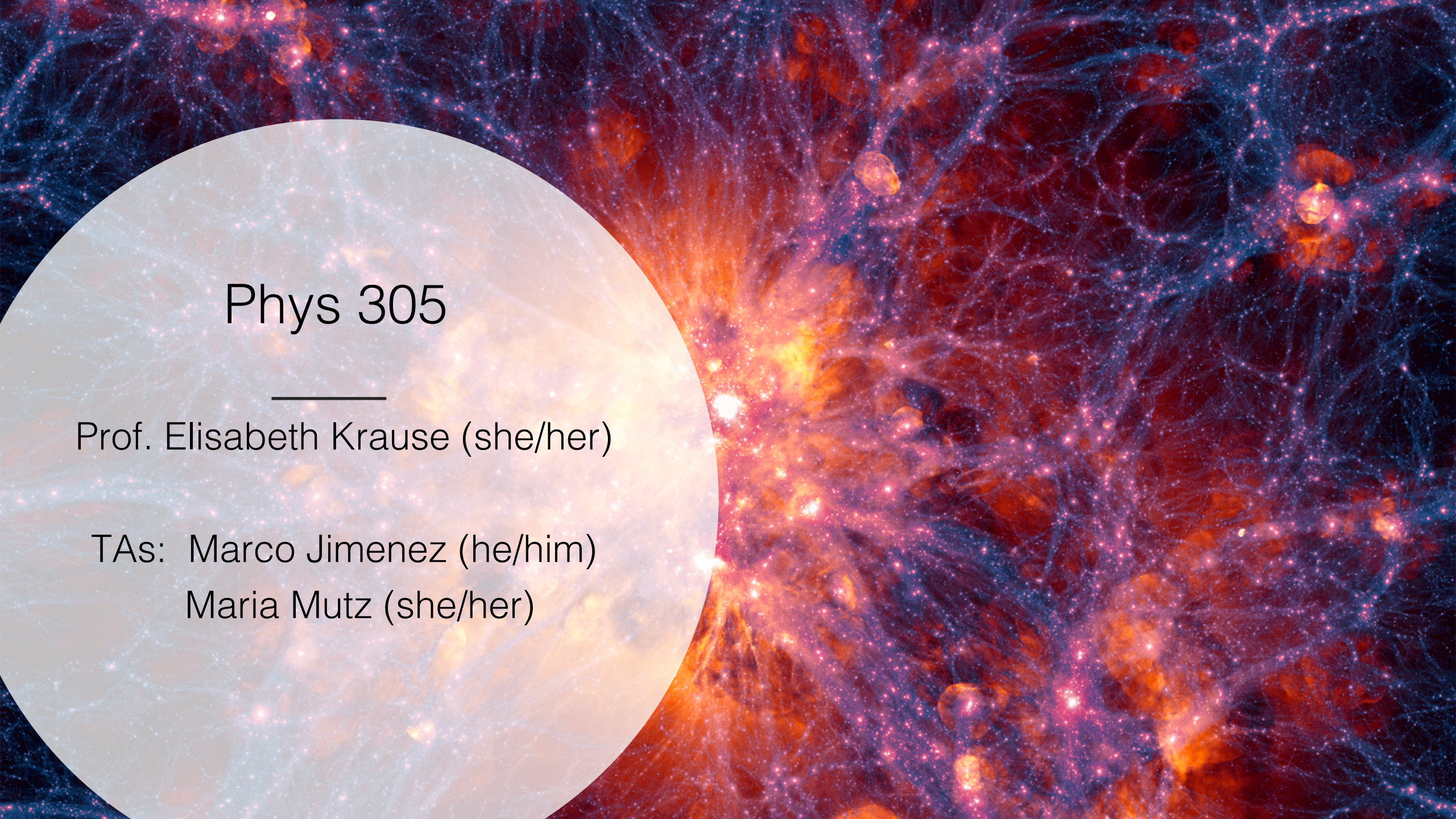


Phys 305

Prof. Elisabeth Krause (she/her)

TAs: Marco Jimenez (he/him)
Maria Mutz (she/her)

Today's Lecture

- Announcements
  - Slideshow from URM Recruiting and Retention Task Force on opportunities for STEM undergrads posted to Course Notes on D2L
  - **Problem Set 1** posted to D2L, due 9/4/2020 at 5:00 (on D2L)
  - **Homework Submission** (pdf in Course Notes on D2L)
  - **Consultation hours** for undergrad physics courses: https://w3.physics.arizona.edu/tutoring, Marco's slot is F 10-11am
- Algorithm: Root finding, part 2
- Python Syntax: lists & arrays, slicings

Homework Submission

- Create a separate file for each problem, the file name should include the problem set number and problem number
- Each file should contain a short high-level description, further comment your code as much as needed!
- Create a directory [your NETID]_[PS#] and copy all files into that directory
- Create a tar file, and submit it on D2L

The goal of this submission process is to familiarize you with your computer's file system – other submission formats (e.g., ipython notebooks) will be graded, but with 5 points deducted.

