$\begin{pmatrix} \frac{a_{00}}{a_{00}} & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

Tridiagonal matrices

$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ 0 & a_{11} - \frac{a_{10}a_{01}}{a_{00}} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ b_1 - \frac{a_{10}b_0}{a_{00}} \\ b_2 \\ \mathsf{b}_3 \\ \mathsf{b}_4 \end{pmatrix}$$

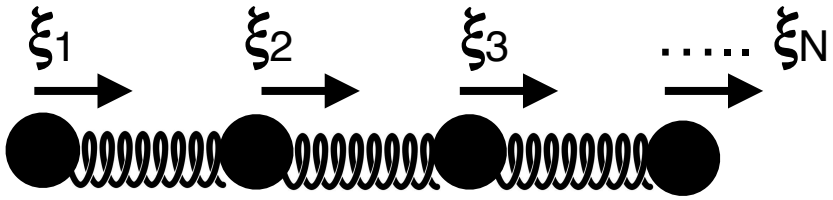
$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ 0 & 1 & \frac{a_{12}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ \frac{b_1 - \frac{a_{10}b_0}{a_{00}}}{a_{11} - \frac{a_{10}a_{01}}{a_{00}}} \\ b_2 \\ \mathsf{b}_3 \\ \mathsf{b}_4 \end{pmatrix}$$

Tridiagonal matrices

$$\begin{pmatrix} 1 & \frac{a_{01}}{a_{00}} & 0 & 0 & 0 \\ 0 & 1 & \frac{a_{12}}{a_{11}-\frac{a_{10}a_{01}}{a_{00}}} & 0 & 0 \\ 0 & 0 & a_{22}-\frac{a_{21}a_{12}}{a_{11}-\frac{a_{10}a_{01}}{a_{00}}} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \frac{b_0}{a_{00}} \\ \frac{b_1-\frac{a_{10}b_0}{a_{00}}}{a_{11}-\frac{a_{10}a_{01}}{a_{00}}} \\ b_2 - a_{21}\frac{b_1-\frac{a_{10}b_0}{a_{00}}}{a_{11}-\frac{a_{10}a_{01}}{a_{00}}} \\ b_3 \\ b_4 \end{pmatrix}$$

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)



Hooke's Law and F=ma

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

First Spring

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

Last Spring

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

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

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

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

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

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 Fi = 0 so that $\xi_i(t) = x_i e^{i\omega t}$, where x_i is the amplitude of the ith mass

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

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

First mass

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

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

Last mass

Our three equations

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

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

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

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

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

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

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

These define a tridiagonal set of simultaneous equations!

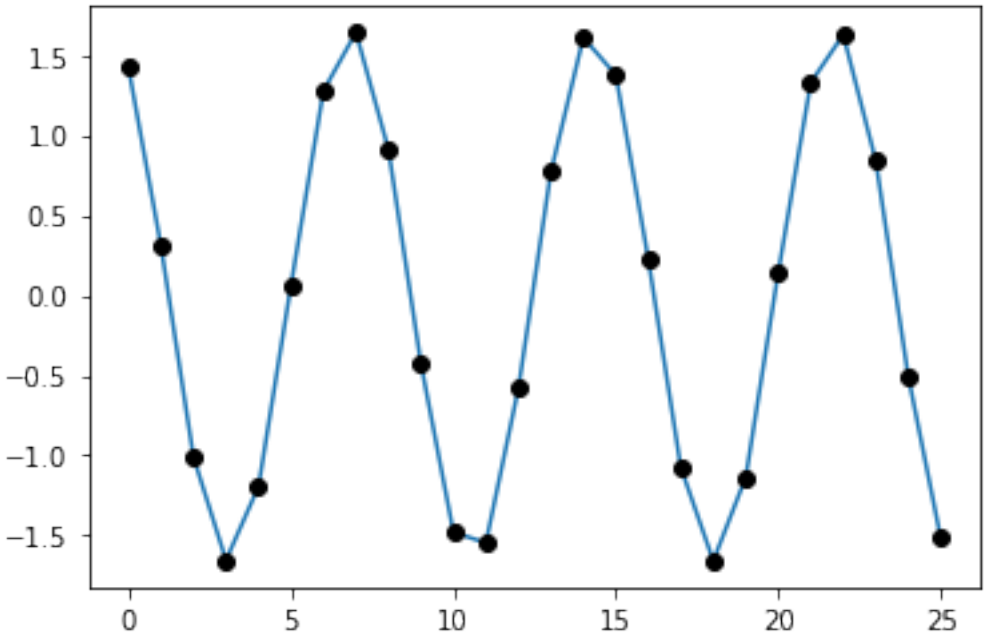
From the book's appendix and online tools


```
#####################################################################
#
# 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 intiail 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()
```



The banded inputs from the textbook

Here is our original banded matrix (tridiagonal or otherwise)

```
[['a00' 'a01' 'a02' '000' '000']
 ['a10' 'a11' 'a12' 'a13' '000']
 ['000' 'a21' 'a22' 'a23' 'a24']
 ['000' '000' 'a32' 'a33' 'a34']
 ['000' '000' '000' 'a43' 'a44']]
```

Let's look closely at the input to the banded program

```
[['---' '---' 'a02' 'a13' 'a24']
 ['---' 'a01' 'a12' 'a23' 'a34']
 ['a00' 'a11' 'a22' 'a33' 'a44']
 ['a10' 'a21' 'a32' 'a33' 'a43']]
```

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

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

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

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

```
# Set up the intiail 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
```

Inverting matrices, as in the book


```
import numpy as np
from numpy.linalg import inv
from numpy import matmul
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)
print("A*A-1 = ",matmul(A,invA))
```

Be careful about multiplying matrices. Nominally just using A*B for numpy matrices isn't matrix multiplication - you need to use matmul for that!

Inverting matrices, as in the book


```
A=
 [[ 1.  7. 10.  0.  3.]
 [ 3.  2.  4.  3.  5.]
 [ 5.  1. 10.  1.  9.]
 [ 7. 10.  7.  4.  0.]
 [ 7.  2.  8.  4.  6.]]
A inverse =
 [[-0.21538462 -0.18461538  0.24615385  0.18461538 -0.10769231]
 [-0.06093514  0.17466063  0.13966817  0.1586727  -0.32458522]
 [ 0.1984917  -0.18280543 -0.18763198 -0.1505279   0.33453997]
 [ 0.18190045  0.20633484 -0.45158371 -0.20633484  0.41447964]
 [-0.11432881  0.26334842  0.21749623  0.06998492 -0.32187029]]
A*A-1 =  [[ 1.00000000e+00 -5.55111512e-17 -5.55111512e-17  1.66533454e-16
   2.77555756e-16]
 [ 0.00000000e+00  1.00000000e+00  1.66533454e-16 -1.66533454e-16
   1.66533454e-16]
 [-2.22044605e-16  5.55111512e-17  1.00000000e+00 -5.55111512e-17
   1.66533454e-16]
 [ 3.33066907e-16  0.00000000e+00 -2.22044605e-16  1.00000000e+00
   0.00000000e+00]
 [ 1.11022302e-16 -1.11022302e-16 -3.33066907e-16 -5.55111512e-16
   1.00000000e+00]]
```

Be careful about rounding!

How to do DQ decomposition to find eigenvalues and eigenvectors

Trying to solve eigenvalue equation $\mathbf{Av} = \lambda\mathbf{v}$. Note that if we have an NxN matrix **A**, the eigenvectors are orthonormal and there are N of them, each with an associated eigenvalue.

We can rewrite the above as
$\mathbf{AV} = \mathbf{VD}$, where **D** is a diagonal matrix with the eigenvalues as entries and **V** is an NxN matrix where each column is an eigenvector

How do we solve for **V** and **D**?

How to do DQ decomposition to find eigenvalues and eigenvectors

The goal is to rewrite the matrix **A** as the product of an orthogonal matrix (**Q**) and an upper-triangular matrix (**R**). The orthogonal matrix is one where the columns and rows are orthonormal vectors. Why do this?

$$\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1 \text{ but then:}$$

$$\mathbf{Q}_1^T\mathbf{A} = \mathbf{Q}_1^T\mathbf{Q}_1\mathbf{R}_1 = \mathbf{R}_1$$

now define:

$$\mathbf{A}_1 = \mathbf{R}_1\mathbf{Q}_1 = \mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1$$

and then define:

$$\mathbf{A}_2 = \mathbf{R}_2\mathbf{Q}_2 = \mathbf{Q}_2^T\mathbf{A}_1\mathbf{Q}_2 = \mathbf{Q}_2^T\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2$$

$$\ldots\ldots \mathbf{A}_k = (\mathbf{Q}_k^T\ldots\mathbf{Q}_1^T)\mathbf{A}(\mathbf{Q}_1\ldots\mathbf{Q}_k)$$

Repeat this until the off-diagonal elements are small enough that they approach zero (or some epsilon)! Then $\mathbf{A}_k$ is a diagonal matrix

How to do DQ decomposition to find eigenvalues and eigenvectors

$$\mathbf{A_k = (Q_k^T \ldots . Q_1^T) A (Q_1 \ldots Q_k)}$$

but if we defined $\mathbf{V = (Q_1 \ldots Q_k)}$ then $\mathbf{A_k = V^T A V}$

so $\mathbf{V A_k = V V^T A V = A V}$, or $\mathbf{A V = V A_k = V D!}$

OK, so how do we do this QR decomposition (which the computer will then repeat over and over again until the off-diagonal elements are small enough that we stop)?

How to do DQ decomposition to find eigenvalues and eigenvectors

The goal is to rewrite the matrix **A** 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 $\mathbf{a_0}$, $\mathbf{a_1}$, … $\mathbf{a_{N-1}}$

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

Define new vectors $\mathbf{u_i}$

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

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

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

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

Reminder:
Trying to solve $\mathbf{Av} = \lambda\mathbf{v}$

How to do DQ decomposition

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

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

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

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

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

Want to prove orthonormality, ie:

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

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

$$0 \leq k < i$$

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

How to do DQ decomposition

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

$$0 \leq k < i$$

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

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

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

$$\mathbf{q_i} \cdot \mathbf{q_k} = \frac{1}{|\mathbf{u_i}|} \left[\mathbf{a_i} \cdot \mathbf{q_k} - (\mathbf{q_k} \cdot \mathbf{a_i})(\mathbf{q_k} \cdot \mathbf{q_k}) \right]$$

How to do DQ decomposition

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

$$0 \leq k < i$$

$$\mathbf{q_i} \cdot \mathbf{q_k} = \frac{1}{|\mathbf{u_i}|} \left[\mathbf{a_i} \cdot \mathbf{q_k} - (\mathbf{q_k} \cdot \mathbf{a_i})(\mathbf{q_k} \cdot \mathbf{q_k}) \right]$$

$$\mathbf{q_i} \cdot \mathbf{q_k} = \frac{1}{|\mathbf{u_i}|} \left[\mathbf{a_i} \cdot \mathbf{q_k} - (\mathbf{q_k} \cdot \mathbf{a_i}) \right]$$

$$\mathbf{q_i} \cdot \mathbf{q_k} = 0$$

$$\mathbf{q_i} \cdot \mathbf{q_i} = \frac{\mathbf{u_i}}{|\mathbf{u_i}|} \cdot \frac{\mathbf{u_i}}{|\mathbf{u_i}|} = \frac{|\mathbf{u_i}|^2}{|\mathbf{u_i}|^2} = 1$$

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 I$\mathbf{q_0}$I is orthonormal as a set, so we can use induction to prove that all of them are!

