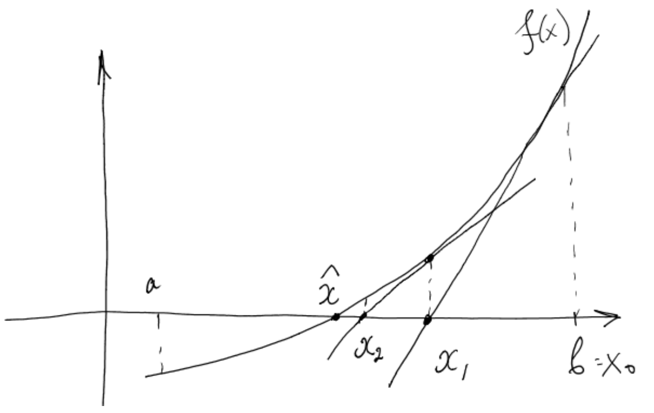
4 Some other methods to find roots of a scalar equation. Heuristic discussion

In this section I briefly describe a few other than bisection methods to finds roots of scalar equations. By no means I will try to cover all the possible details, the goal here is the practicality of the methods and their implementations and not in establishing theorems. I will deal with some of the theoretical aspects later.

4.1 Newton–Raphson method

I start with another method that sometimes is treated in calculus courses. The geometrical idea of this method can be seen in the figure below.



In words, I start with an assumption, that I bracketed my root between a and b (that is, $f(a)f(b) < 0$). After it I choose any point from the interval $[a, b]$ to be my starting point x_0, in the figure I simply take $x_0 = b$ but any other possible choice is acceptable. After it, I find an equation of the tangent line to the graph of f at the point $(x_0, f(x_0))$. Recall that this equation has the form

$$y - f(x_0) = f'(x_0)(x - x_0).$$

Next, I find the point of intersection of this tangent line with the x-axis, and this point will be my next approximation of the unknown root, I will call it x_1. Since at this point $y = 0$ I find

$$-f(x_0) = f'(x_0)(x_1 - x_0),$$

or, solving for x_1,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

I can continue this process by finding the tangent line at the point $(x_1, f(x_1))$, find x_2, etc. In general, I have the following *recurrent* formula for the sequence of my (I hope) approximations of the unknown root $\hat{x}$:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, \ldots \tag{4.1}$$

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

Note that in order to apply the Newton–Raphson method, we must assume that the function is not just continuous but also differentiable since we need to compute its derivative at each step. So, why do we care about this specific method if we already saw that simple and robust bisection works great on even larger set of functions? Here is an explanation by an example.

Consider again the equation

$$x^2 - 2 = 0$$

and take $x_0 = 2$. Since $f'(x) = 2x$, (4.1) becomes

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right).$$

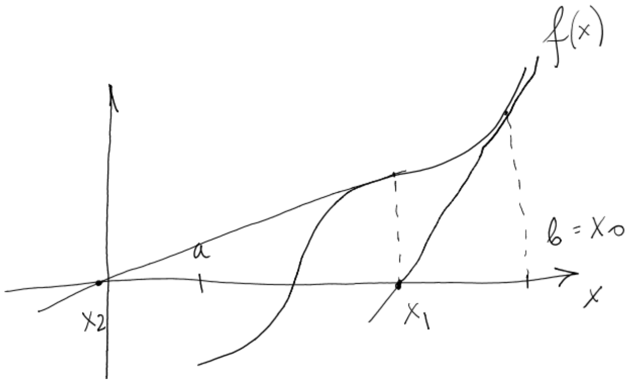
Here is my sequence (x_n)that I computed using Mathematica:

```
x0 = 2
x1 = 1.5
x2 = 1.41666666666667
x3 = 1.41421568627451
x4 = 1.41421356237469
#The actual value of the square root of 2 is
#1.41421356237310
```

You can see that already x_2 has two correct digits, x_3 has 5 correct digits, and x_4 has 11 correct digits. I will leave it as an exercise to decide how many bisections you need to perform to get 11 correct digits, but clearly this number is much bigger than 4. This is the sole reason the this method is so in use.

