

Math 151A Homework #3

Nathan Solomon

November 8, 2024

Problem 0.1.

(a)

$$g(x) = \frac{1}{2} \left(x + \frac{a}{x} \right)$$
$$g(\sqrt{a}) = \frac{1}{2} \left(\sqrt{a} + \frac{a}{\sqrt{a}} \right) = \frac{1}{2} (\sqrt{a} + \sqrt{a}) = \sqrt{a}$$

- (b) We just showed $g(p) = p$, and since $g'(x) = \frac{1}{2} \left(1 - \frac{a}{x^2} \right)$, we also have $g'(p) = 0$. But since $g''(x) = \frac{a}{4x^3}$, $g''(p) = 1/(4\sqrt{a}) \neq 0$. $g \in C^2([a, b])$ is a function such that $g(p) = p$, $g'(p) = 0$, and $g''(p) \neq 0$, so by the theorem from lecture 7 on FPI convergence rate, p_n will converge quadratically (that is, with order $\alpha = 2$) to p for p_0 sufficiently close to p .

Problem 0.2.

- (a) $f(x) = e^x - 1 - x - x^2/2 = x^3/6 + x^4/24 + x^5/120 + \dots$, so $f(0) = f'(0) = f''(0) = 0$, but $f'''(0) = 1 \neq 0$. Therefore $x = 0$ is a zero of f with multiplicity 3.
- (b) Since the multiplicity of $x = 0$ is not 1, we are not guaranteed even linear convergence with Newton's method, but we are still guaranteed quadratic convergence with the modified version of Newton's method. That's why the modified version converges so much faster:

```
import math
```

```
def f(x):
```

```
    return math.exp(x) - 1 - x - x**2/2
```

```
def f_prime(x):
```

```
    return math.exp(x) - 1 - x
```

```
def f_prime_prime(x):
```

```
    return math.exp(x) - 1
```

```
def mu(x):
```

```
    return f(x) / f_prime(x)
```

```
def mu_prime(x):
```

```
    return 1 - f(x) * f_prime_prime(x) / f_prime(x)**2
```

```
def newtons_method(x_0, tolerance, max_iterations):
```

```
    print(f"\nNewton's method with {x_0=}, {tolerance=}, {max_iterations=}")
```

```

x_n = x_0
n = 0
residual = abs(f(x_n))
print(f"{n=:02}__{x_n=:+1.8f}__{residual=:1.17f}")
while abs(x_n) > tolerance and n < max_iterations:
    x_n -= f(x_n) / f_prime(x_n)
    n += 1
    residual = abs(f(x_n))
    print(f"{n=:02}__{x_n=:+1.8f}__{residual=:1.17f}")

def modified_newtons_method(x_0, tolerance, max_iterations):
    print(f"\nModified_Newton's_method_with_{x_0=},{tolerance=},{max_iterations=}")
    x_n = x_0
    n = 0
    residual = abs(f(x_n))
    print(f"{n=:02}__{x_n=:+1.8f}__{residual=:1.17f}")
    while abs(x_n) > tolerance and n < max_iterations:
        x_n -= mu(x_n) / mu_prime(x_n)
        n += 1
        residual = abs(f(x_n))
        print(f"{n=:02}__{x_n=:+1.8f}__{residual=:1.17f}")

newtons_method(1, 1e-6, 1000)
modified_newtons_method(1, 1e-6, 1000)