How to do DQ decomposition

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

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

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

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

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

Rewriting things:

$$\mathbf{a_0} = |\mathbf{u_0}|\mathbf{q_0}$$

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

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

Let's multiply it out and check this

A **Q** **R**

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

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
```

```
# 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)
```

```
[[ 2.10000000e+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]]
[[ 2.10000000e+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 -0.2581989   0.77459667]
 [ 0.62330228  0.52740965  0.25819889 -0.51639778]
 [ 0.52740965 -0.62330227  0.51639779  0.25819889]]
```

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^(0.5*x)+e^(0.4*x)
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))
```

```
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
2.914421788441389
2.9144217884413943
At the end x =  2.9144217884413943 and func =  2.914421788441389
```

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

$$x' = f(x) = f(x^*) + (x - x^*)f'(x) + ... = x^* + (x - x^*)f'(x^*) + ...$$

$$(x' - x^*) = (x - x^*)f'(x^*)$$

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))
```

```
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
2.718281828459045
5.143474962376301e-09
At the end x =  5.143474962376301e-09 and func =  2.718281828459045
```

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))
```

```
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
```

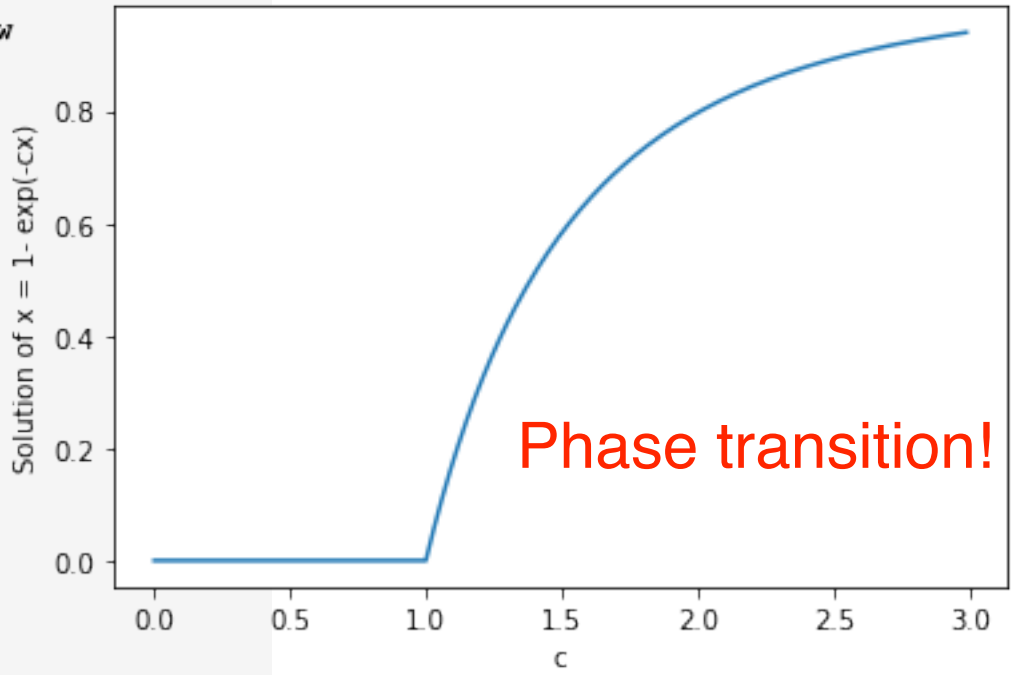
Sometimes a different form will have better slope and convergence

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

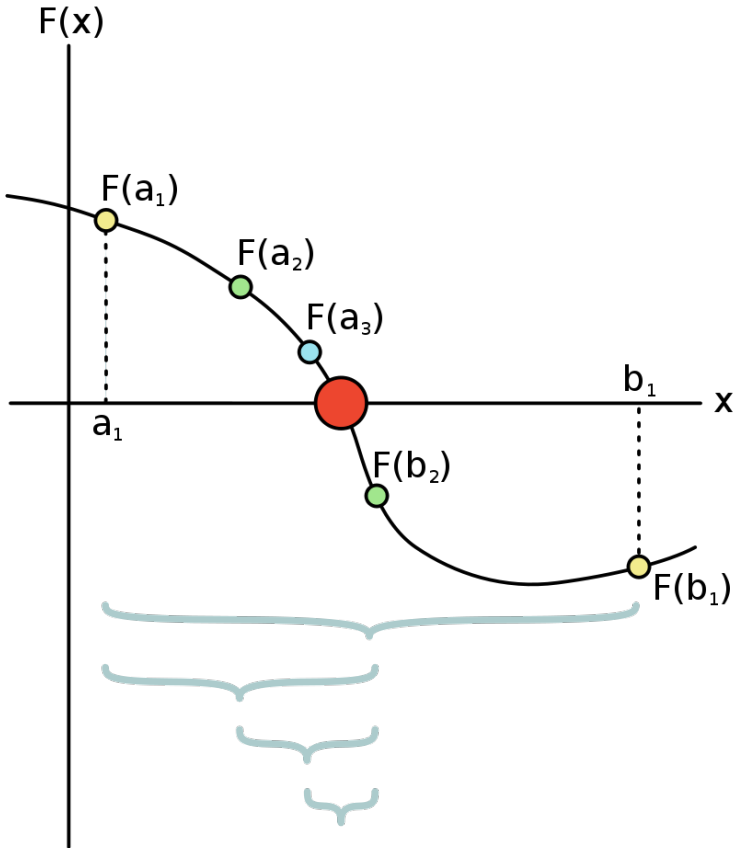
Quick check: 1=1^(1-1^3), yes!

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 cpoints:
  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

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

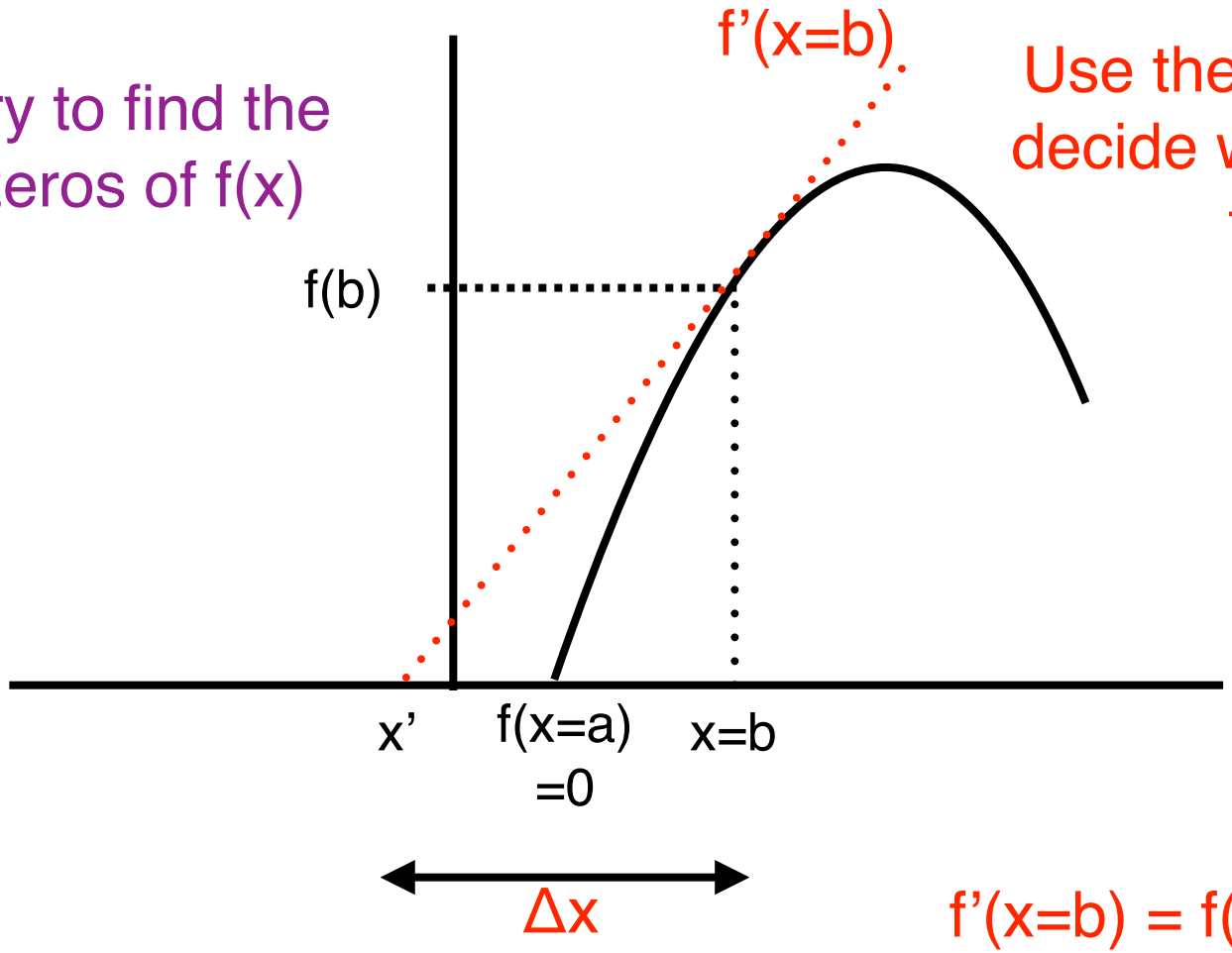

```
[13] from math import sin,exp,cos,log
     def f(x):
       return exp(1-x**3)-x ## x=exp(1-x**3)

     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)
```

```
It took 30 iterations  and midpoint =  1.0000000009313226
```

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

Newton's method

Try to find the zeros of f(x)



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

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

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

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

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

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

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

How fast do we converge on the solution?

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

Define errors on our estimates such that $x^* = x+\varepsilon = x' + \varepsilon'$ (x is our current estimate, x' is our next estimate)

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

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

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?

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

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

Current error is ε, after one iteration it is ε^2, after two iterations it is ε^4, after three iterations ε^8. After N iterations it will be $\varepsilon^{2\wedge N}$. 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))
```

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

VERY QUICK!

```
[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))
```

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

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

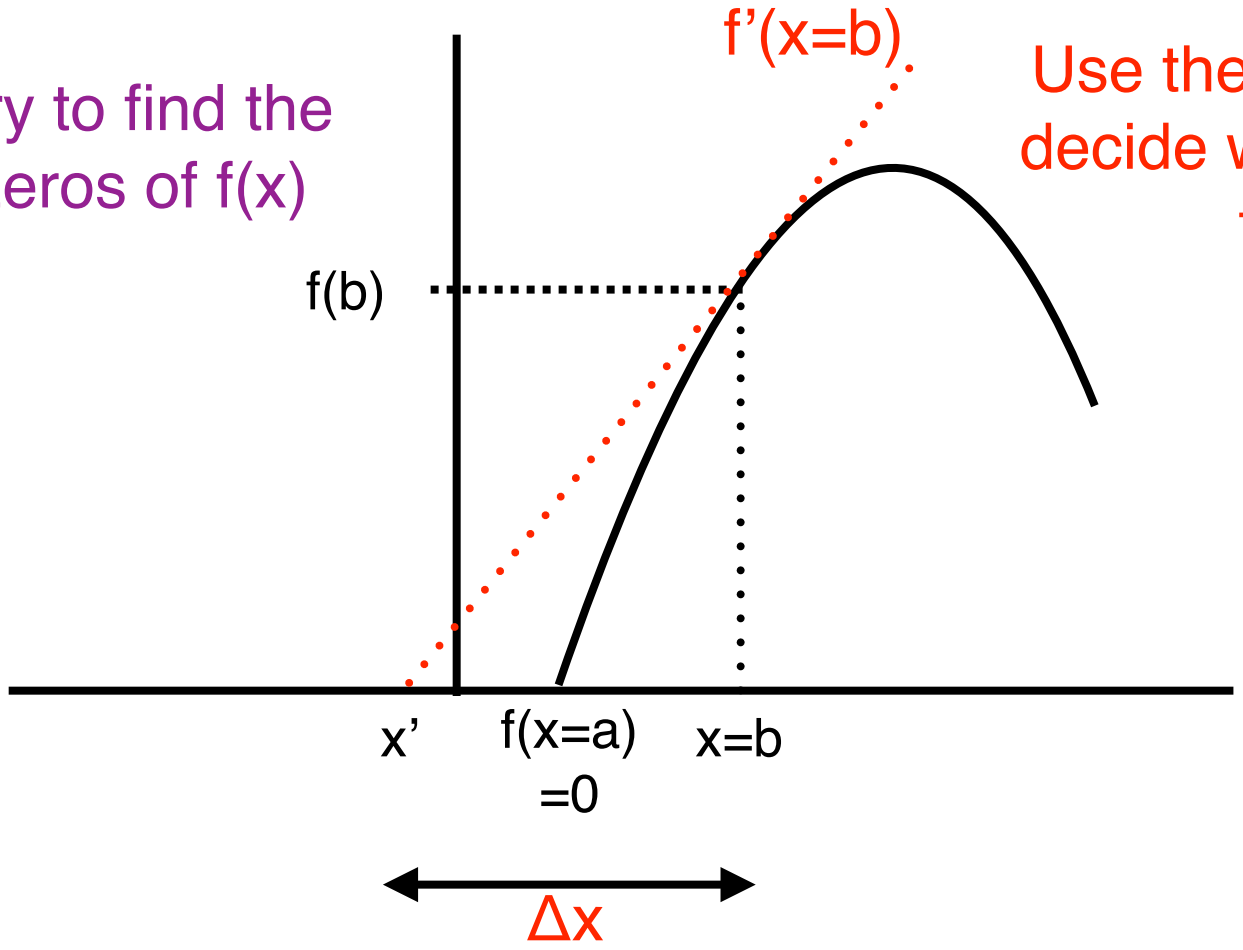
How do we use Newton's method?

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. Oftentimes, 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)



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

New guess:

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

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

Newton's method for two or more variables

We have not position x but
a vector $\mathbf{x} = (x_1, x_2, \ldots x_N)$
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 $\mathbf{f}(\mathbf{x}^*) = 0$

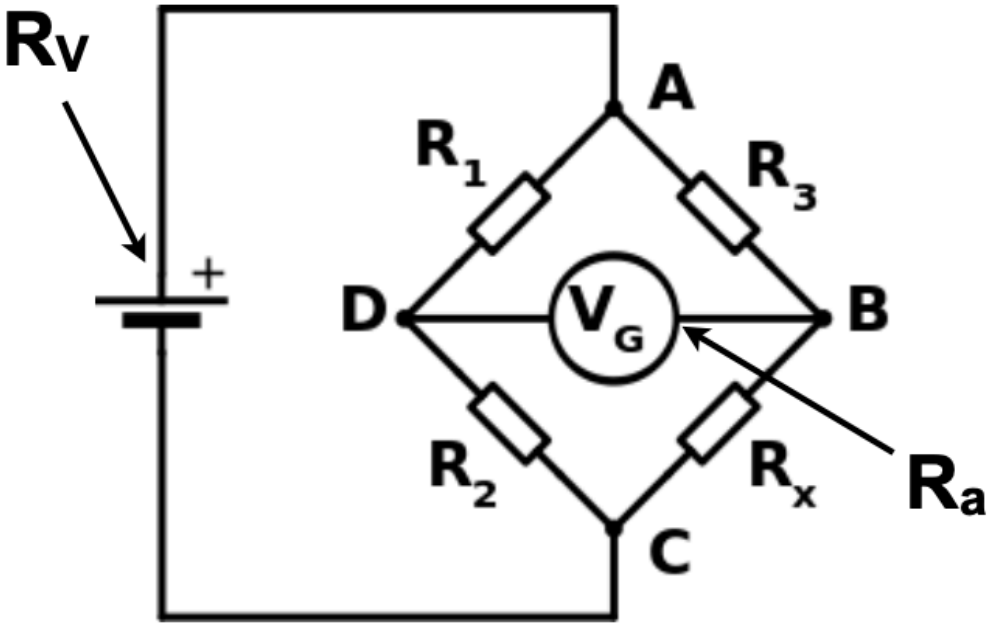
Need to solve system of equations!

$\mathbf{f}(\mathbf{x}^*) = \mathbf{f}(\mathbf{x}) + \mathbf{J}\cdot(\mathbf{x}^*-\mathbf{x}) + \ldots$
with **J** the Jacobian matrix,
$\mathbf{J}\cdot\Delta\mathbf{x} = f(x)$

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

Wheatstone bridge (from Sal)

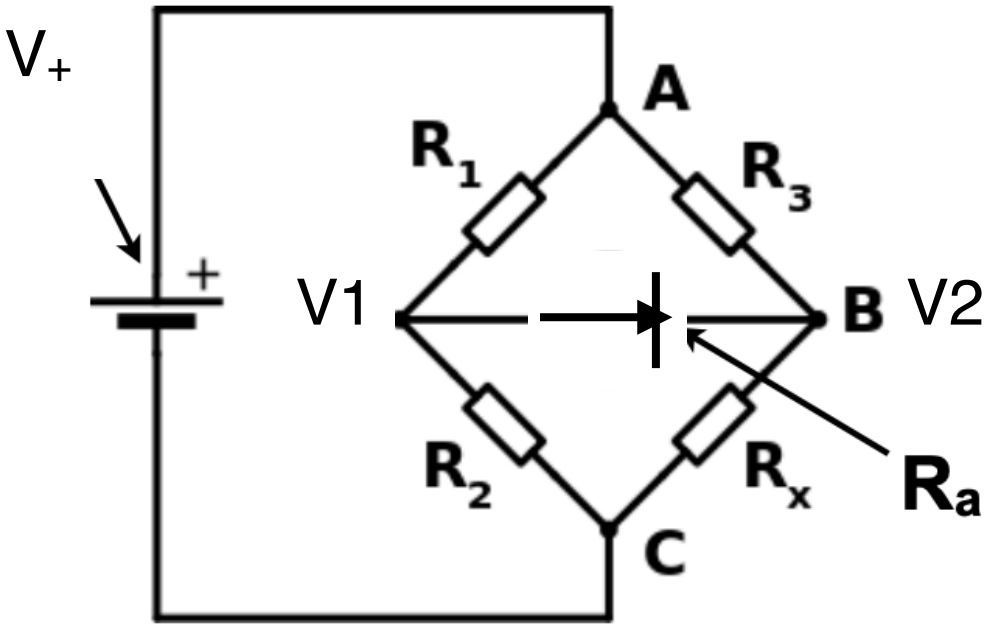
– http://en.wikipedia.org/wiki/Wheatstone_bridge



Solve for Rx given R1, R2, R3, i and V_G. Adjust R2 until V_G is zero, tells us Rx

At solution, Rx = (R2 / R1) R3

Exercise 6.17



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

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

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
```