If this method is so great, why do we even care about anything else? There should be a catch. Actually, there are quite a few things that may go wrong.

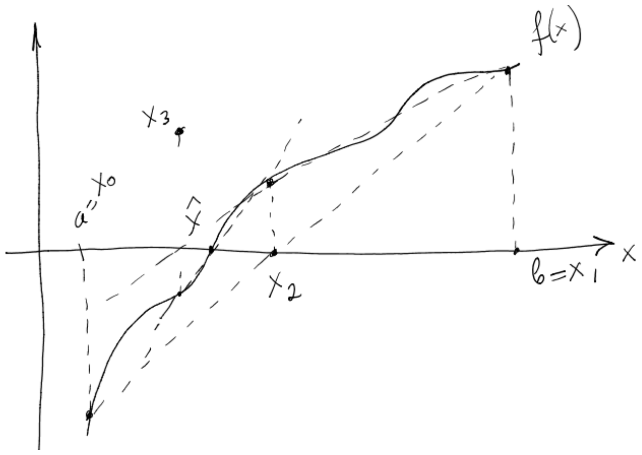
First, we need to know the derivative. This is very easy for a function like x^2-2 but sometimes can be quite difficult to figure out. Second, since we dividing by $f'(x_n)$ we should assume that $f'(x_n) \neq 0$ at any step. But the biggest problem is illustrated by the following figure. Note that already second



approximation leaves the bracketing interval $[a, b]$ and therefore the method clearly is not guaranteed to converge! The outcome depends on a given function f.

4.2 Secant method

A close relative of the Newton–Raphson method is the secant method, which has the advantage of not requiring the knowledge of the derivative of the function. The idea is given in the following figure.



In words, I start with two initial points (my usual a and b) and decide who is x_0 and who is x_1 (there is some freedom of choice here but no perfect recipe). After it I find the equation of the straight line (*secant line*, hence the name of the method) connecting points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. This equation has the form

$$y = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0).$$

Next I find its intersection with the x-axis, and this will be my next approximation:

$$x_2 = x_0 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_0).$$

In general I have the following recurrent formula:

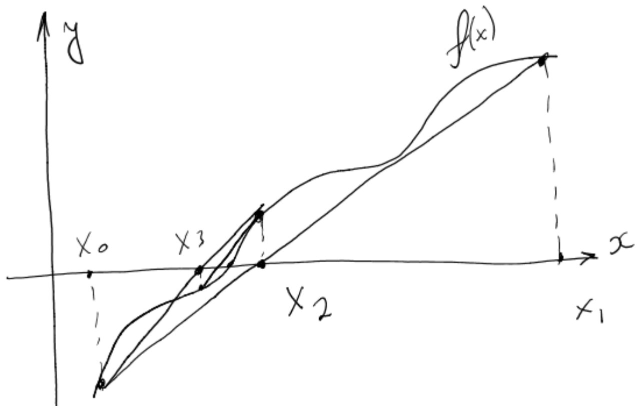
$$x_{n+1} = x_{n-1} - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_{n-1}), \quad n = 1, 2, 3, \ldots \tag{4.2}$$

So, we have some advantages, what is the disadvantage? Similar to the Newton–Raphson method the secant method does not have to converge. I will leave it for you to come up with a graph of a function such that the secant method very quickly sends root approximation beyond the bracketing interval (you can see in the figure that already the interval (x_2, x_1) does not bracket the root).

4.3 False position method

This is the method, which is almost identical to the secant method from the previous section with the only difference: at each step the root is bracketed. That is, the next approximation of the root is computed with the help of (4.2), the difference is that instead of x_n and x_{n-1} in the computations one uses x_n and the last iterate x_k, where $f(x_n)$ and $f(x_k)$ have different signs, see the figure for the illustration and compare it with the previous figure.

The false position method always converges (as the bisection method), but since at some steps it uses older points than the secant method, it can be somewhat slower. There are also situations, when bisection easily beats both the secant and false position methods in terms of the number of steps required to get sufficiently close to the root. Can you think of an example of such function (I am asking for a graph, not a formula)?



