Problem 1

Sphere *floating* in water, i. e. the sphere is still.

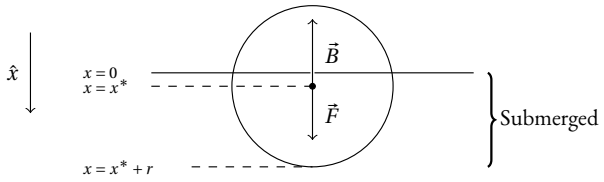

Figure 1: *Sphere in equilibrium.*

The sphere has a density of 0.6 times that of water. Archimedes' principle states that the buoyant force exerted og the sphere is equal (in magnitude) to the *weight* of the displaced fluid. Using density $\rho = 1$ for water, the weight of the displaced fluid is

$$-\vec{B} = m_{\text{fluid}} \cdot g = v_{\text{fluid}} \cdot g,$$

and we know that the weight of the sphere itself is

$$\vec{F} = m_{\text{sphere}} \cdot g = 0.6 \cdot v_{\text{sphere}} \cdot g.$$

At equilibrium, these balance out:

$$0 = \sum_{\vec{f}\text{ force}} \vec{f} = \vec{F} - \vec{B} = g(0.6 v_{\text{sphere}} - v_{\text{fluid}})$$

Obviously, the volume of displaced fluid is equal to the volume of the part of the sphere that is submerged. We divide the equation by g, and thus

$$v_{\text{submerged}} = v_{\text{fluid}} = 0.6 \cdot v_{\text{sphere}} \tag{1}$$

we also know that

$$v_{\text{sphere}} = \frac{4\pi r^3}{3} \tag{2}$$

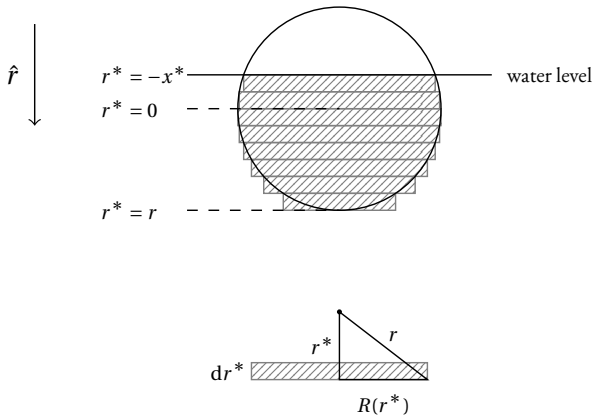

Figure 2: *Volume of the submerged part of the sphere.*

and with some analysis, we can work out the volume of the submerged part of the sphere. Our strategy is to integrate over cylinder-shaped volume elements.

Each volume element has a volume of $\pi \cdot R(r^*)^2 \cdot \mathrm{d}r^*$, and we need to integrate from water level and downward in a coordinate system on the sphere. Thus,

$$v_{\text{submerged}} = \pi \int_{r^* = -x^*}^{r^* = r} R(r^*)^2 \mathrm{d}r^* \tag{3}$$

where

$$R(r^*) = \sqrt{r^2 - (r^*)^2}.$$

We now have our mathematical model. It remains to find a solution x^* such that

$$I(x^*) := \frac{v_{\text{submerged}}}{v_{\text{sphere}}} = \frac{3}{4\pi r^3} \pi \int_{r^* = -x^*}^{r^* = r} R(r^*)^2 \mathrm{d}r^* = 0.6$$

Given an algorithm to compute definite integrals, this can probably already be solved quite easily, but we don't

know any such algorithms yet, so our strategy is to evaluate the integral by hand, and try to create a root-finding problem.