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

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

```
while error > target:
  # Calculate matrix elements and determinant
  # Remember, this is just a 2x2 matrix with element that are the derivatives
  # Let's check that the derivatives look right!
  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
```

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

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

```
# Calculate vector elements

# this is the bug ugly piece
y = I0*(exp(((V1-V2)/VT)-1))

# Our two functions equal to zero
z1 = (V1-Vp)/R1 + V1/R2 + y
z2 = (V2-Vp)/R3 + V2/R4 - y

# Calculate delta x using the inverse
# Why? If J dot dx = f(x), then dx = J-1 f(x)
deltaV1 = ia*z1 + ib*z2
deltaV2 = ic*z1 + id*z2

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

# Calculate new voltages, remember the minus sign
V1 -= deltaV1
V2 -= deltaV2

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

```
3.466244495626595 2.8006332565601078
```

JWST (Ex 6.16)

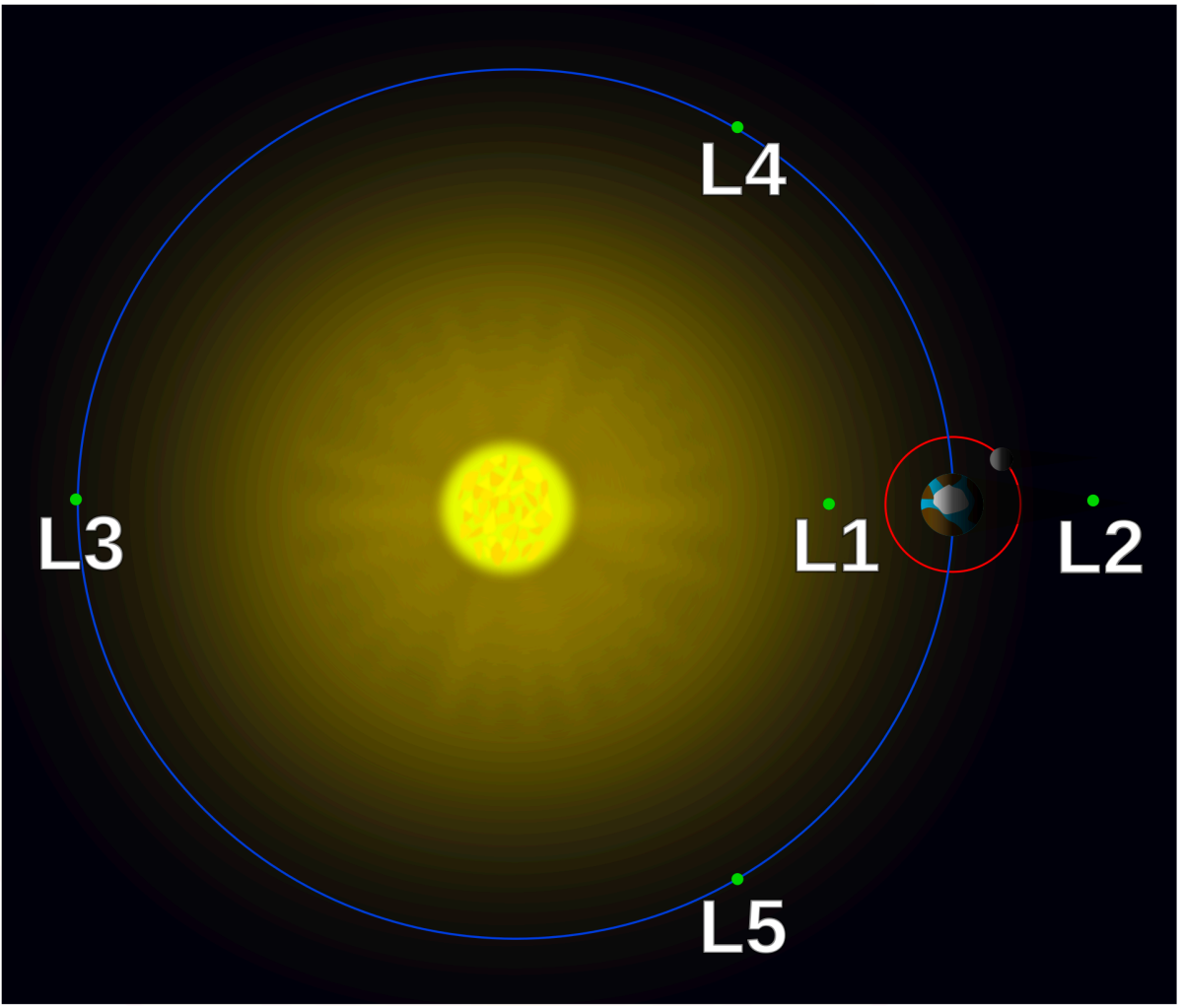
The Hubble Space Telescope is … old news!

Hello to the era of the James Web Space Telescope

JWST (6.16)

“Cosmic Cliffs” of the Carina Nebula - one of JWST’s first photos

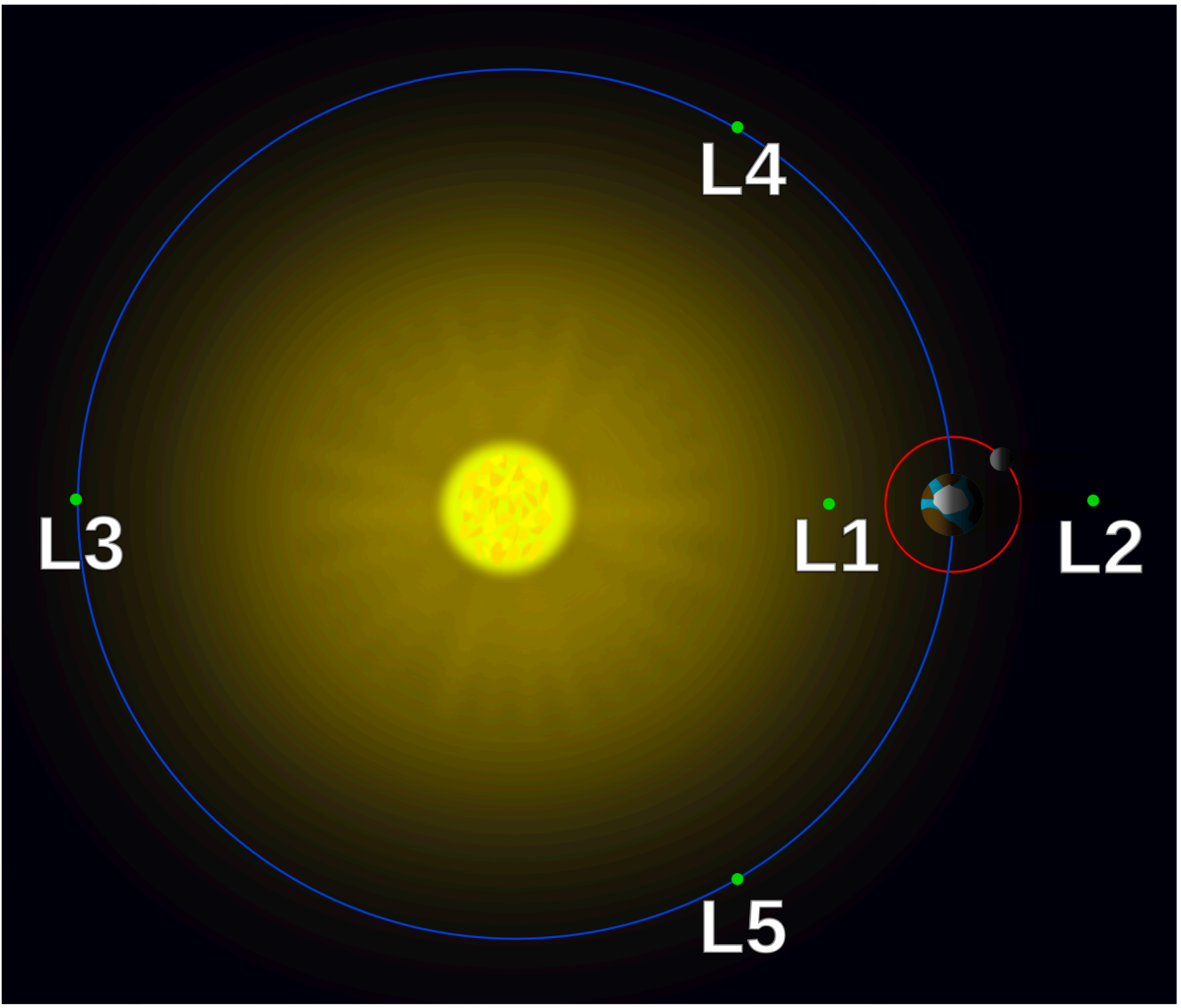
JWST (6.16)



JWST orbits at the L2 Lagrange Point. Lagrange Points are spots of equilibrium in the Earth-sun orbit! Here, the combined gravitational force of the Earth and Sun combine to provide the necessary centripetal force for JWST to have the same period as that of the Earth

