AMATH301_Homework3_writeup

January 26, 2023

1 Homework 3 writeup solutions

1.1 Name: Aqua Karaman

1.1.1 Section C

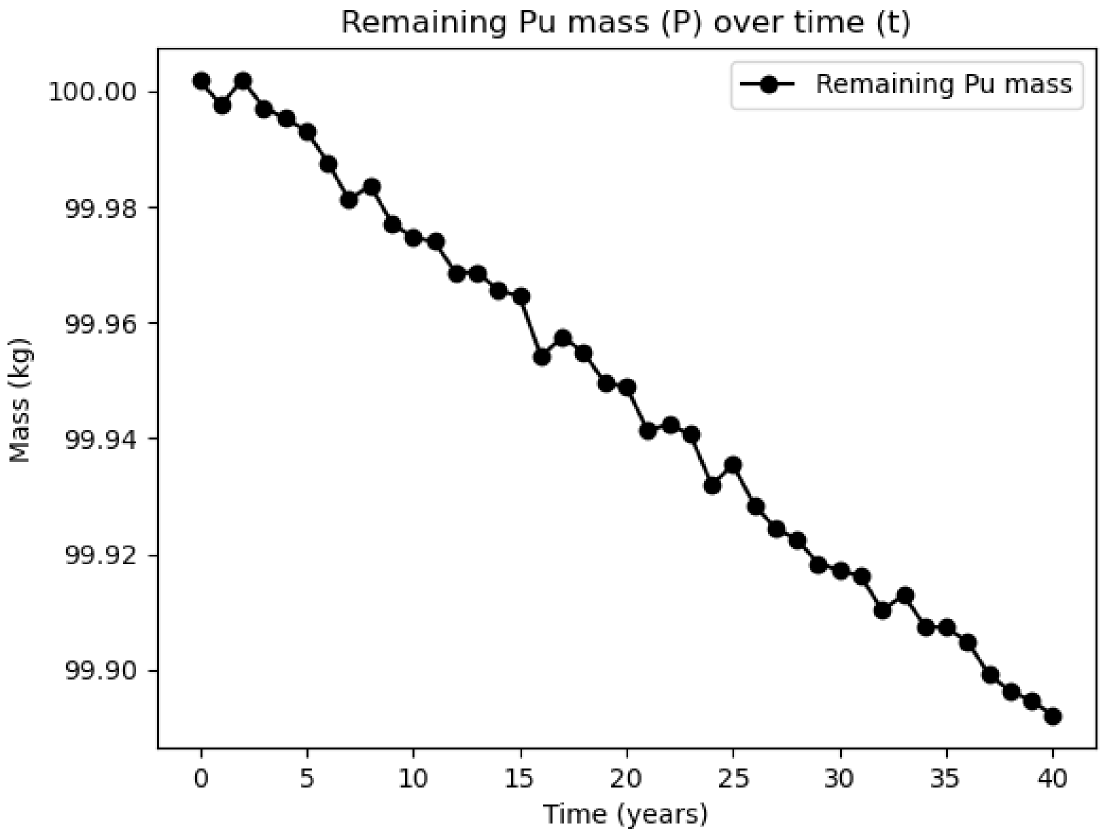
1.2 Problem 1

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

############## Problem 1 ################
M = np.genfromtxt('Plutonium.csv', delimiter=',')
t = M[0, :]
P = M[1, :]

## Part a
# Compute h from the t array
h = t[1] - t[0]
A1 = h

## Part b
fd = (P[1] - P[0])/h
A2= fd

## Part c
bd = (P[-1] - P[-2])/h
A3 = bd

## Part d
# Uncomment the line below to get A4
nd_fd = (-3*P[0] + 4*P[1] - P[2])/(2*h)
A4 = nd_fd

## Part e
nd_bd = (3*P[-1] - 4*P[-2] + P[-3])/(2*h) # nd for nd in second bc why not lol
A5 = nd_bd
```

```python
## Part f
# You may want to use a for loop here
deriv = np.zeros(41)
deriv[0] = nd_fd
deriv[-1] = nd_bd
for k in range(1, len(t)-1):
        deriv[k] = (P[k+1] - P[k-1])/(2*h) # fill in here