$$\begin{aligned}
I(x^*) &= \frac{3}{4r^3}\int_{r^*=-x^*}^{r^*=r} R(r^*)^2 \mathrm{d}r^* \\
&= \frac{3}{4r^3}\int_{r^*=-x^*}^{r^*=r} r^2 - (r^*)^2 \mathrm{d}r^* \\
&= \frac{3}{4r^3}\left(r^2\int_{r^*=-x^*}^{r^*=r} \mathrm{d}r^* - \int_{r^*=-x^*}^{r^*=r} (r^*)^2 \mathrm{d}r^* \right) \\
&= \frac{3}{4r^3}\left(r^2\left| r^* - \frac{1}{3}\right|_{r^*=-x^*}^{r^*=r} (r^*)^3 \right) \\
&= \frac{3}{4r^3}\left(r^2(r+x^*) - \frac{1}{3}\left(r^3 + (x^*)^3\right)\right) \\
&= \frac{3}{4r^3}\left(\frac{2}{3}r^3 + r^2x^* - \frac{1}{3}(x^*)^3 \right) \\
&= \frac{1}{2} + \frac{3}{4r}x^* - \frac{1}{4r^3}(x^*)^3 \\
&= \frac{x^*}{4r}\left(3 - \left(\frac{x^*}{r}\right)^2\right) + \frac{1}{2}
\end{aligned}$$

All that remains is to solve the equation

$$I(x^*) - 0.6 = \frac{x^*}{4r}\left(3 - \left(\frac{x^*}{r}\right)^2\right) + \frac{1}{2} - 0.6 = 0$$

Our analysis reveals that $I(x^*)$ is a polynomial of degree three, which is good because its image is all of $\mathbf{R}$, so a root like this is guaranteed to exist. *Unfortunately, three roots exist.* We need to define a criteria to determine which solution is the correct one.

Recall

x^* is the position of the center of the sphere along the $\hat{x}$-axis. Clearly, $x^* < -r$ is not a solution that can correspond to anything in the real world, as this means the sphere is hovering in mid-air. Similarily, $x^* > r$ is not a very good solution, since this is possible only if the sphere has density $\rho \geq 1$, and by assumtion $\rho = 0.6$, hence a physical solution must exist in the interval $[-r, r]$.

Approximating Solutions

We know that if the model corresponds to a physical situation, then a solution must exist in the interval $[-r, r]$. We also know that there exists exaclty three solutions in all of $\mathbf{R}$. If we can find two impossible solutions *outside of out interval*, we can be sure that the solution is unique in our interval!

If we factor our equation, we see that

$$I(x^*) - 0.6 = \frac{x^*}{4r}\left(3 - \left(\frac{x^*}{r}\right)^2\right) + \frac{1}{2} - 0.6$$

now, we look for solutions to the equation $I(x^*) - 0.5 = 0$. These solutions will be quite close to the solutions of $I(x^*) - 0.6$, and are super easy to find.

$$\begin{aligned}
I(x^*) - 0.5 &= \frac{x^*}{4r}\left(3 - \left(\frac{x^*}{r}\right)^2\right) = 0 \\
\implies x^* = 0 \quad &\text{or} \quad 3 - \left(\frac{x^*}{r}\right)^2 = 0\,.
\end{aligned}$$

This yields two solutions,

$$x^* = \pm\sqrt{3}r\,,$$

which are both far outside the interval $[-r, r]$. Not only does this mean that it is safe to search for the original solution on this interval by bisection, but the last solution

($x^* = 0$) of the simplified problem tells us that it is going to be smart to start searching close to zero, which may helps us approximating the solution faster.

This does *not* prove that Newtons method will converge on the *entire* interval, but starting at $x_0 = 0$, it will, and we will find the solution very quickly.

Problem 2

a. Want to use the Taylor series of f around the point a, $T_a\{f\}(x)$, to show that

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x),$$

in other words, that the error we expect if we compute

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x},$$

is proporional to Δx, i. e. the error goes to zero as Δx approaches zero.