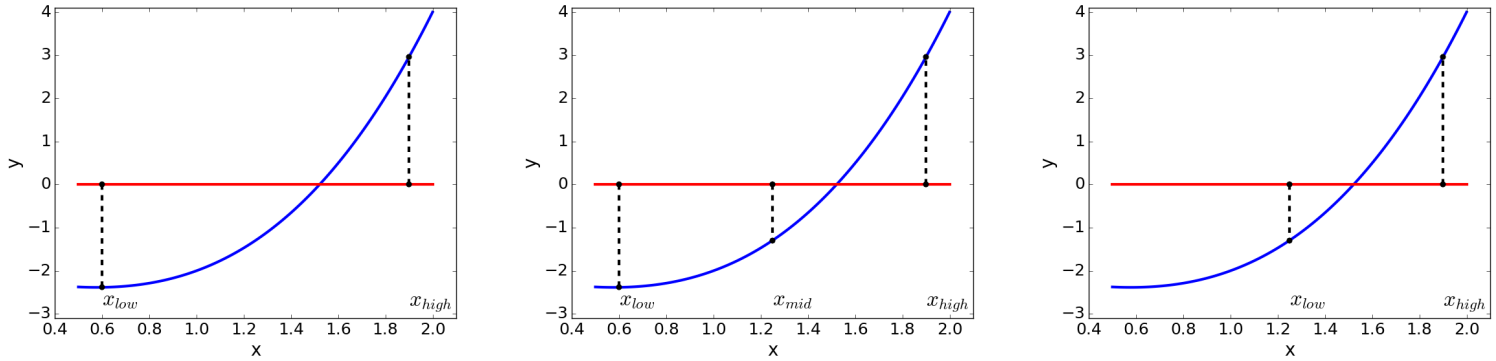
Root Finding: Bispection (Recap)

Start from x_{lo}, x_{high} bracketing the root ($f(x_{lo})f(x_{high}) < 0$)

1. Set $x_{mid} = (x_{hi} + x_{lo})/2$. x_{mid} must lie between the two bracketing values, and hence must be closer to the desired solution than one of them.
2. Now replace one of the bracketing values by x_{mid}; choose the one which will preserve the invariant. That is, if $f(x_{mid})f(x_{hi}) < 0$, then assign $x_{lo} = x_{mid}$, otherwise assign $x_{hi} = x_{mid}$.
3. Repeat until some condition is satisfied.

Iterative root finding algorithm, which always succeeds in finding an approximate solution iff starting values fulfill $f(x_{lo})f(x_{high}) < 0$.

The approximation error decreases linearly with the number of iterations ->slow and expensive!



The Taylor Approximation

Approximate a function f(x) by its first n derivatives
very important tool for many STEM problems
more often than not, n=1

Suppose that some function $f(x)$ has a continuous n-th derivative $f^{(n)}(x)$ on the interval $[a, b]$. Integrating this derivative we have

$$\begin{aligned} \int_a^x f^{(n)}(x)dx &= f^{(n-1)}(x)\Big|_a^x \\ &= f^{(n-1)}(x) - f^{(n-1)}(a). \end{aligned} \tag{1}$$

Integrating again, we have

$$\int_a^x \left\{ f^{(n-1)}(x) - f^{(n-1)}(a) \right\} dx = f^{(n-2)}(x) - f^{(n-2)}(a) - (x-a)f^{(n-1)}(a). \tag{2}$$

The Taylor Approximation

If we keep on going, we can get back to an n-times integrated expression which has the original $f(x)$ in it:

$$\begin{aligned}\int_a^x \ldots \int_a^x f^{(n)}(x)(dx)^n &= f(x) - f(a) - (x-a)f'(a) \\ &\quad - \frac{(x-a)^2}{2!}f''(a) - \ldots - \frac{(x-a)^{n-1}}{(n-1)!}f^{(n-1)}(a). \end{aligned} \tag{3}$$

Solving for $f(x)$, we have the familiar expression for Taylor's series:

$$\boxed{f(x) = f(a) + (x-a)f'(a)} + \frac{(x-a)^2}{2!}f''(a) + \ldots + \frac{(x-a)^{n-1}}{(n-1)!}f^{(n-1)}(a) + R_n \tag{4}$$