A6 = deriv

## Part g
decay_rate = -1/P * deriv
A7 = decay_rate

## Part h
mean_decay = np.average(decay_rate)
A8 = mean_decay

## Part i
half_life = np.log(2) / mean_decay
A9 = half_life

## Part j
nd_deriv = (-2*P[22] + P[23] + P[21])/h**2
A10 = nd_deriv
```

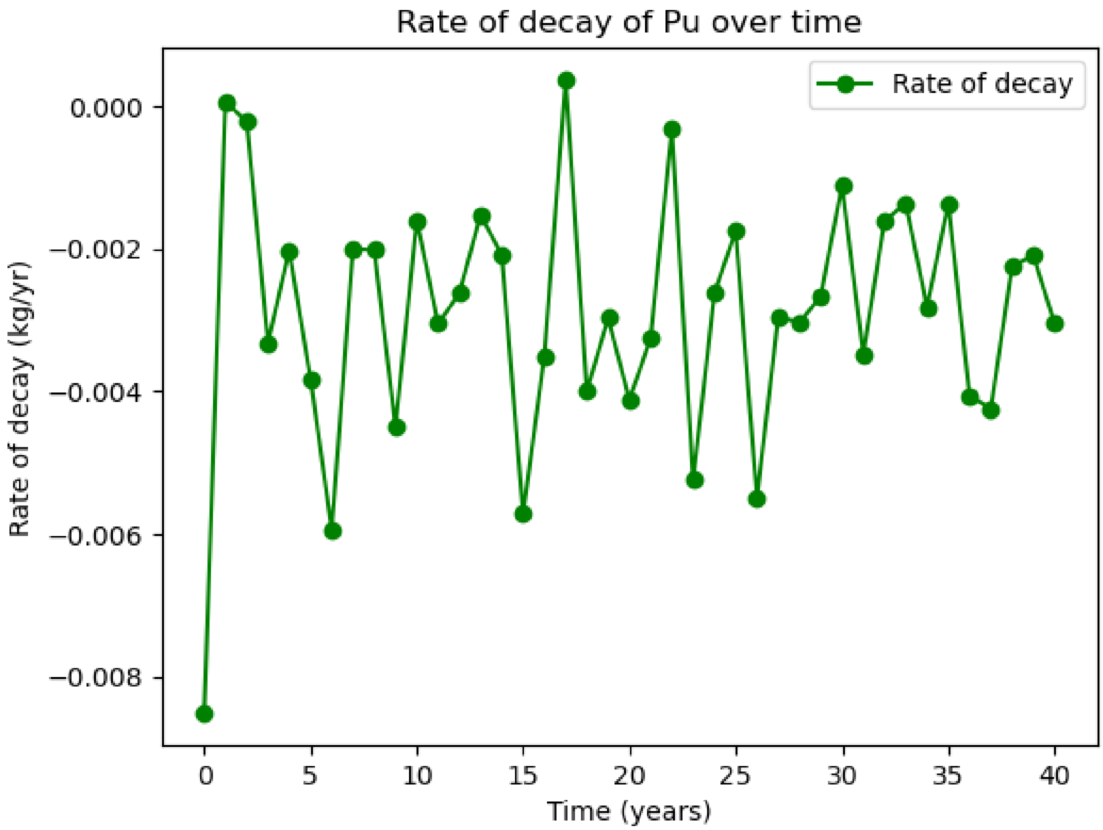
1.2.1 Part a

[ ]:
```python
plt.figure('data')
plt.plot(t, P, '-ok', label='Remaining Pu mass')
plt.title('Remaining Pu mass (P) over time (t)')
plt.xlabel('Time (years)')
plt.ylabel('Mass (kg)')
plt.legend()
plt.show()
```



1.2.2 Part b

[ ]:
```
plt.figure('derivative')
plt.plot(t, deriv, '-og', label='Rate of decay')
plt.title('Rate of decay of Pu over time')
plt.xlabel('Time (years)')
plt.ylabel('Rate of decay (kg/yr)')
plt.legend()
plt.show()
```



1.2.3 Part c

The plot of the derivative (shown in Part b) is extremely jaggedy and inconsistent, rather than smooth. The beginning three sets of data points are surprising, as there is a quite large jump between the first two visually, and a very small jump between the second and third points. Overall, it makes sense the graph doesn't display a smooth curve, as the data itself is not smooth—that is, it isn't cotinuous, but recorded once every year, unlike a smooth curve.

1.2.4 Part d

Because the data is super jumpy and frankly inconsistent, it makes sense to use the arithmetic mean to calculate the half-life. Additionally, the data appears to jump up and down often enough that an average between a few points would likely still produce useful, more consistent data.

1.3 Problem 2

[ ]:
```
############# Problem 2 ###############
# You are going to want to define the integrand as an anonymous function.
mu = 85
sigma = 8.3
integrand = lambda x: np.exp(-(x-mu)**2/(2*sigma**2))/np.sqrt(2*np.pi*sigma**2)
```