https://en.wikipedia.org/wiki/Lagrange_point#/media/File:Lagrange_points_simple.svg

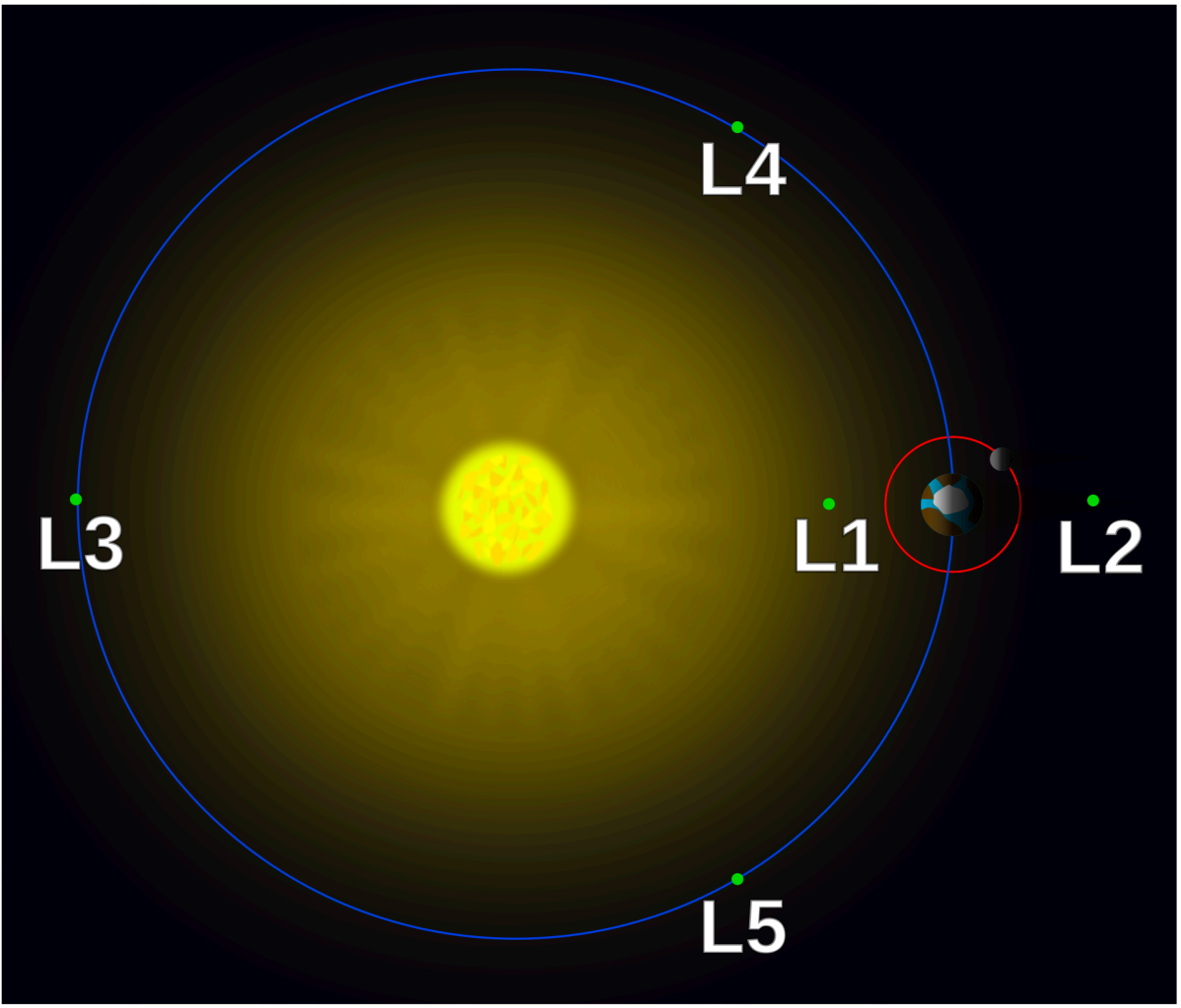
JWST (6.16)



Any ideas as to why L2 is the best spot for JWST?

https://en.wikipedia.org/wiki/Lagrange_point#/media/File:Lagrange_points_simple.svg

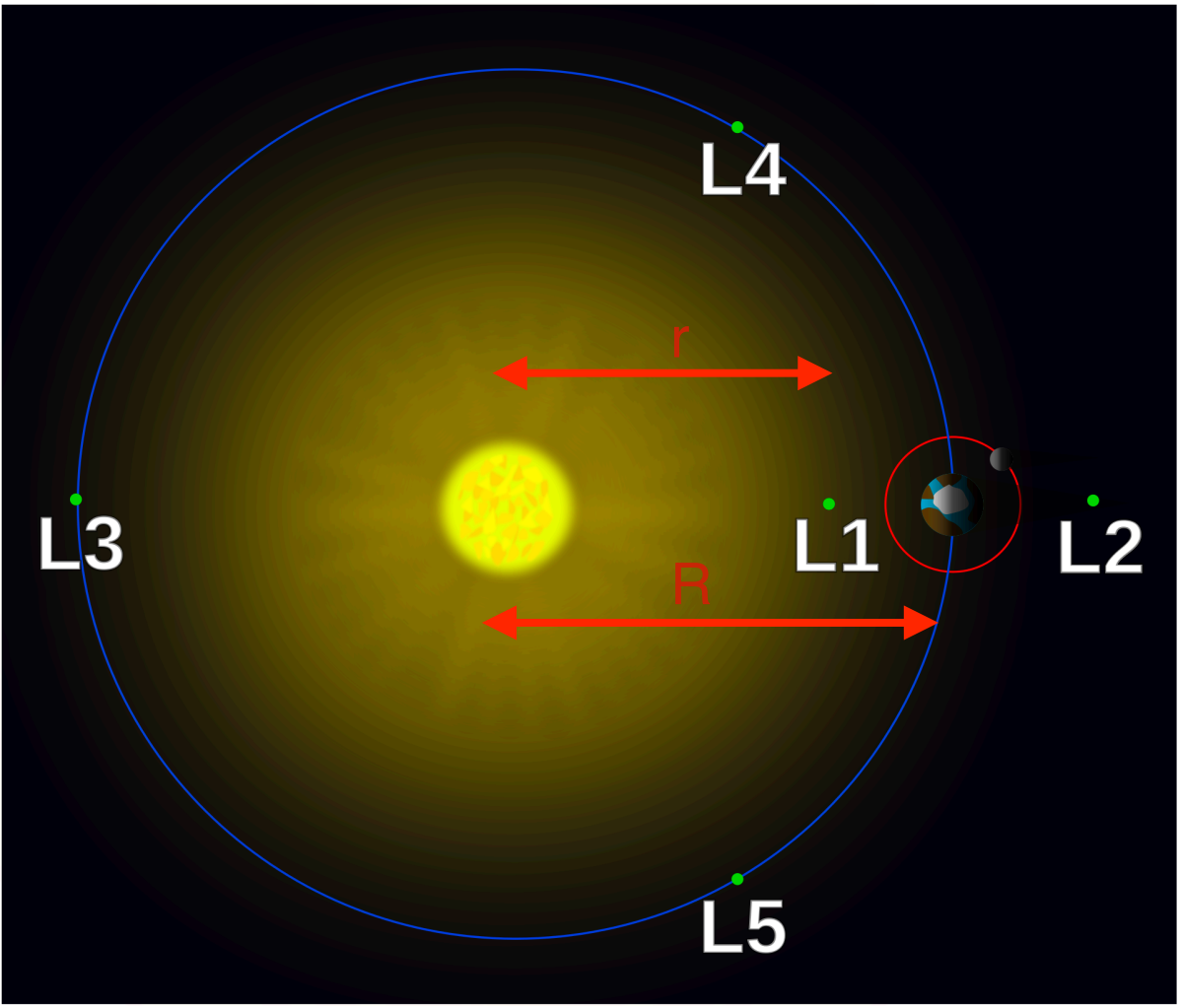
Ex 6.16



Any ideas as to why L2 is the best spot for JWST?

https://en.wikipedia.org/wiki/Lagrange_point#/media/File:Lagrange_points_simple.svg

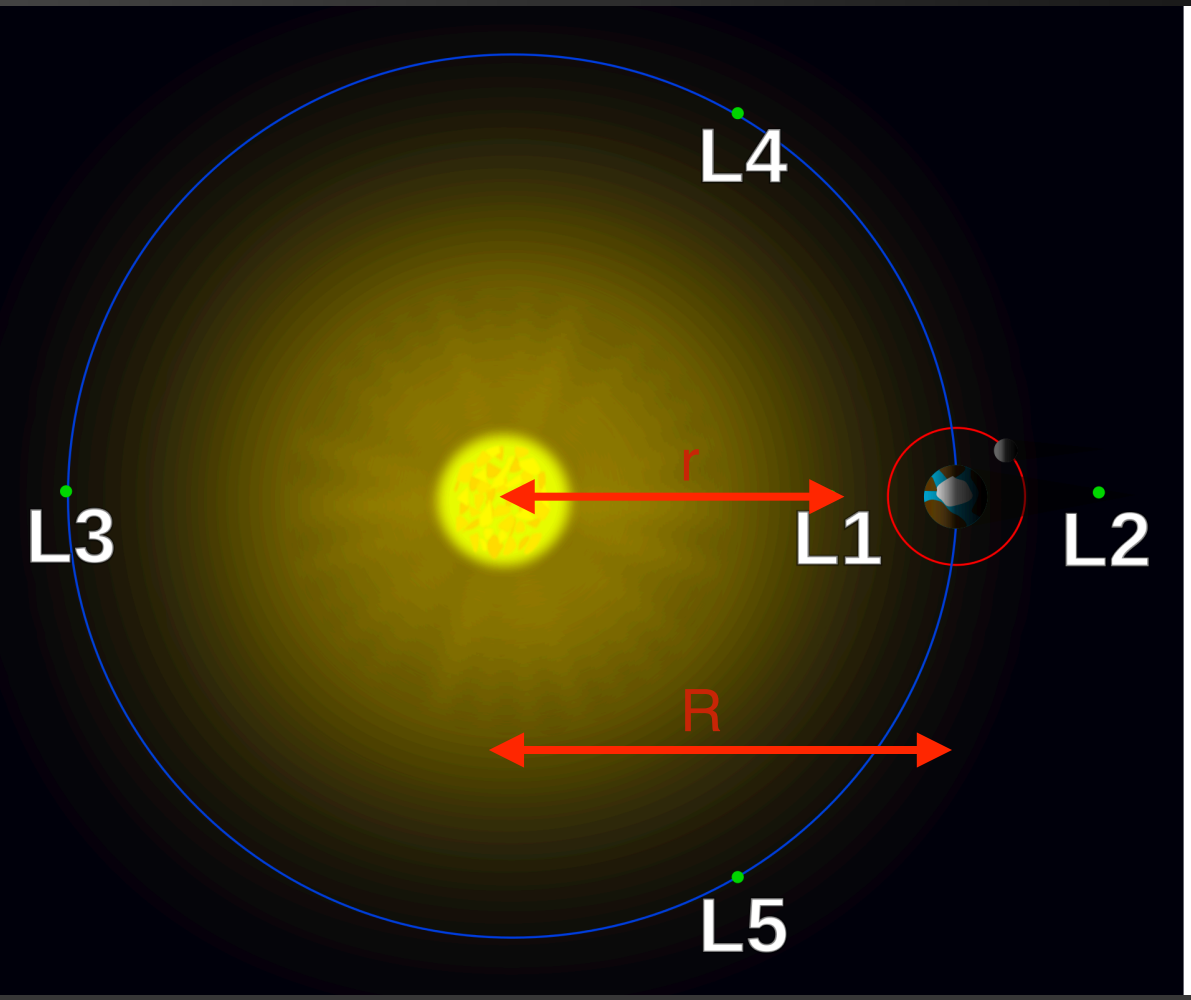
Ex 6.16



Let's look at the L1 point (following 6.16). Note that it looks at the L1 point between a satellite, the earth and the moon. Here we look at the L1 point between a satellite, the sun and the Earth, but the same idea holds

https://en.wikipedia.org/wiki/Lagrange_point#/media/File:Lagrange_points_simple.svg

Ex 6.16



If distance from earth to satellie is r and distance from earth to moon is R then distance from satellite to moon is R-r As in the diagram the two gravitational forces are in opposite directions, and it is orbiting Earth, so the force towards the Earth is bigger. The sum of the forces is then:

$\frac{GM\mu}{r^2} - \frac{Gm\mu}{(R-r)^2}$, where μ is the mass of the satellite. This is then equal to the required centripetal force $\mu r\omega^2$:

$\frac{GM\mu}{r^2} - \frac{Gm\mu}{(R-r)^2} = \mu r\omega^2$. Canceling the common factor we get:

$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = r\omega^2$

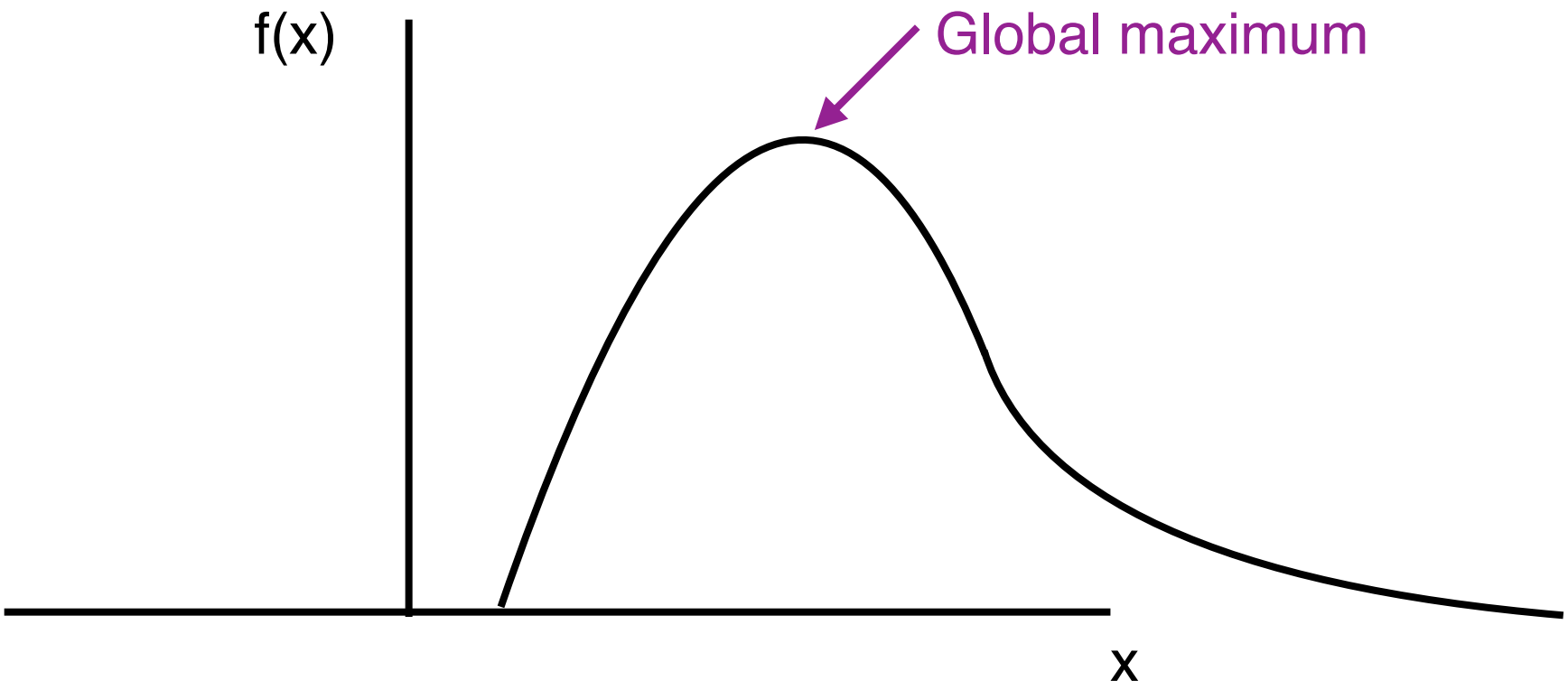
Ex 6.16