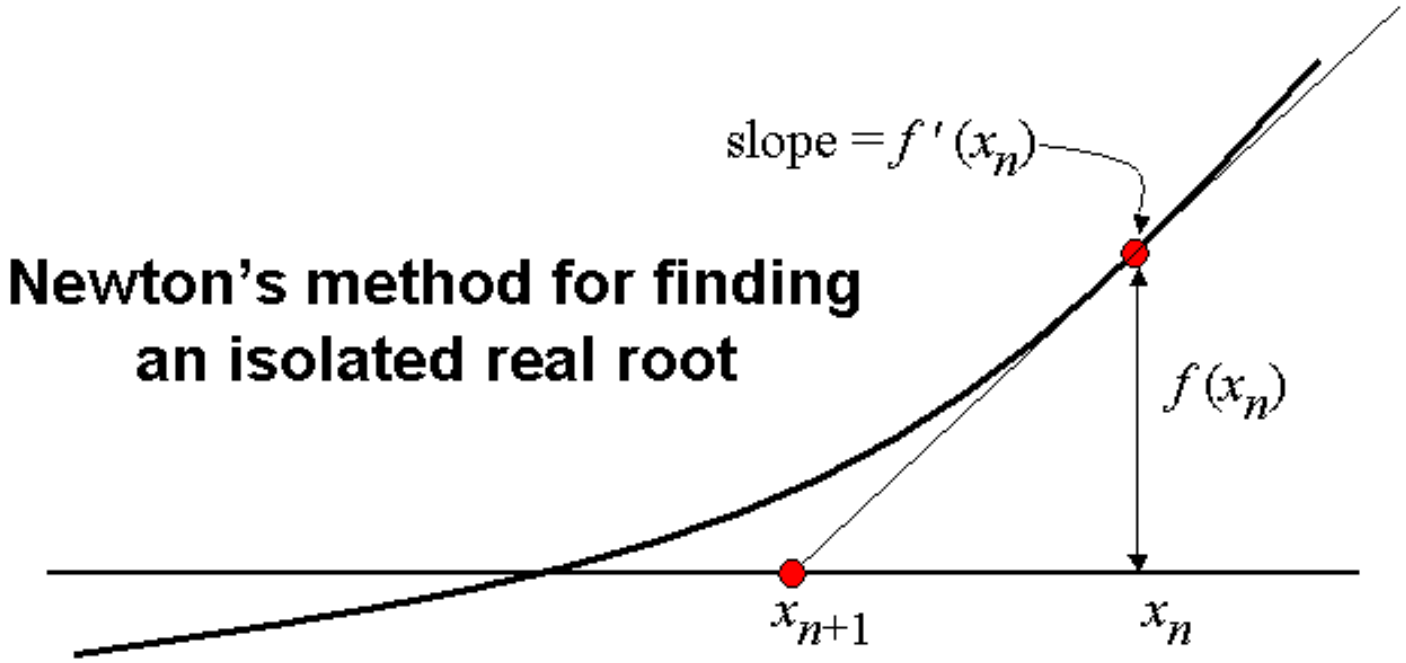
where the last term R_n is the multiple integral on the left hand side of equation (3),

$$R_n = \int_a^x \ldots \int_a^x f^{(n)}(x)(dx)^n. \tag{5}$$

R_n remainder term, which quantifies the error of this approximation – details in RootFinding.pdf notes

Root Finding: Newton Method

use the first-order Taylor approximation (i.e., information about the slope of our function) to improve the root finding approximation in each step



$$x_{n+1} = x_n - \frac{f'(x_n)}{f(x_n)}$$

Root Finding: Newton Method

Once again, we are trying to solve

$$f(x) = 0$$

If we make a guess at the solution, x_0, we can expand f around the guess as a Taylor series. Taking the first two terms, we have

$$f(x) \sim f(x_0) + (x - x_0) f'(x_0)$$

If we set the right-hand side of this equal to zero, we can solve for the difference $\delta = x - x_0$:

$$\delta = -\frac{f(x_0)}{f'(x_0)}$$

We can then "correct" our guess, and develop a sequence of approximations by iteration. If our guess was x_i, then we can improve our guess as

$$\delta_i = -\frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} = x_i + \delta_i$$

or

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$$

Root Finding: Newton Method

If $\epsilon_i = x - x_i$ is the error at the i-th iteration, we can re-write our Taylor series as