```
# Let's also define the left and right bounds of the integral
left = 110
right = 130

## Part a
scipy_int = scipy.integrate.quad(integrand, left, right)
A11 = scipy_int[0]
print(A11)

## Part b
# To define the h array, we can take 2 to the power of an array.
power = -np.linspace(1, 16, 16)
# Now create h from that array!
h = 2**(power)
lhr     = np.zeros(16)
rhr     = np.zeros(16)
mpr     = np.zeros(16)
trap    = np.zeros(16)
simpson = np.zeros(16)

for k in range(16):
        xlhr = np.arange(left, right+h[k], h[k])
        ylhr = integrand(xlhr)
        lhr[k] = h[k] * np.sum(ylhr[:-1])
A12 = lhr

for k in range(16):
        xrhr = np.arange(left, right+h[k], h[k])
        yrhr = integrand(xrhr)
        rhr[k] = h[k] * np.sum(yrhr[1:])
A13 = rhr

for k in range(16):
        xmpr = np.arange(left, right+h[k], h[k])
        x_avg = (xmpr[:-1] + xmpr[1:])/2
        ympr = integrand(x_avg)
        mpr[k] = h[k] * np.sum(ympr[:-1])
A14 = mpr

for k in range(16):
        xtrap = np.arange(left, right+h[k], h[k])
        ytrap = integrand(xtrap)
        trap[k] = (h[k]/2) * (ytrap[0] + 2 * np.sum(ytrap[1:-1]) + ytrap[-1])
A15 = trap

for k in range(16):
```

```
        xsimpson = np.arange(left, right+h[k], h[k])
        ysimpson = integrand(xsimpson)
        simpson[k] = (h[k] / 3) * (ysimpson[0] + 4*np.sum(ysimpson[1:-1:2]) +
 ↪2*np.sum(ysimpson[2:-2:2]) + ysimpson[-1])
A16 = simpson
```

