Math 4610 – HW 5

Jake Daniels A02307117

October 24, 2022

Task 1:

To start task 1 I wrote the following code for the implicit Euler method:

```
import numpy as np
from Methods.newt import newtonsMethod

def implicitEuler(f, df, x0, t0, T, n):
    h = (T - t0) / n
    f0 = f(t0, x0)
    tval = []
    xval = []
    tval.append(t0)
    xval.append(x0)
    for i in range(1, n):
        t1 = t0 + h
        x1 = newtonsMethod(lambda x: x - x0 - h * f(t1, x), lambda x: 1 - h * df(t1, x), x0, .00000001, 50, "n")
        x0 = x1
        t0 = t1
        tval.append(t0)
        xval.append(x0)
    return tval, xval
```

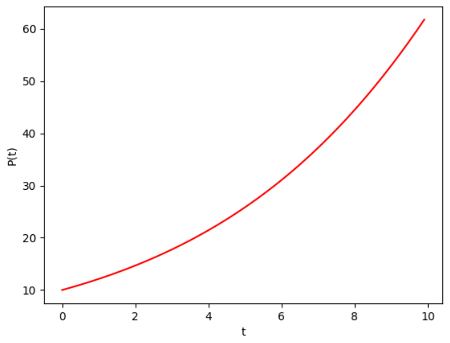
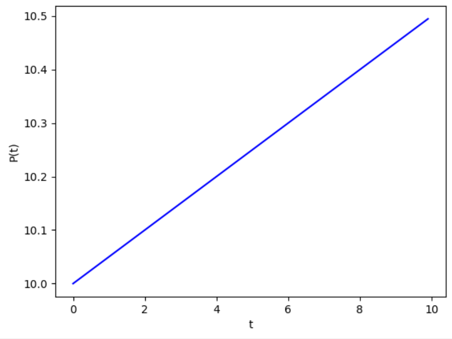
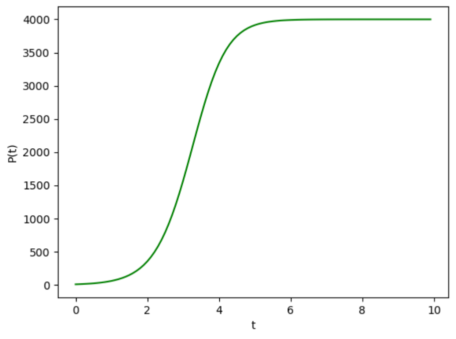
I modified my original Newton's Code and rewrote it for this project so I'm going to paste it below (I had to modify it since my original code asked the user for the function, derivative, etc and I didn't want to have to do that n amount of times):

```
import numpy as np

def table(a,b,c):
    print("(1)" , a, " (2) ", b, " (3) ", c)

def newtonsMethod(f, df, x0, tol, maxIter, choice):
    error = 10 * tol
    iter = 0
    if choice == "y":
        while error > tol and iter < maxIter:
            f0 = f(x0)
            df0 = df(x0)
            x1 = x0 - f0 / df0
            error = np.abs(x1 - x0)
            table(iter + 1, x1, error)
            iter = iter + 1
            x0 = x1
    elif choice == "n":
        while error > tol and iter < maxIter:
            f0 = f(x0)
            df0 = df(x0)
            x1 = x0 - f0 / df0
            error = np.abs(x1 - x0)
            iter = iter + 1
            x0 = x1
    return x0
```

I then created the following file to pass a given logistic equation into the implicit Euler method for some alpha and beta:

```
import numpy as np
import matplotlib.pyplot as plt
from Methods.explicitEuler import explicitEuler
from Methods.implicitEuler import implicitEuler

def logisticEquation(a, b, p0):
    p = lambda t, x: a * x - b * x * x
    dp = lambda t, x: a - 2 * b * x
    tval, pval = implicitEuler(p, dp, p0, 0, 10, 100)
    return tval, pval

tApprox1, xApprox1 = logisticEquation(0.2, 0.0005, 10.0)
tApprox2, xApprox2 = logisticEquation(0.01, 0.0005, 10.0)
tApprox3, xApprox3 = logisticEquation(2.0, 0.0005, 10.0)
plt.plot(tApprox1, xApprox1, color='red')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
plt.plot(tApprox2, xApprox2, color='blue')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
plt.plot(tApprox3, xApprox3, color='green')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
```

Running this code yielded the following graphs as an output:

Task 2:

For task 2 we are going to start by solving the ODE analytically.

$$P' = \alpha P - \beta P^2$$
$$\frac{1}{\alpha P - \beta P^2}\frac{dP}{dt} = 1 \implies \frac{1}{-P^2(\beta - \alpha/P)}\frac{dP}{dt} = 1$$
$$\int \frac{1}{-P^2(\beta-\alpha/P)}\frac{dP}{dt}dt = \int \frac{1}{-P^2(\beta-\alpha/P)}dP = \int dt = t + C$$