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + (x - x_i)^2 \frac{f''(\zeta)}{2}$$

The last term is known as the "remainder term", and follows from the mean value theorem (see the end of these notes for how one might prove this). The value ζ lies somewhere in the interval $[x_i, x]$; we need not know its precise value. Dividing by $f'(x)$ and rearranging, we have

$$x - x_i + \frac{f(x_i)}{f'(x_i)} = -\frac{f''(\zeta)}{2f'(x_i)}(x - x_i)^2$$

Using our Newton iteration from the previous paragraph, we have

$$\epsilon_{i+1} = -\frac{f''(\zeta)}{2f'(x_i)}\epsilon_i^2$$

Hence, Newton's method converges quadratically – it about doubles the number of significant figures in the result per iteration.

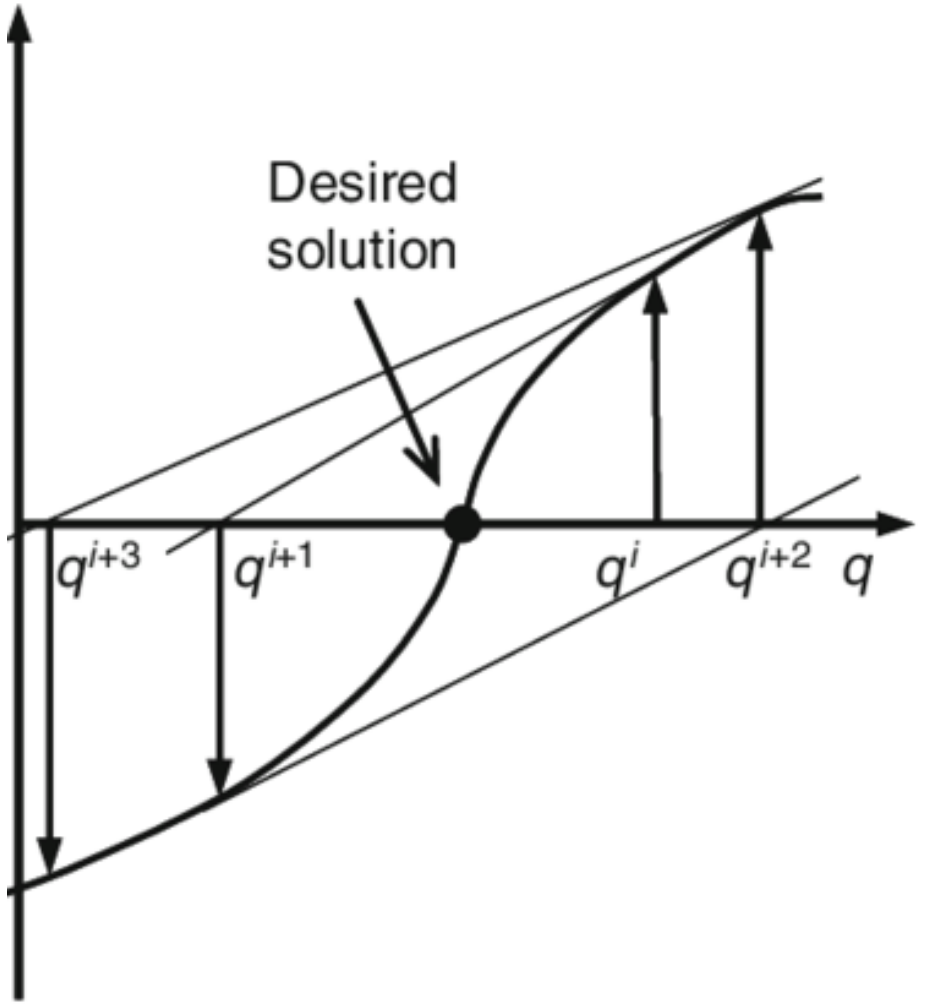
Newton Method

If it converges, the Newton Method converges quadratically

However, conversion is not guaranteed and depends on the function (and starting point)

more details in pdf note

Note: division by f’ numerically at saddle points!