Fix x and Δx. We expand the Taylor series to two terms, *also including the error term*, around the point x.

$$\begin{aligned} f(x^*) &= T_x\{f\}(x^*) \\ &= f(x) + f'(x)(x - x^*) + \mathrm{error}(2) \end{aligned}$$

Which yields

$$f'(x) = \frac{f(x^*) - f(x) - \mathrm{error}(2)}{x - x^*}$$

By evaluating the series at $x^* = x + \Delta x$

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{\mathrm{error}(2)}{\Delta x}$$

To show that the error term is $\mathcal{O}(\Delta x)$ is straight forward:

$$E(\Delta x) := \frac{\mathrm{error}(2)}{\Delta x} = \frac{1}{\Delta x}\frac{f''(\widetilde{x})}{2!}(x^* - x)^2$$

for some $\widetilde{x} \in [x, x^*]$. Again, we evaluate the error term at $x^* = x + \Delta x$.

$$\begin{aligned} &= \frac{f''(\widetilde{x})}{2\Delta x}(\Delta x)^2 \\ &= \frac{f''(\widetilde{x})}{2}\Delta x \end{aligned}$$

as long as f'' does not blow up on $I = [x, x + \Delta x]$. Because of the fact that I is bounded and closed, f'' attains its maximum and minimum on I. Thus we can let

$$L = \operatorname{diam} f(I) = \max_{x,y \in I} \left| f''(x) - f''(y) \right|$$

Which makes

$$|E(\Delta x)| \leq L\Delta x.$$

Testing the approximation

We want to compute the error commited by our formula for some function which we know how to differentiate symbolically. We want to compute

$$\left| \frac{\mathrm{d}}{\mathrm{d}x} \cos x. \right|_{x = \pi/4}$$

We know that the symbolic derivative of $\cos x$ is $-\sin x$.

Listing 1: *Approximate differentiation program*

```
1  # src/alg.py
2  def ddx(f, dx):
3      return lambda x: (f(x + dx) - f(x)) / dx
```

`ddx` returns a function of type `float -> float` that closes over `f` and `dx`. This new function in turn evaluates the derivative at a point `x`.

Listing 2: Using the differentiation program

```
import src.alg as alg
import numpy as np
from math import sin, cos, pi

def errors(deltas):
    for dx in deltas:
        dcosdx = alg.ddx(cos, dx)
        approx = dcosdx(pi/4)
        exact  = -sin(pi/4)
        yield (abs(approx - exact), dx)

dxs = (1/2**k for k in range(30))
data = list(errors(dxs))
error, delta = zip(*data)
```

Plotting `error` against `delta` in a logarithmic coordinate system confirms that the error is indeed linear in Δx.

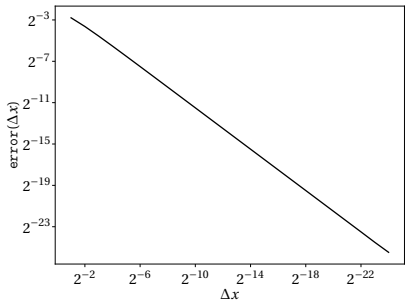

Figure 3: *The absolute error of the approximation as a function of* Δx *for* $f(x) = \cos x$ *and* $x = \pi/4$

b.

Listing 3: Iterating with `sqrt`

```
from math import sqrt
import random as rand

def iter_sqrt(N):
    x  = rand.random() * 100
    xh = x

    for l in range(1, N):
        for k in range(1, l):
            xh = sqrt(xh)
        for k in range(1, l):
            xh = xh**2

        e  = abs(x - xh)
        er = abs(x - xh) / abs(x)

        yield (e, er, l)

data = list(iter_sqrt(N = 100))
es, ers, ls = zip(*data)
```

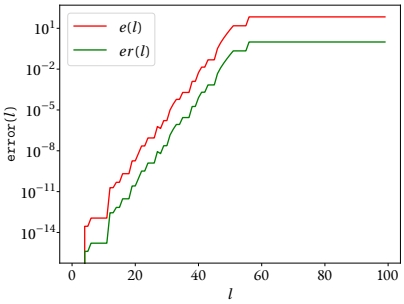

Figure 4: *The absolute and relative error committed by the algorithm.*

Problem 3

I will use the interval $I = [1, 5]$, because we agreed to that in class. This is because the methods do not converge on $[-1, 5]$.