```

```

Newton's_method_with_x_0=1,tolerance=1e-06,max_iterations=1000
n=00__x_n=+1.00000000__residual=0.21828182845904509
n=01__x_n=+0.69610560__residual=0.06753849522061861
n=02__x_n=+0.47811290__residual=0.02061871165303474
n=03__x_n=+0.32528512__residual=0.00623499263339255
n=04__x_n=+0.21985780__residual=0.00187302505944356
n=05__x_n=+0.14793388__residual=0.00056013544459508
n=06__x_n=+0.09923640__residual=0.00016700016443702
n=07__x_n=+0.06643295__residual=0.00004968763000134
n=08__x_n=+0.04441177__residual=0.00001476321366354
n=09__x_n=+0.02966279__residual=0.00000438240689480
n=10__x_n=+0.01979969__residual=0.00000130009935675
n=11__x_n=+0.01321069__residual=0.00000038553289180
n=12__x_n=+0.00881198__residual=0.00000011429490871
n=13__x_n=+0.00587681__residual=0.00000003387760031
n=14__x_n=+0.00391883__residual=0.00000001004026650
n=15__x_n=+0.00261298__residual=0.00000000297537972
n=16__x_n=+0.00174218__residual=0.00000000088169007
n=17__x_n=+0.00116154__residual=0.00000000026126051
n=18__x_n=+0.00077440__residual=0.00000000007741430
n=19__x_n=+0.00051628__residual=0.00000000002293816
n=20__x_n=+0.00034419__residual=0.00000000000679660
n=21__x_n=+0.00022947__residual=0.00000000000201381
n=22__x_n=+0.00015298__residual=0.00000000000059670
n=23__x_n=+0.00010200__residual=0.00000000000017688
n=24__x_n=+0.00006799__residual=0.00000000000005248
n=25__x_n=+0.00004529__residual=0.00000000000001558
n=26__x_n=+0.00003009__residual=0.00000000000000465

```

```

n=27 x_n=+0.00001982 residual=0.000000000000000123
n=28 x_n=+0.00001356 residual=0.000000000000000033
n=29 x_n=+0.00000996 residual=0.000000000000000021
n=30 x_n=+0.00000569 residual=0.000000000000000005
n=31 x_n=+0.00000869 residual=0.000000000000000007
n=32 x_n=+0.00000693 residual=0.000000000000000006
n=33 x_n=+0.00000458 residual=0.000000000000000002
n=34 x_n=+0.00000261 residual=0.000000000000000007
n=35 x_n=-0.00001870 residual=0.000000000000000113
n=36 x_n=-0.00001223 residual=0.000000000000000031
n=37 x_n=-0.00000805 residual=0.000000000000000008
n=38 x_n=-0.00000566 residual=0.000000000000000005
n=39 x_n=-0.00000227 residual=0.000000000000000001
n=40 x_n=-0.00000506 residual=0.000000000000000003
n=41 x_n=-0.00000760 residual=0.000000000000000011
n=42 x_n=-0.00000381 residual=0.000000000000000001
n=43 x_n=-0.00000305 residual=0.000000000000000001
n=44 x_n=-0.00000423 residual=0.000000000000000005
n=45 x_n=+0.00000168 residual=0.000000000000000001
n=46 x_n=-0.00000534 residual=0.000000000000000008
n=47 x_n=+0.00000014 residual=0.000000000000000000

```

```

Modified Newton's method with x_0=1, tolerance=1e-06, max_iterations=1000
n=00 x_n=+1.00000000 residual=0.21828182845904509
n=01 x_n=-0.11308312 residual=0.00023435150458198
n=02 x_n=-0.00103017 residual=0.00000000018216402
n=03 x_n=-0.00000009 residual=0.00000000000000003

```

- (c) If we change the tolerance from 10^{-6} to 10^{-10} , Newton's method doesn't converge, even in 1000 iterations. This is not surprising – what did surprise me was to see that the modified version of Newton's method also did not converge, even in 1000 iterations. Although that modified method should converge quadratically in theory, round-off errors in the denominator of the iterate got in the way of that.

Problem 0.3.

\hat{p}_n converges to $p = 1$ significantly faster than p_n does.

```

p = 1
def p(n):
    return 1 + 1 / n
def p_hat(n):
    return p(n) - (p(n+1) - p(n))*2 / (p(n+2) - 2 * p(n+1) + p(n))

for n in range(1, 8):
    print(f" {n}={p(n)} {p_hat(n)=1.5 f}")

```

```

n=1 p(n)=2.00000 p_hat(n)=1.25000
n=2 p(n)=1.50000 p_hat(n)=1.16667
n=3 p(n)=1.33333 p_hat(n)=1.12500
n=4 p(n)=1.25000 p_hat(n)=1.10000
n=5 p(n)=1.20000 p_hat(n)=1.08333

```

n=6 p(n)=1.16667 p_hat(n)=1.07143
n=7 p(n)=1.14286 p_hat(n)=1.06250

Problem 0.4.