Root Finding: Best of Newton & Bisection

- Bisection gives guaranteed, but slow convergence
- Newton iteration give quadratic convergence, but only when it works
- Combine these two for guaranteed, and improved convergence:
  - Choose starting values bracketing the root
  - Take midpoint as starting point for Newton iteration
  - In each Newton iteration, check whether next guess is within starting interval
    - If yes, continue using Newton iteration
    - If no, next step using Bisection, then try Newton again

At worst, this is bisection with some wasted effort in Newton iterations; at best, converges quadratically within Newton iteration loop

Pseudocode for combining Newton & Bisection

newtonsafe(f, x_1, x_2, ε):

```
fl = f(x1)
fh = f(x2)

if fl < 0:
    xl = x1
    xh = x2
else:
    xl = x2
    xh = x1

xm = (xl+xh)/2
dxold = |xh − xl|
dx = dxold
fm = f(xm)
dfm = f′(xm)

while:
    if ((xm − xh)*dfm − fm) * ((xm − xl)*dfm − fm) > 0 or |2fm| > |dxold*dfm|:
        dxold = dx
        dx = 0.5*(xh − xl)        Bisection
        xm = xl + dx
    else:
        dxold = dx
        dx = fm/dfm               Newton
        xm = xm − dx

    if |dx| < ϵ: return xm

    fm = f(xm)
    dfm = f′(xm)

    if fm < 0:
        xl = xm
    else:
        xh = xm
```

Python Syntax: Lists

- a ***list*** in python is an ordered collection of objects of any type.
  - objects in square brackets, e.g. **testlist = [1, “a string”, 3.0, “last”]**
  - You can make list of lists, e.g. **lol = [[1, 2, 3], [4, 5, 6]]**
- **To access elements of a list**, you put the list element (starting with 0) in square brackets
  - **somelist[1]** for *second* element
  - can access without list name: **[4, 5, 6][2]**
  - can access within nested lists: **lol[1]** givels [4, 5, 6] (i.e. 2nd element)
  - **lol[1][0]** gives 4, i.e. first element of 2nd element
- **empty lists** are a good starting point **a = []**
- **add single elements** with “dot append” function: **a.append(2)**
- list elements are iterable: **for x in somelist:**
  print(x)

Python Syntax: List Slicing

- Access elements of lists (and other containers) with **slicing**:
  - **ll[n:m]** gives elements n+1 to m
  - **ll[:n]** gives first n elements
  - **ll[n:]** gives elements n+1 to the end
- Stride or stepsize given by a third entry after a 2nd colon
  - **ll[::n]** step size n, try **ll[::3]**
  - **ll[::-1]** elements in reverse order
  - **ll[::-n]** elements in reverse order with stepwise n
- can combine stride and slicing, e.g.
  - **ll[10:15:2]**
  - ll**[15:10:-2]** because stepping backwards
  - **ll[10:15][::-2]** it takes practice to understand difference!

Python Syntax: List Slicing (part2)

- List elements and slicing with *negative* integers
- We already saw that negative integers for the stepsize (the third entry) means that elements are extracted in reverse order.
- Similarly, negative integers for the start or stop index (either of the first two entries) count backwards from the end. Thus:
  - **ll[-1]** gives the *last element of the list*
  - **ll[-n]** gives the n-th last element, i.e. -2 the second to last
  - **ll[-n:]** gives the *last n* elements
  - **ll[n:-m]** gives from element n+1 to the element m-th last element, e.g.
  - **ll[0:-1]** is just another way to give the whole list (from first to last elements)

Python Syntax: Arrays

- ***Arrays*** and **lists** are both container objects used in Python to store data. They both can be used to store any data type (real numbers, strings, etc.), and they both can be indexed and iterated through.
- For example, you can divide an array by 3, and each number in the array will be divided by 3. *If you try to divide a list by 3, Python will tell you that it can't be done, and an error will be thrown.*
- For practically all calculations, we will use NumPy arrays
- Note: arrays need to be declared before you can use them
  - e.g.: a = np.array([0,1,2,3,4]), b=np.arange(5), c=np.zeros(5) declare one-dimensional arrays
  - while NumPy has functions that append elements to an array, this does not happen in place; numpy.append(a,b) creates a new array with length(a)+length(b)

Python Syntax: Array Slicing

- ***Arrays*** can be multi-dimensional
  - A = np.zeros (5,3) creates a two-dimensional array
  - B = np.zeros(5,3,2) creates a three-dimensional array
- Arrays can be sliced like lists
  - a[2:4] accesses elements 3 and 4 of a
  - A[2:4,1:2] accesses elements [3,2], [4,2], [3,3], and [4,3]
  - B[2:4,1:2,:] accesses elements [3,2,1], [4,2,1], [3,3,1], [4,3,1], [3,2,2], [4,2,2], [3,3,2], [4,3,2]
- Each array has ***attributes*** ndim (the number of dimensions), shape (the size of each dimension), and size (the total size of the array)
  - print(”B ndim: ", B.ndim) 3
  - print(”B shape:", B.shape) (5,3,2)
  - print(”B size: ", B.size) 30