Listing 4: *Bisection and Newtons method*

```
MAX_ITER = 100

def bisect(f, I):
    a, b = I

    assert a < b
    assert f(a) * f(b) < 0

    for _ in range(MAX_ITER):
        m = 0.5 * (a + b)
        yield m
        if f(a) * f(m) < 0:
            b = m
        elif f(b) * f(m) < 0:
```

```
            a = m
        else: continue # then f(m) == 0, do nothing


def fpi(g, x0, tol=10E-6):
    # g contraction, finds x = g(x) by iteration
    xn    = g(x0) # x1
    xn_m1 = x0    # x0
    for _ in range(MAX_ITER):
        yield xn
        xn, xn_m1 = g(xn), xn


def newton(f, dfdx, x0, tol=10E-6):
    g = lambda x: x - f(x) / dfdx(x)
    yield from fpi(g, x0)

def pairs(it):
    # this technique is deprecated, but whatever
    # used to get pairwise consecutive elements from
    # an iterator
    x = next(it)
    y = next(it)
    while True:
        yield(x, y)
        x, y = y, next(it)
```

As you can see, we can implement Newtons method as a generic fixed-point iteration. I have opted for differentiating f symbolically and providing it to `newton`, but one might use `ddx` as defined previously.

The functions are defined in a way, such that they yield consecutively better approximations, instead of returning the best approximation. This is just to illustrate the convergence. Moreover, we are interested in two types of convergence, which the following figures show.

Listing 5: *Comparing the two methods*

```
def g(x): return x**5 - 4*x + 2
def dgdx(x): return 5*x**4 - 4

n = list(alg.newton(g, dgdx, 1))
b = list(alg.bisect(g, (1, 5)))

def abs_err(data):
    for xn, xn_p1 in alg.pairs(iter(data)):
        yield abs(xn - xn_p1)

i, err_abs_n = zip(*list(enumerate(abs_err(n))))
_, err_abs_b = zip(*list(enumerate(abs_err(b))))

def f_err(f, data):
    for xn in data:
        yield abs(f(xn))

j, err_f_n = zip(*list(enumerate(f_err(g, n))))
_, err_f_b = zip(*list(enumerate(f_err(g, b))))
```

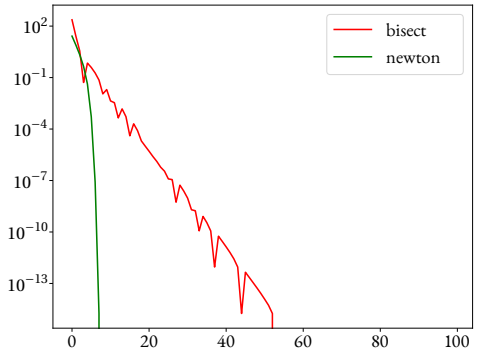

Figure 5: *Speed of convergence* $|f(x_k)|$.

Somehow, the bisection method converges in a very erratic way.

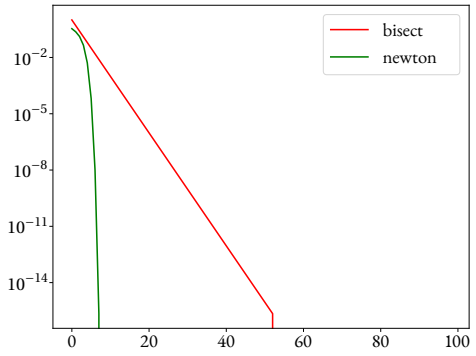

Figure 6: *Speed of convergence* $|x_{k+1} - x_k|$.

Plotting the errors versus `i` and `j`, which are the iteration count, we see *clearly* that Newtons method converges quadratically, while bisection converges linearly.