Let $u = \beta - \frac{\alpha}{P}$ then $du = \frac{\alpha}{P^2}dP$ and we can rearrange such that $-\frac{1}{\alpha}du = -\frac{1}{P^2}dP$, substituting this into the equation we get:

$$\int \frac{1}{-P^2(\beta-\alpha/P)}dP = -\frac{1}{\alpha}\int \frac{1}{u}du = -\frac{1}{\alpha}ln|u| = -\frac{1}{\alpha}ln|\beta - \frac{\alpha}{P}| = t + C$$

Doing some algebra gets us down to the equation:

$$P = \frac{\alpha}{\beta + Ce^{-\alpha t}}$$

Using the condition $P(0) = P_0$ we can solve for C and get that $C = \frac{\alpha}{P_0} - \beta$, plugging this in and doing some more algebra yields our final analytic answer of:

$$P = \frac{\alpha P_0 e^{\alpha t}}{\alpha + \beta P_0(e^{\alpha t} - 1)}$$

Using the above equation, I wrote the following code to calculate values using the exact equation:

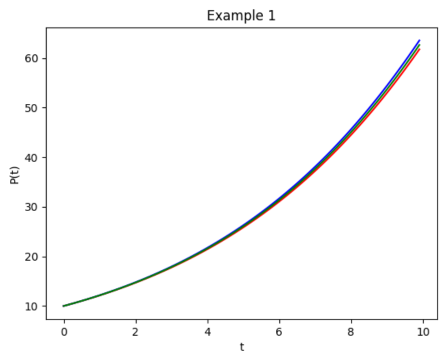
```
def exactLogistic(a, b, p0, T, n):
    h = T / n
    tval = []
    pval = []
    t0 = 0
    tval.append(t0)
    pval.append(p0)
    for i in range(1, n):
        t1 = t0 + h
        p1 = (a * p0 * np.exp(a * t1)) / (a + b * p0 * (np.exp(a * t1) - 1))
        tval.append(t1)
        pval.append(p1)
        t0 = t1
    return tval, pval
```

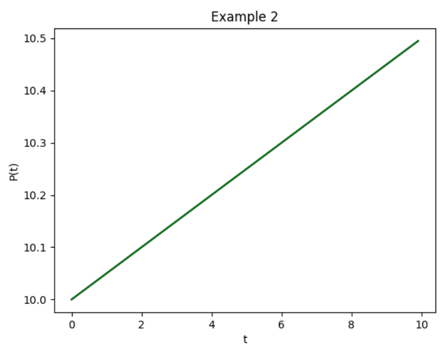
Task 3:

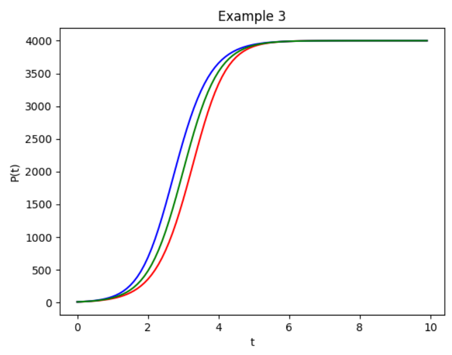
I modified the code above to compare the explicit, implicit, and exact equations:

```python
import numpy as np
import matplotlib.pyplot as plt
from Methods.explicitEuler import explicitEuler
from Methods.implicitEuler import implicitEuler

def logisticEquation(a, b, p0):
    p = lambda t, x: a * x - b * x * x
    dp = lambda t, x: a - 2 * b * x
    tval1, pval1 = explicitEuler(p, p0, 0, 10, 100)
    tval2, pval2 = implicitEuler(p, dp, p0, 0, 10, 100)
    tval3, pval3 = exactLogistic(a, b, p0, 10, 100)
    return tval1, pval1, tval2, pval2, tval3, pval3

tApprox1, xApprox1, tApprox2, xApprox2, tApprox3, xApprox3 = logisticEquation(0.2, 0.0005, 10.0)
tApprox4, xApprox4, tApprox5, xApprox5, tApprox6, xApprox6 = logisticEquation(0.01, 0.0005, 10.0)
tApprox7, xApprox7, tApprox8, xApprox8, tApprox9, xApprox9 = logisticEquation(2.0, 0.0005, 10.0)
plt.title("Example 1")
plt.plot(tApprox1, xApprox1, color='red')
plt.plot(tApprox2, xApprox2, color='blue')
plt.plot(tApprox3, xApprox3, color='green')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
plt.title("Example 2")
plt.plot(tApprox4, xApprox4, color='red')
plt.plot(tApprox5, xApprox5, color='blue')
plt.plot(tApprox6, xApprox6, color='green')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
plt.title("Example 3")
plt.plot(tApprox7, xApprox7, color='red')
plt.plot(tApprox8, xApprox8, color='blue')
plt.plot(tApprox9, xApprox9, color='green')
plt.xlabel('t')
plt.ylabel('P(t)')
plt.show()
```