```
# 6.16
G = 6.674e-11 #Newton
M = 5.974e24 # mass of earth (kg)
m = 7.348e22 # mass of moon (kg)
R = 3.844e8 # distance from Earth to moon (m)
omega = 2.662e-6 # angular speed, rad/s
target = 1e-10

def f(r):
  return G*M/(r**2) - G*m/((R-r)**2) - omega*omega*r

# Initial guess
r1 = 0.00001*R
r2 = 0.9999*R
f2 = f(r1)

# Main loop
while abs(r1-r2) > target:
  f1,f2 = f2,f(r2)
  r1,r2 = r2,r2-f2*(r2-r1)/(f2-f1)

print("Distance to the L1 point is",r2," meters")
print("Distance to the L1 point is",0.001*r2," km")
```

```
Distance to the L1 point is 326045071.66535544  meters
Distance to the L1 point is 326045.07166535547  km
```

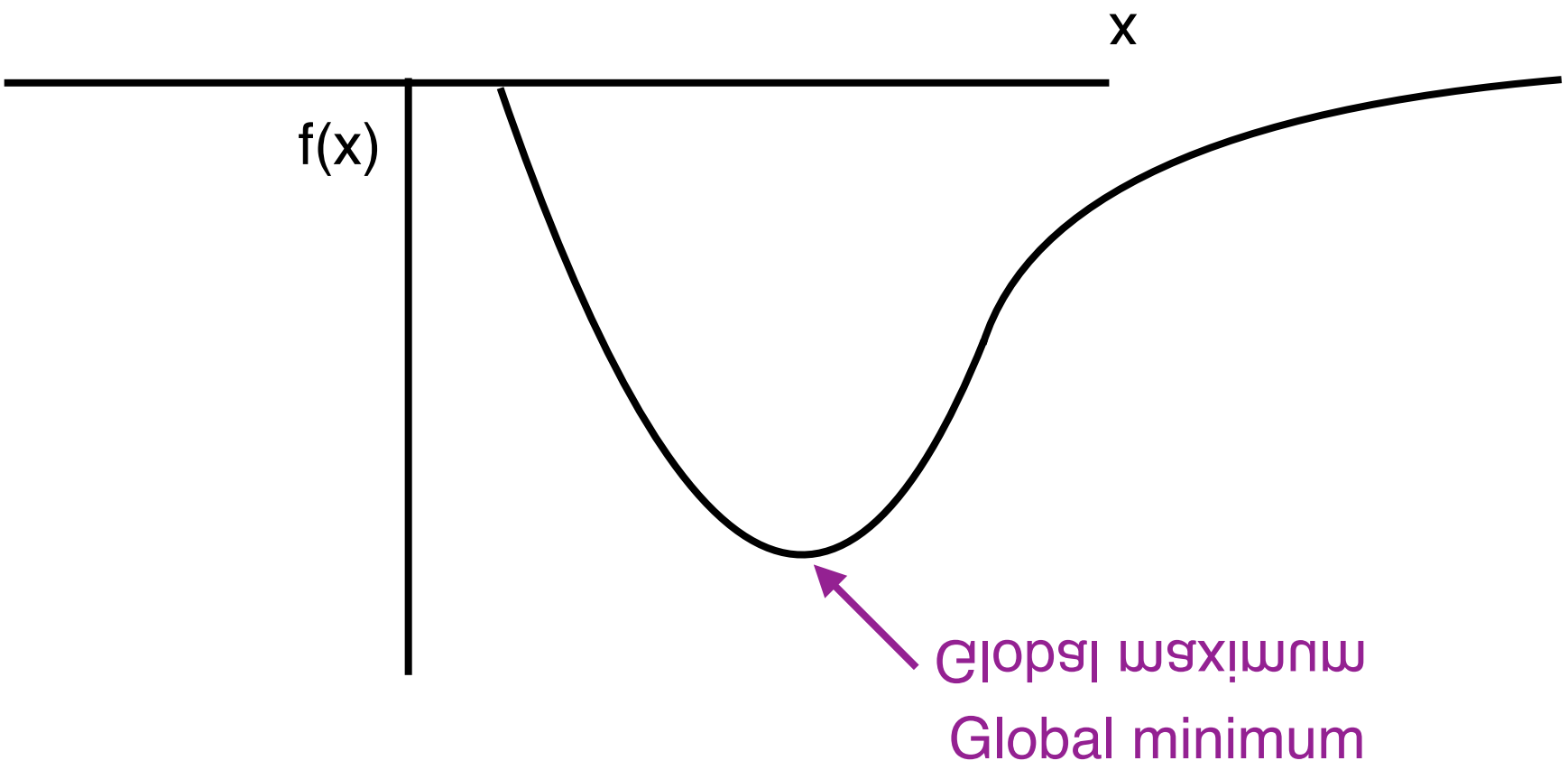
Quick solution!

Minimization and maximization problems



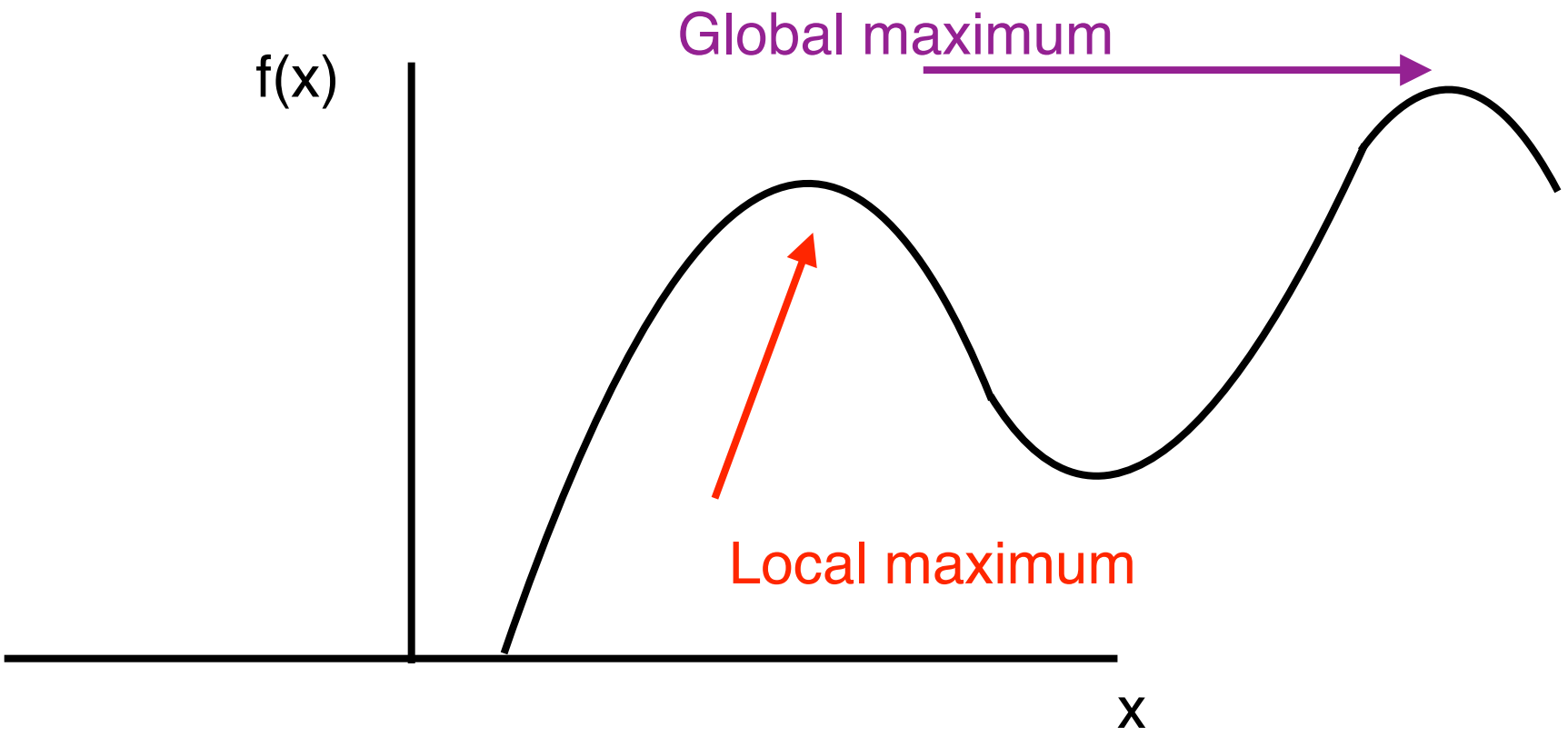
Important question to ask:
How to find the maximum value of f(x)?

Minimization and maximization problems



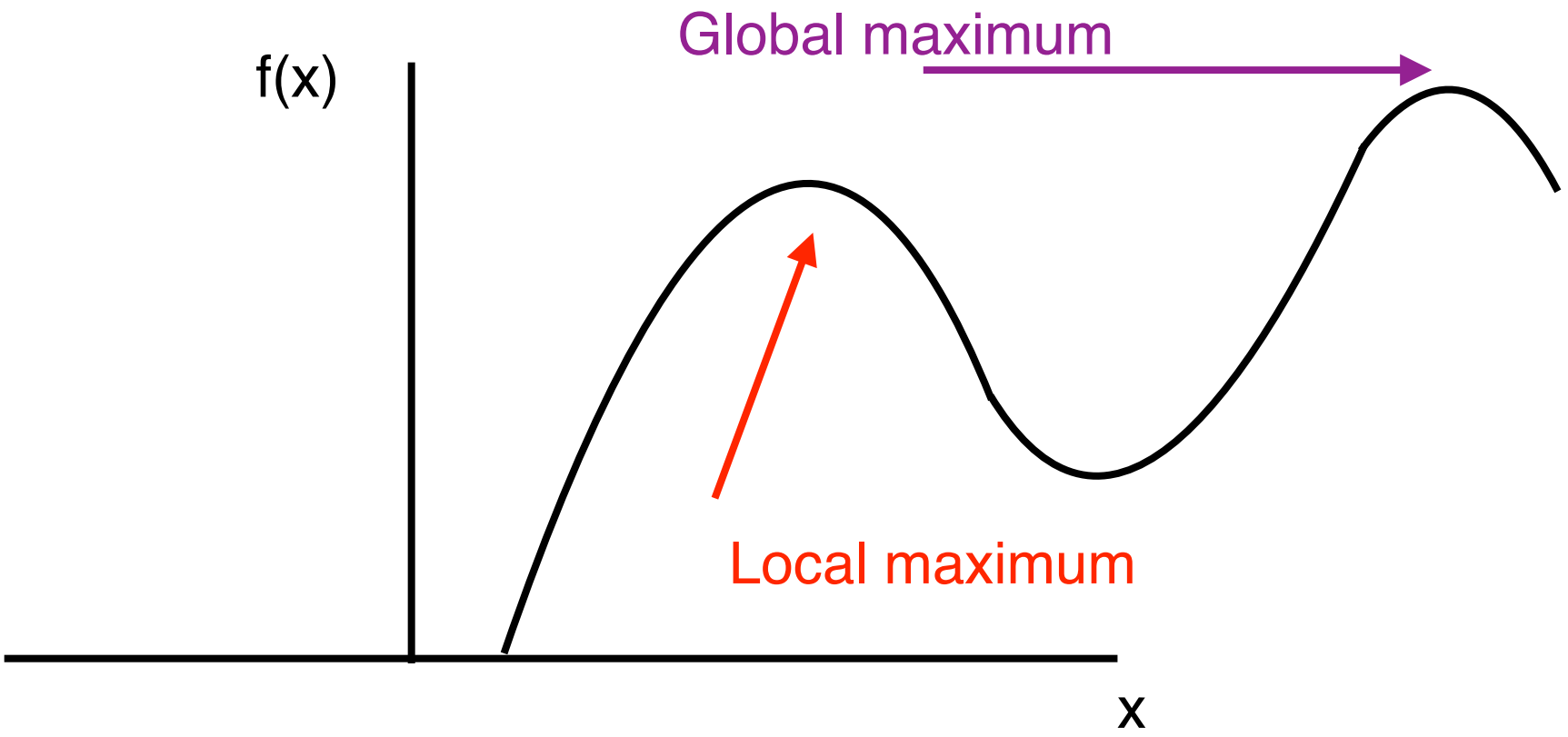