4.4 Ridders' method

4.5 Order and rate of convergence

I already mentioned several times the term *the speed of convergence*, but never defined it precisely. Let me do it here. Assume that my numerical procedure returns a sequence (x_n) of approximations of the exact root $\hat{x}$ (think about the centers of the intervals for the bisection method, the sequence of values x_n for the other methods). At each step I make an *absolute error*

$$\varepsilon_i = |x_i - \hat{x}|.$$

(Of course, in reality I do not know ε_i since I do not know $\hat{x}$, but I often can *estimate* these quantities.) If my method works, I naturally should have that $\varepsilon \to 0$. But this sequence can approach zero with various speed (whatever it means for the sequences), I will give examples later. First the precise definition.

Definition 4.1. *Let $(\varepsilon)_{n=0}^{\infty}$ be the sequence of absolute errors for a particular numerical method that converges to zero: $\varepsilon_n \to 0$ as $n \to \infty$. We say that this method has the order q and the rate of convergence μ if*

$$\lim_{n\to\infty} \frac{\varepsilon_{n+1}}{(\varepsilon_n)^q} = \mu.$$

Since the bisection method is so simple analytically, we can see what this speed is for this method. It should be clear that since we shrink our intervals twice each time, the absolute errors should also approximately half at each step, that is,

$$\frac{\varepsilon_{n+1}}{\varepsilon_n} \approx \frac{1}{2},$$

therefore, according to the definition above, this method has the order 1 (it is said to be *linear*) and rate of convergence 1/2.

How one can figure out q and μ in more complicated cases? To answer this question exactly one usually needs more involved analysis, examples will be given later. If the method was already implemented, one can try to *estimate* these quantities. Here an example of such estimate for the bisection method.

First, I will need a list of errors. Here is my modified bisection function that returns a list of errors of this method given that I already know the exact value of the root.

```python
import math

def fun1(x):
    return x*x-2

def bisection_errors(f, a, b, N, root):
    '''gets function f, the boundaries of the interval
    where a root should be, the number of bisections that will
    be performed, and the exact value of the root.
    Returns the list of absolute errors at each step.
    '''
    fa, fb = f(a), f(b)

    if fa * fb >= 0:
        raise Exception("The sings of f at the boundaries should be opposite.")

    if a > b:
        a, b = b, a #make sure that a is the left boundary and b is the right one

    errors = []

    for i in range(N):

        dx = b - a
        mid = a + dx/2
        fmid = f(mid)

        errors.append(math.log(math.fabs(mid - root)))

        if fa * fmid < 0:
            b, fb = mid, fmid
        else:
            a, fa = mid, fmid

    return errors

errors = bisection_errors(fun1, 1.2, 1.7, 30, 2**(1/2))# I use as the text x*x-2
```

Note two things. I use package `math`, since I am using the absolute value function and `log` function. More importantly, I return not the errors but their logarithms. It is important because the size of errors quickly becomes very small, and if I were to collect simply errors this list would contain values of dramatically different order of magnitude, which would be difficult to put in a figure (which I plan to do). Next, I plot $\log \varepsilon_{n+1}$ versus $\log \varepsilon_n$.