(a)

$$P(x) := \sum_{k=0}^n f(x_k) L_{n,k}(x), \quad L_{n,k}(x) := \prod_{i \in [0,n] \cap \mathbb{Z} - \{k\}} \frac{x - x_i}{x_k - x_i}.$$

Let $n = 2, x_0 = 1, x_1 = 2, x_2 = 3$. Then

$$L_{2,0} = \frac{(x-2)(x-3)}{(1-2)(1-3)} = \frac{x^2 - 5x + 6}{2}$$

$$L_{2,1} = \frac{(x-1)(x-3)}{(2-1)(2-3)} = -x^2 + 4x - 3$$

$$L_{2,2} = \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{x^2 - 3x + 2}{2}$$

$$P(x) = \ln(2)(-x^2 + 4x - 3) + \ln(3) \frac{x^2 - 3x + 2}{2}.$$

(b) **import** math

def P(x):

return math.log(2) * (-x**2+4*x-3) + math.log(3) * (x**2-3*x+2)/2

print(f"{P(1)=}")

print(f"{P(2)=}")

print(f"{P(3)=}")

print(f"{P(1.5)=}")

print(f"{P(2.4)=}")

P(1)=0.0

P(2)=0.6931471805599453

P(3)=1.0986122886681098

P(1.5)=0.3825338493364452

P(2.4)=0.889855072497425

When $x = 1.5$, the absolute error is 0.0229312587717192 and the relative error is 0.05655544290534098.

When $x = 2.4$, the absolute error is 0.014386335143525164 and the relative error is 0.016432722871416054.

(c) This is a pretty bad method, but it shows that the error is maximized when $x = 1.367$, and that the error at that point is 0.024817.

def error(x):

return abs(P(x) - math.log(x))

import numpy as np

for x **in** np.linspace(1,3,100):

print(f"x=:1.5f} \ {error(x)=:1.5f}")

print(' \n')

x=1.34343 error(x)=0.02474

x=1.36364 error(x)=0.02482

x=1.38384 error(x)=0.02479

for x **in** np.linspace(1.34,1.39,100):

print(f"x=:1.5f} \ {error(x)=:1.10f}")

```
# x=1.36677 error(x)=0.0248176530
# x=1.36727 error(x)=0.0248177110
# x=1.36778 error(x)=0.0248177061
```

Problem 0.5.

(a)

$$P(x) = -0.00252225x^5 + 0.2866292x^4 - 10.793792x^3 + 157.31208x^2 + 1642.7517x + 179323$$

```
import numpy as np
# x is years since 1960, y is United States population (in thousands)
x = [0, 10, 20, 30, 40, 50]
y = [179_323, 203_302, 226_542, 249_633, 281_422, 308_746]
N = 5
assert len(x) == N + 1 and len(y) == N + 1
V = np.vander(x, N+1)
coeffs = np.linalg.inv(V) @ np.matrix([y]).T
print("P(x) = " + " ".join([f"{coeffs[i,0]:+} x^{N-i}" for i in range(N+1)]))
def P(x):
    return round((np.matrix([x**(N-i) for i in range(N+1)]) @ coeffs)[0, 0])
for x in [0, 10, 20, 30, 40, 50, 60]:
    print(f"year {1960+x}: {P(x):6}")
actual_value = 329_500
print(f"Relative error: {abs(P(60) - actual_value) / actual_value}")
```

(b) This method significantly underestimates the US population in 2020. This is an example of the Runge phenomenon.

```
year 1960: P(x)=179323
year 1970: P(x)=203302
year 1980: P(x)=226542
year 1990: P(x)=249633
year 2000: P(x)=281422
year 2010: P(x)=308746
year 2020: P(x)=266165
Relative error: 0.1922154779969651
```