Finding the minimum and finding the maximum is equivalent (just flip by multiplying with -1)!

Minimization and maximization problems



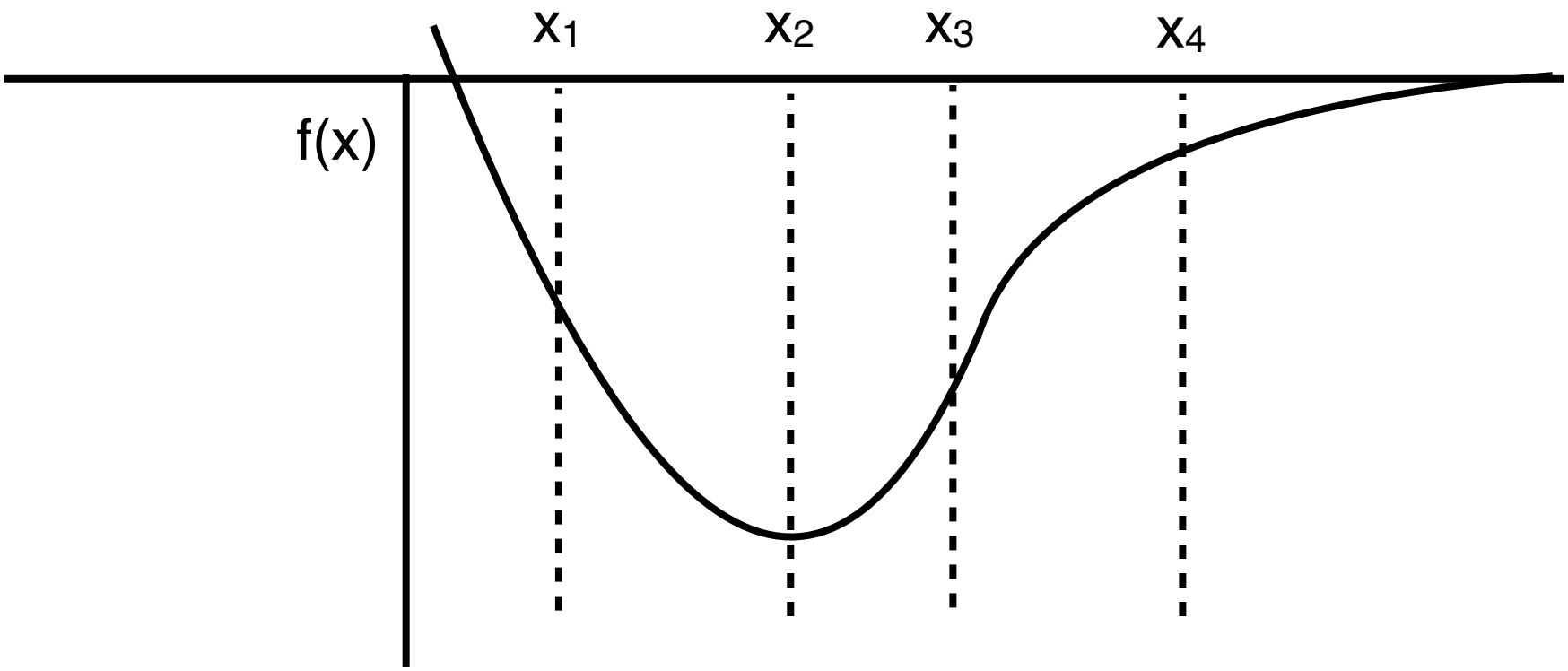
Note that we may be interested in the global maximum or a local maximum, and sometimes we may not know which we have found!

Minimization and maximization problems



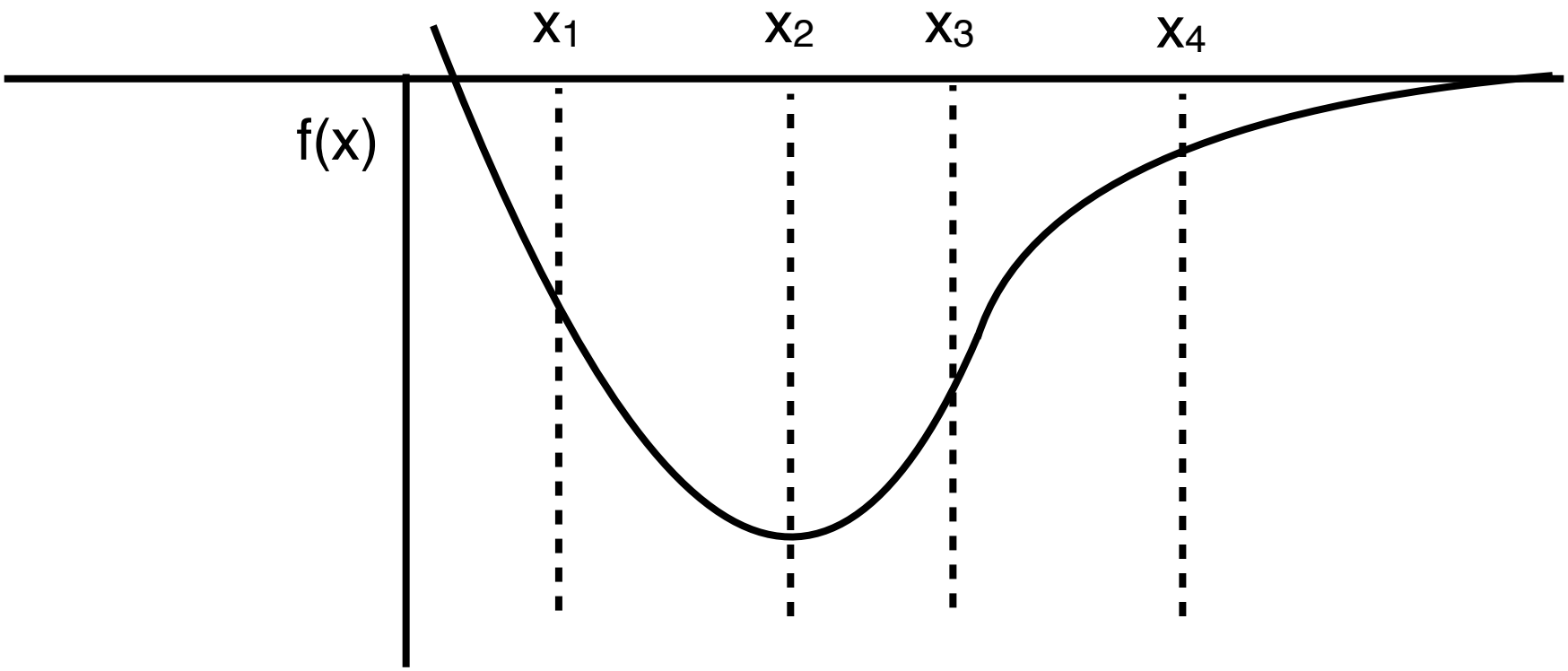
One thing that **differentiates** maxima/ minima from all other points is that the derivative at that point is zero

Golden ratio search



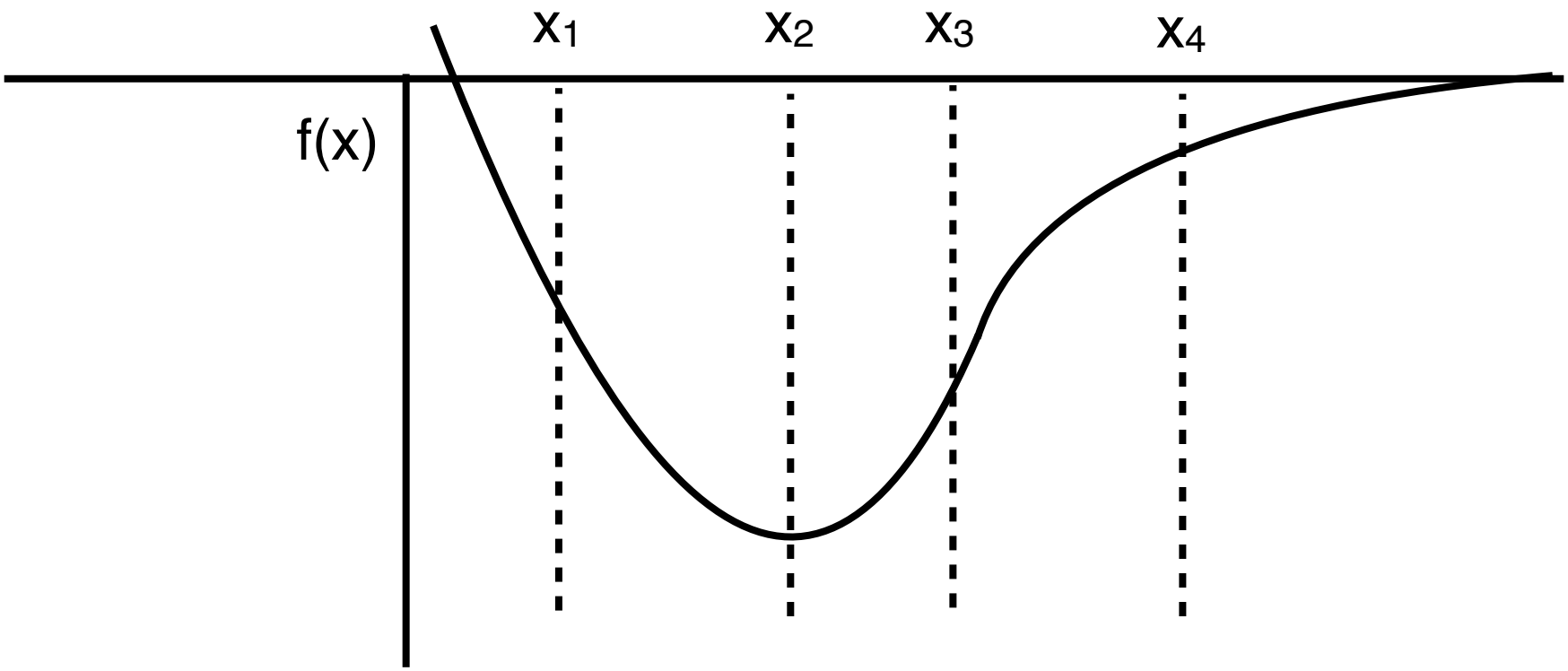
Want to find the (local) minimum. We choose 4 points to start. Pick x_1 and x_4 such that we expect to find a minimum between them

Golden ratio search



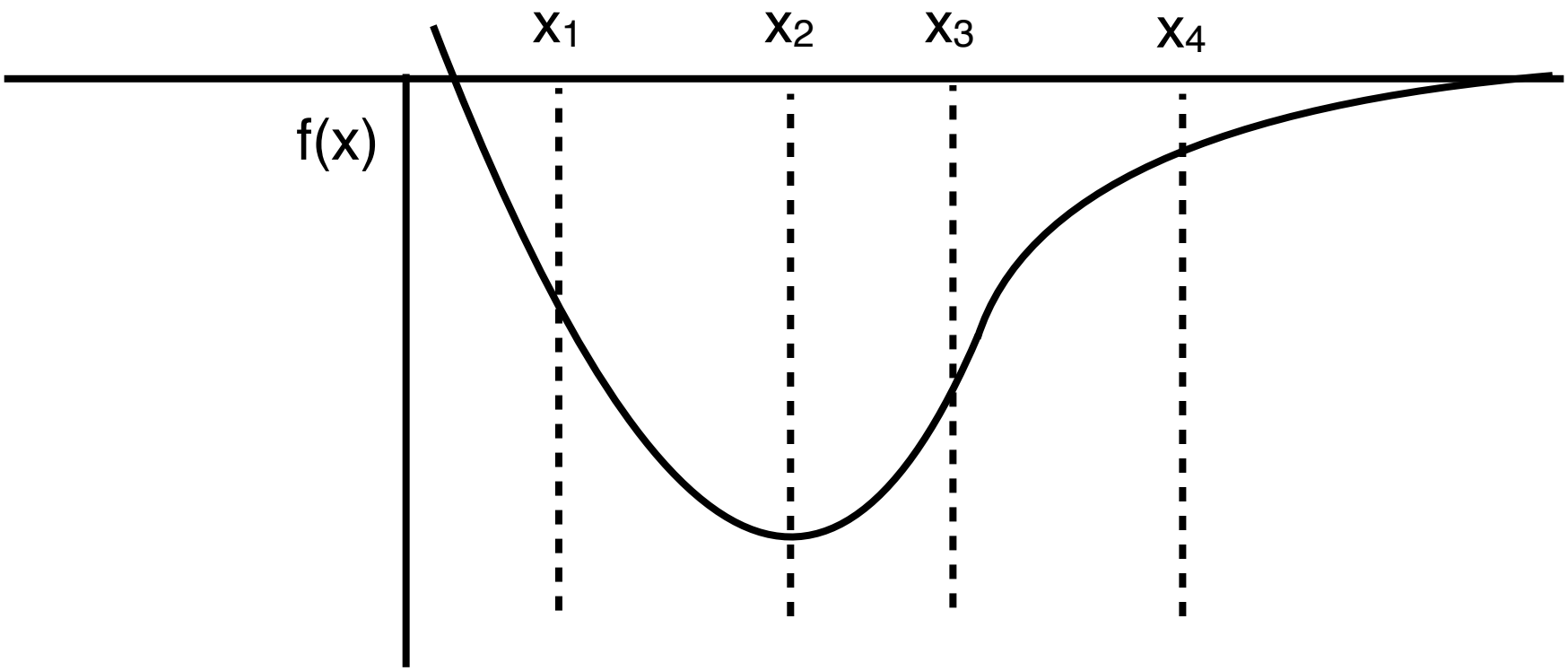
We check that at least $f(x_2)$ or $f(x_3)$ is less than $f(x_1)$ and $f(x_4)$. In that case there must be a minimum between x_1 and x_4 since the function goes down and them back up