Why this strange choice? Note that from above I have

$$\varepsilon_{n+1} \approx \frac{1}{2}\varepsilon_n,$$

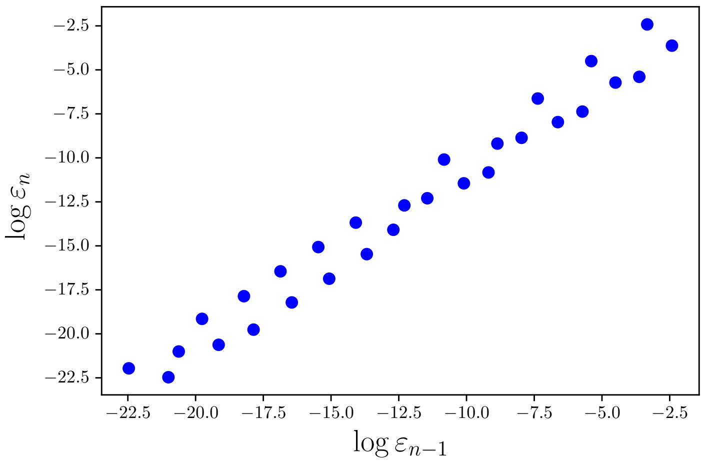
after taking logs on both sides I find

$$\log \varepsilon_{n+1} \approx \log \frac{1}{2} + \log \varepsilon_n.$$

That is, if my code works correct the data points in this particular plot should be close to the straight line with the slope 1 and shift $\log \frac{1}{2} \approx -0.693$.

```
import math
import matplotlib.pyplot as plt
%matplotlib inline

plt.rc('text', usetex=True)#I use LaTeX, you do not need to do it
plt.plot(errors[:-1],errors[1:],'bo') #taking all the errors other than the last one first
    and all but the first one
plt.xlabel(r'\LARGE $\log\varepsilon_{n-1}$') #I use LaTeX for my text, you do not need to
    do this
plt.ylabel(r'\LARGE $\log\varepsilon_{n}$')
plt.savefig('fig4_5.eps', format='eps') #saving the figure in a file to use in these notes
```



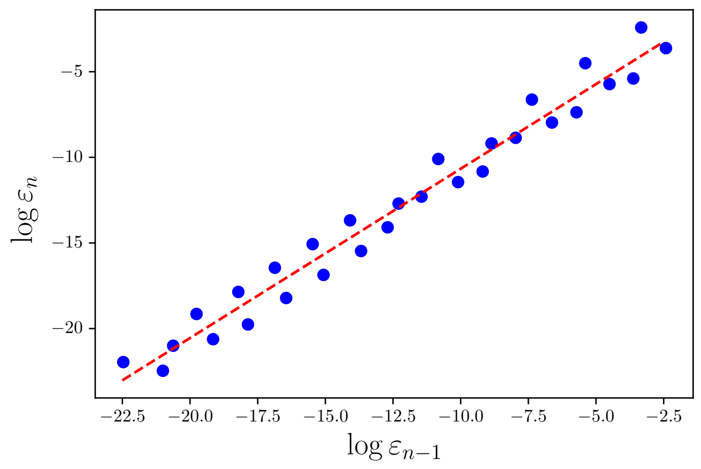
And indeed I see a straight line. Is this the right slope and shift? The following command will answer this question:

```
import numpy as np #load package numpy

m,b = np.polyfit(errors[:-1],errors[1:],deg = 1)#find the best linear approximation of my
    data (this is linear regression if you heard these words before)
#m is the slope, b is the shift
#m=0.9883961293784365
#b=-0.7836534441153487

x = np.linspace(-22.5,-2.5,10) #helps me to plot my straight line
y = [m*i+b for i in x] #helps me to plot the straight line
```

```
plt.rc('text', usetex=True)
plt.plot(errors[:-1],errors[1:],'bo')
plt.plot(x,y,'r--') #add my line to the previous figure
plt.xlabel(r'\LARGE $\log\varepsilon_{n-1}$')
plt.ylabel(r'\LARGE $\log\varepsilon_{n}$')
plt.savefig('fig4_5.eps', format='eps')
```



You can see that even though my shift is not exactly what I expected, the fit looks very good. In general, note that if

$$\frac{\varepsilon_{n+1}}{(\varepsilon_n)^q} \approx \mu$$

then

$$\varepsilon_{n+1} \approx \mu \varepsilon_n^q,$$

and after taking logs

$$\log \varepsilon_{n+1} \approx \log \mu + q \log \varepsilon_n.$$

That is, the same procedure as above but for a different method will get the approximation of q as the slope of the best linear approximation, and approximation of $\log \mu$ as the shift of the best linear approximation. You will need to do it in your first project for this semester, which I will discuss next.