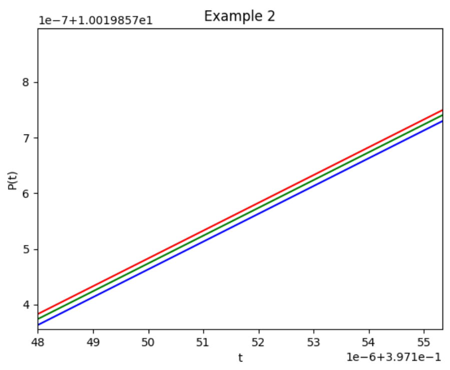
These output the following graphs:







The explicit Euler method, uses tangent lines to approximate our function, whereas the implicit Euler method uses secant lines. This creates a difference in their approximations. If a function is concave ($f'' > 0$), then by definition it will lie above its tangent line and below its secant line. If a function is convex ($f'' < 0$) it will have the opposite property. Meaning when a function is concave the explicit Euler method will under approximate the exact solution and the implicit Euler method will over approximate the exact solution. This can be seen in the graphs for examples 1 and 3 as the blue line (implicit approximation) is greater than the green line (exact), and the red line (explicit) is less than the green line. This is the opposite for the graph for example 2. From this we can conclude that the exact solution for the given alpha, beta, and initial P value, is concave for example 1 and 3 and convex for example 2. (Note: the graph below is a zoomed in version of graph 2)



Task 4:

For the trapezoid rule, I created the following code:

```
import numpy as np

def trapezoid(f, a, b, n):
    h = (b - a) / n
    sum = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        x = a + i * h
        fx = f(x)
        sum = sum + fx
    sum = sum * h
    return sum
```

I created the following code to test the code for n=2,4,8,16:

```
for i in range(1, 5):
    n = 2 ** i
    sum = trapezoid(lambda x: np.exp(- x * x), 0, np.pi / 4, n)
    print("n = ", n, " ", sum)
```

That outputted the following:

```
n =  2   0.6388862805734845
n =  4   0.6471507696813964
n =  8   0.6491991053630145
n =  16   0.6497100964398593
```

This sequence of numbers appears to be approaching some number around .6497 to .6498.

Task 5:

I created the following code to approximate an integral using Simpson's Rule:

```
import numpy as np

def simpsons(f, a, b, n):
    h = (b - a) / n
    sum  = f(a) + f(b)
    for i in range(1, n):
        x = a + i * h
        fx = f(x)
        if i % 2 == 0:
            sum = sum + 2 * fx
        else:
            sum = sum + 4 * fx
    sum = sum * h / 3
    return sum
```

Here is the code I used to test the code for the same integral as given in task 4:

```
for i in range(1, 5):
    n = 2 ** i
    sum= simpsons(lambda x: np.exp(- x * x), 0, np.pi / 4, n)
    print("n = ", n, " ", sum)
```

That yielded a very similar output to a problem above:

```
n =  2   0.6503097748895396
n =  4   0.6499055993840338
n =  8   0.6498818839235537
n =  16   0.6498804267988076
```

This converges to a similar value around .6498.

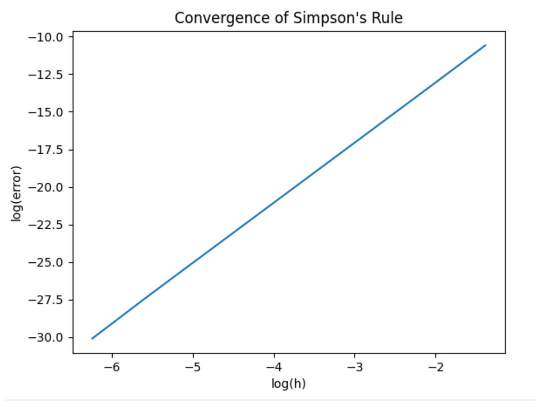
Now for the estimation of accuracy. I wrote the following code to test our approximations for $h = \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\ldots$. I computed the error by taking the absolute value of the difference between my approximations and the (approximate) exact value which I got by letting n=1000 for my code above. The code is as follows:

```
import numpy as np
from Methods.simpsons import simpsons
import matplotlib.pyplot as plt

def compConv(n):
    logError = []
    logH = []
    f = lambda x: np.exp(- x * x)
    exact = simpsons(f, 0, np.pi / 4, 1000)
    for i in range(0,8):
        sum = simpsons(f, 0, np.pi / 4, n)
        error = np.abs(exact - sum)
        logError.append(np.log(error))
        logH.append(np.log(1 / n))
        n = n * 2
    return logH, logError
```

I then wrote the following code to plot the actual log-log plot:

```
logH, logError = compConv(4)
plt.plot(logH, logError)
plt.xlabel("log(h)")
plt.ylabel("log(error)")
plt.title("Convergence of Simpson's Rule")
plt.show()
```

Which yielded the following graph:


Convergence of Simpson's Rule
log(error)
log(h)