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

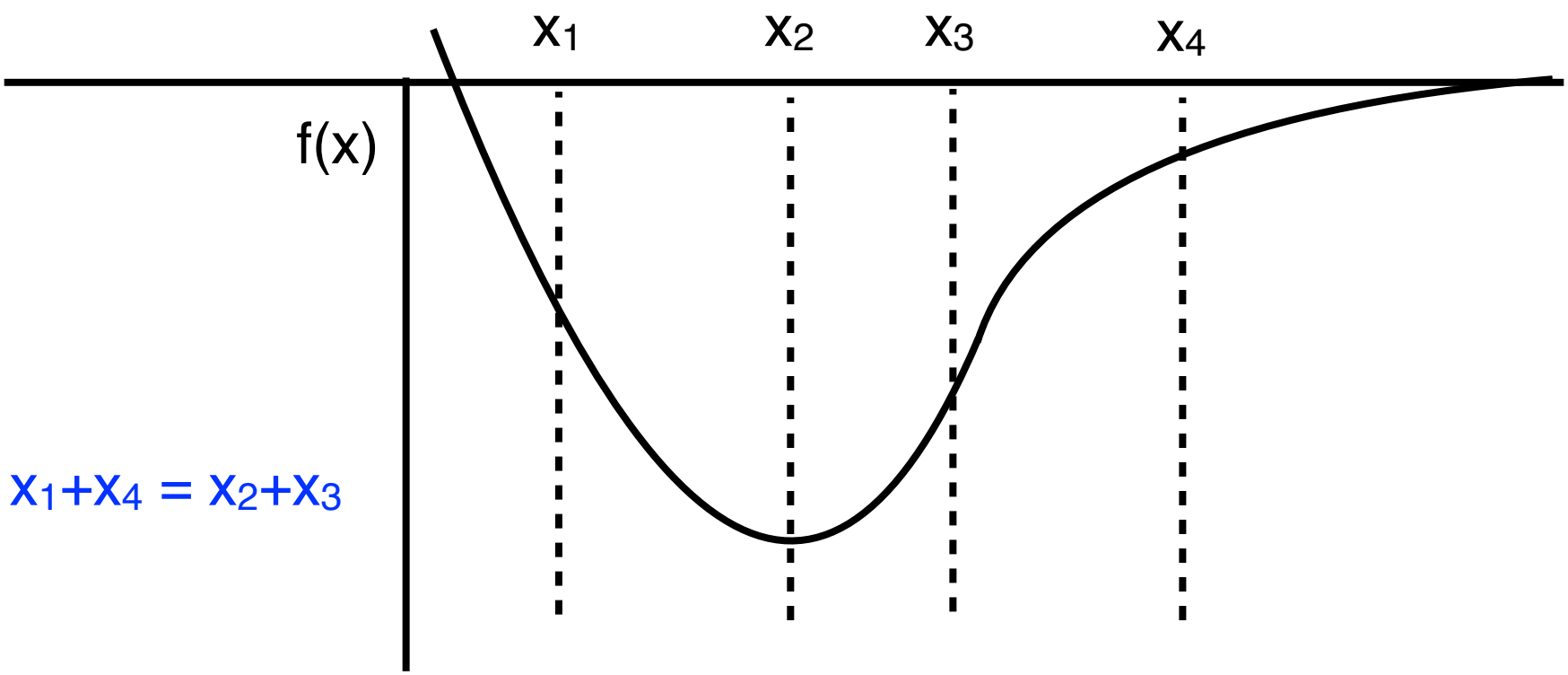


We try to put x_2 and x_3 the same distance from the midpoint of x_1 and x_4 (we don't know any better which side to prefer). Midpoint is $(x_1+x_4)/2$, so:

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

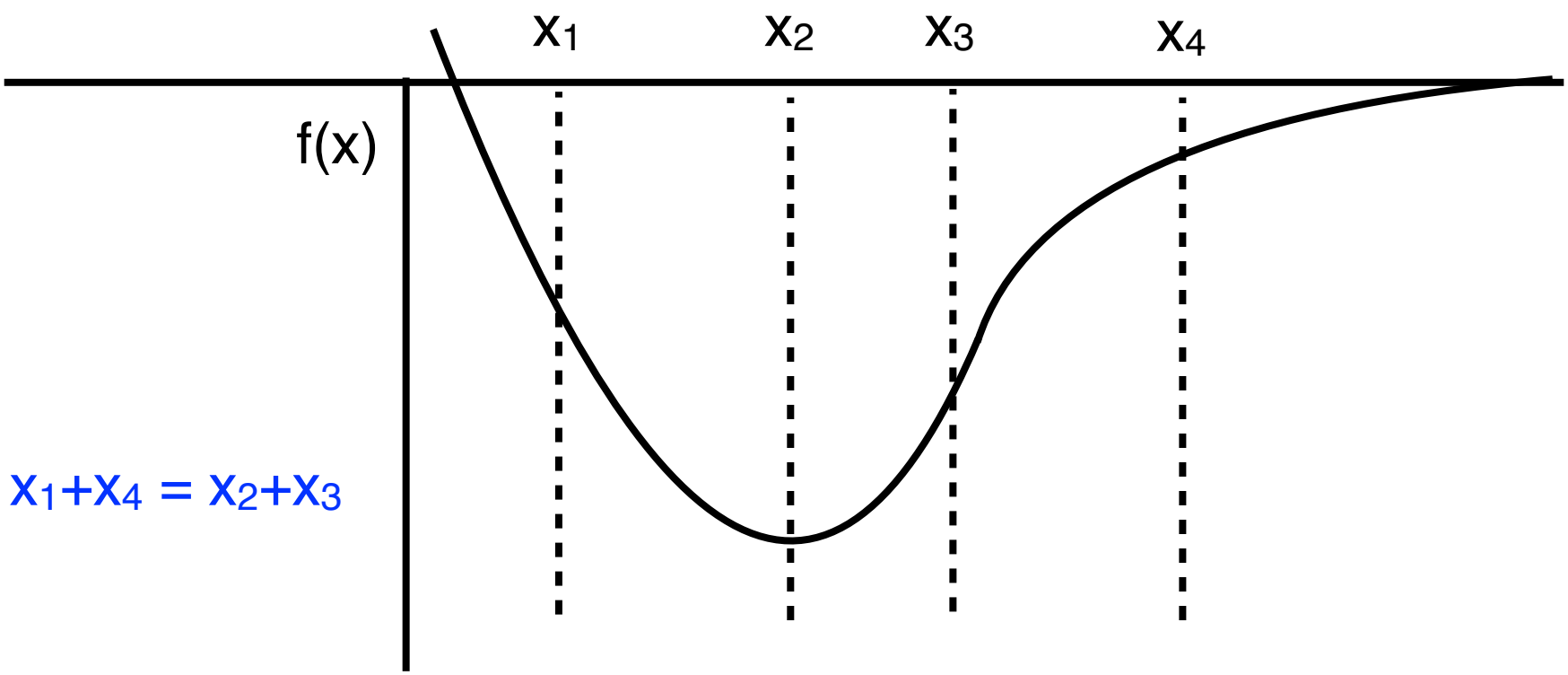
$$x_1+x_4 = x_2+x_3$$

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

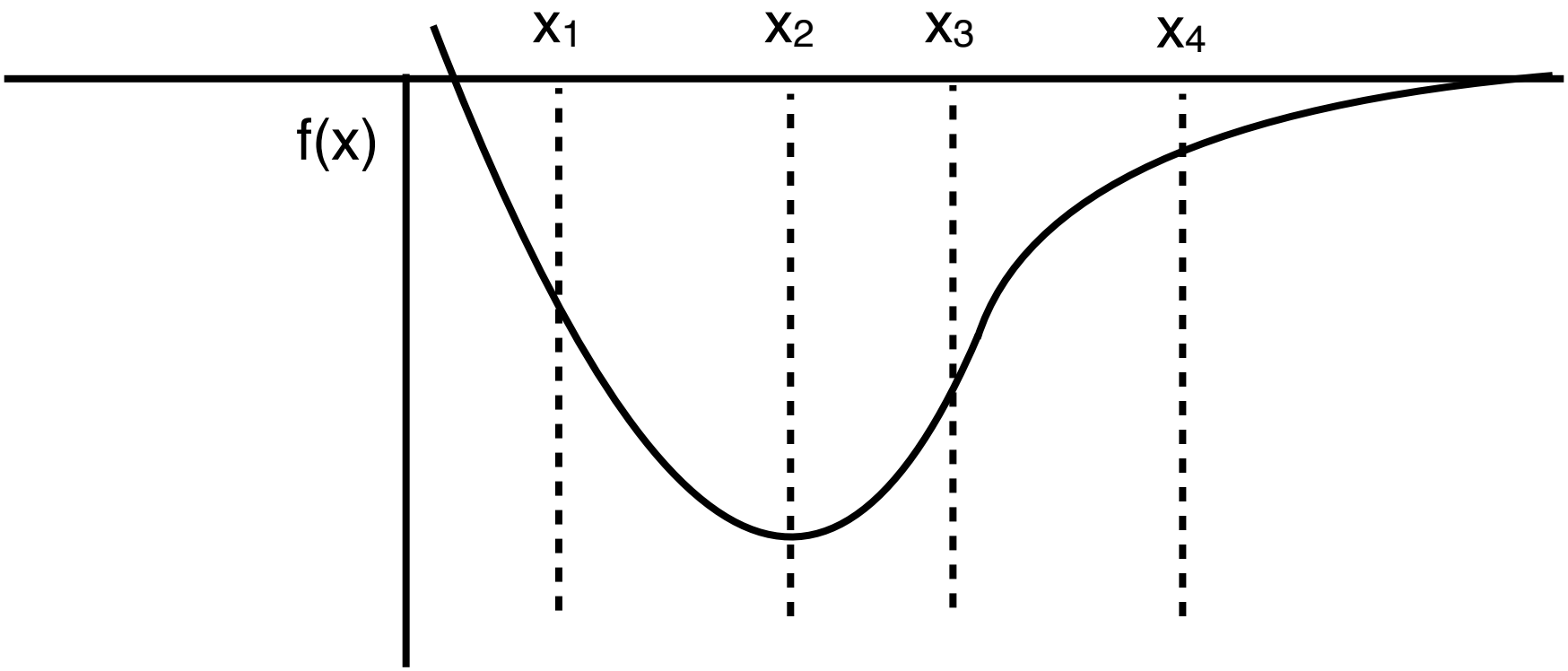
Golden ratio search



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



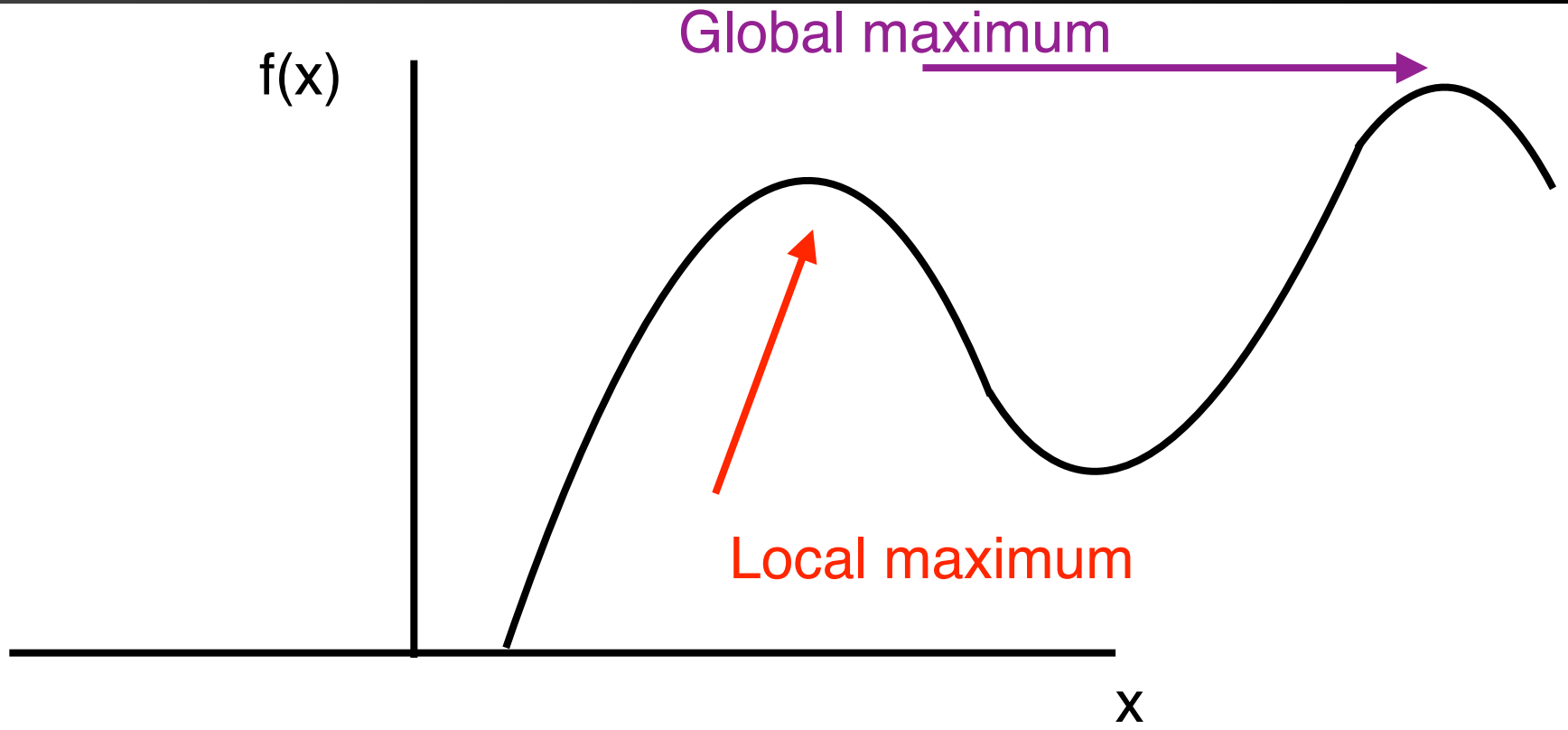
$z=(x_2-x_1)/(x_3-x_1)+1=(x_3-x_1)/(x_2-x_1)$

$1/z+1 = z$, or $1+z=z^2$

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

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

Gauss-Newton Method and Gradient Descent



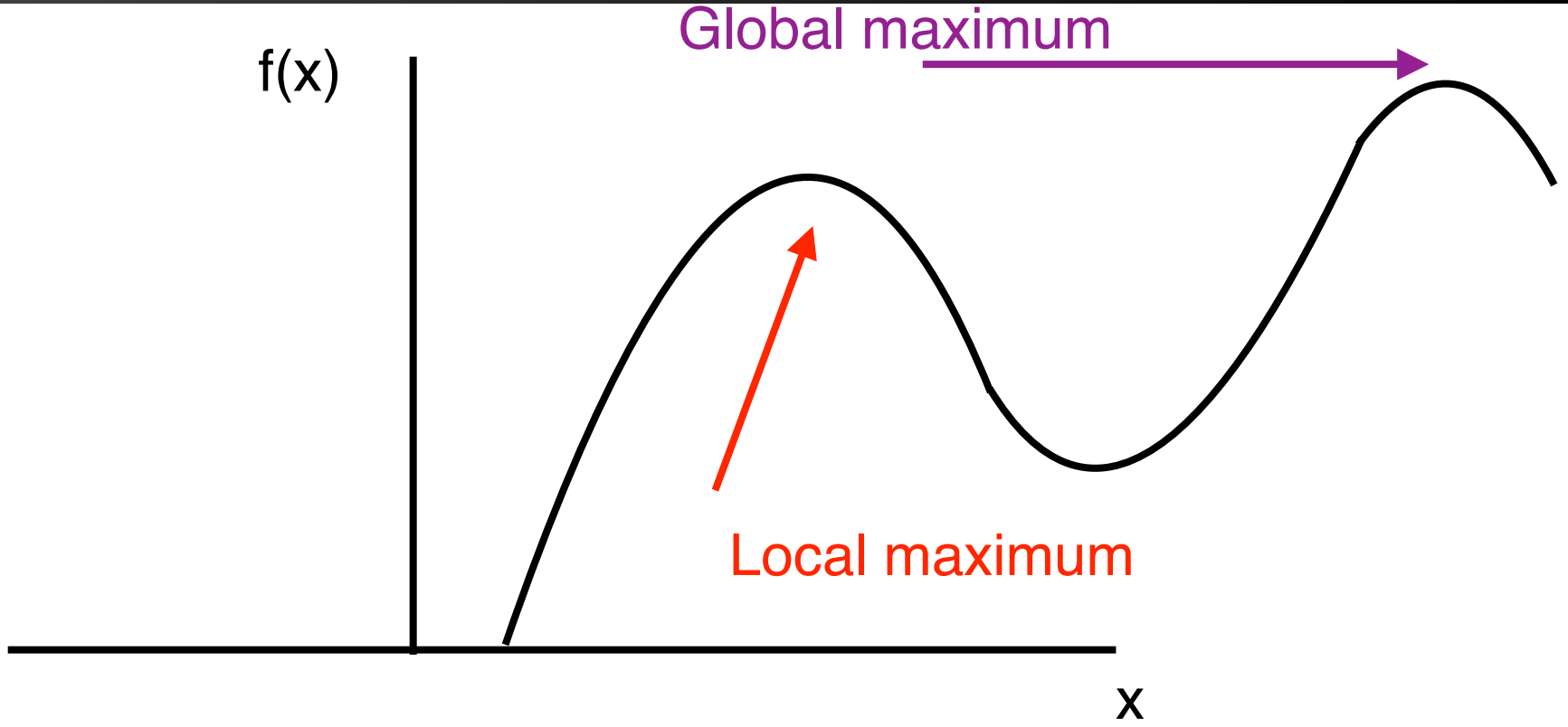
Can find a minimum or a maximum by locating a point where $f'(x) = 0$. But we know how to solve $g(x) = f'(x) = 0$ using the Newton's method! Start with one guess and then

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

Challenge: what is f''(x)? Solution: Call it a constant, ie γ so

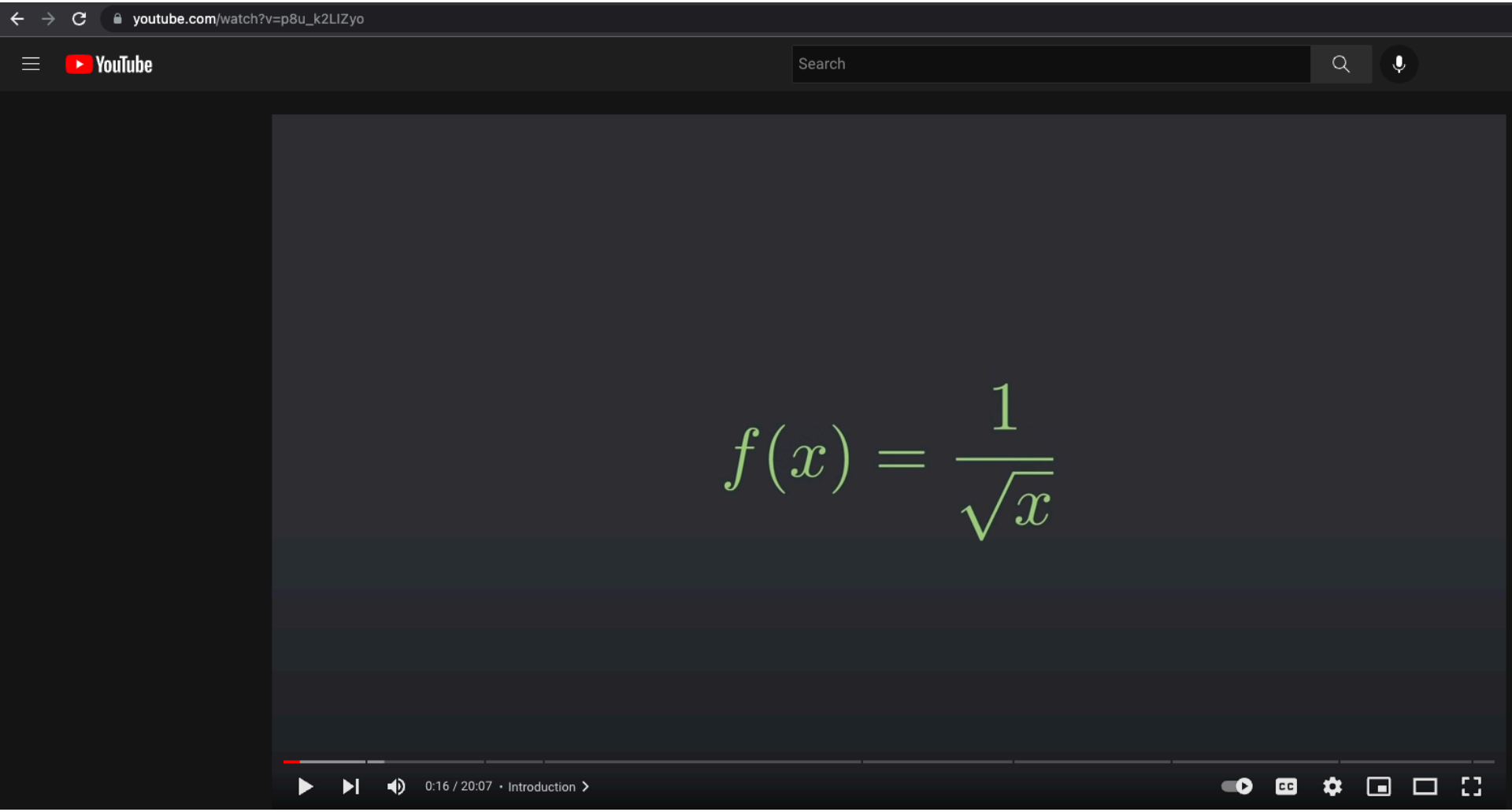
$$x' = x - \gamma f'(x)$$

Choice of sign in Gradient Descent



x'=x-γf'(x), if γ is positive we move toward the minimum of a function (let's check this above), if γ is negative we move toward the maximum of a function (let's check in the picture above). If γ 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

A fun video to watch on your own time



https://www.youtube.com/watch?v=p8u_k2LIZyo

Homework #4

https://classroom.github.com/a/7jzzvD8W

From textbook: 6.7, 6.13, 6.18.
Also, for 510 students, problem 6.9.
Finally, also for 510 students, use Newton's method to find sqrt(a), where a is a constant. Test this for a = 2. How quickly does this converge?