```
0.0012974274669396554
[0.00143006 0.00136277 0.00132985 0.00131358 0.00130549 0.00130145
 0.00129944 0.00129843 0.00129793 0.00129768 0.00129755 0.00129749
 0.00129746 0.00129744 0.00129744 0.00129743] [0.00117258 0.00123403 0.00126549
0.0012814  0.0012894  0.00129341
 0.00129542 0.00129642 0.00129692 0.00129718 0.0012973  0.00129736
 0.0012974  0.00129741 0.00129742 0.00129742] [0.00129547 0.00129694 0.0012973
0.0012974  0.00129742 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743] [0.00130132 0.0012984  0.00129767
0.00129749 0.00129744 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743] [0.00129743 0.00129743 0.00129743
0.00129743 0.00129743 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743 0.00129743
 0.00129743 0.00129743 0.00129743 0.00129743]
```

1.3.1 Part a

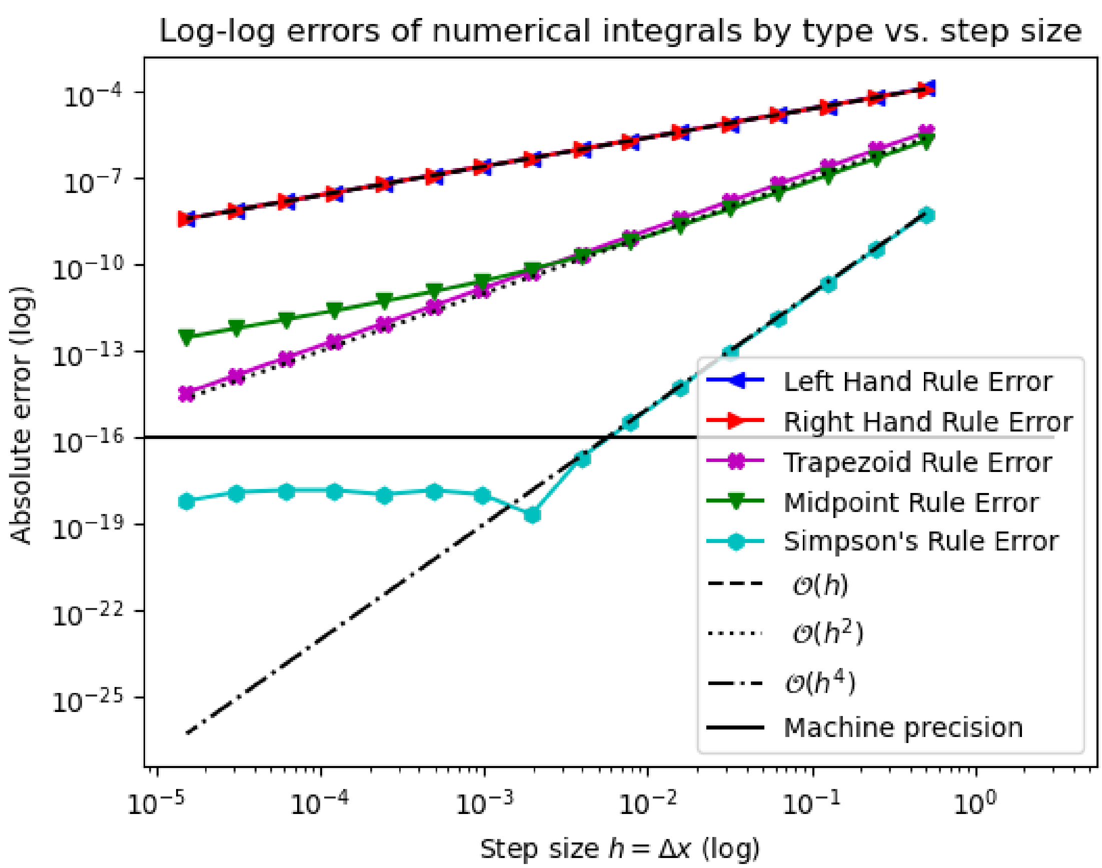
[ ]:
```
lhr_error = np.abs(lhr - scipy_int[0])
rhr_error = np.abs(rhr - scipy_int[0])
trap_error = np.abs(trap - scipy_int[0])
mpr_error = np.abs(mpr - scipy_int[0])
simpson_error = np.abs(simpson - scipy_int[0])
```

1.3.2 Part b-f

[ ]:
```
c = 10**(-3.6)
C = 10**(-5)

plt.figure('errors')

plt.loglog(h, lhr_error, '-b<', label='Left Hand Rule Error')
plt.loglog(h, rhr_error, '-r>', label='Right Hand Rule Error')
plt.loglog(h, trap_error, '-mX', label='Trapezoid Rule Error')
plt.loglog(h, mpr_error, '-gv', label='Midpoint Rule Error')
plt.loglog(h, simpson_error, '-ch', label="Simpson's Rule Error")

plt.loglog(h, c*h, '--k', label=' $\mathcal{O}(h)$ ')
plt.loglog(h, C*h**2, ':k', label=' $\mathcal{O}(h^2)$ ')
plt.loglog(h, 10**(-7)*h**4, '-.k', label='$\mathcal{O}(h^4)$')
```

```
plt.plot([10**(-16), 10**(-16), 10**(-16), 10**(-16)], '-k', label='Machine␣
 ↪precision')

plt.xlabel('Step size $h = \Delta x$ (log)')
plt.ylabel('Absolute error (log)')

plt.title('Log-log errors of numerical integrals by type vs. step size')

plt.legend()

print(trap_error)
print(simpson_error)
plt.show()
```

```
[3.89162321e-06 9.73173804e-07 2.43310202e-07 6.08285976e-08
 1.52072148e-08 3.80180780e-09 9.50452204e-10 2.37613066e-10
 5.94032666e-11 1.48508159e-11 3.71270289e-12 9.28174855e-13
 2.32042250e-13 5.80100204e-14 1.45025051e-14 3.62535522e-15]
[5.71532127e-09 3.57333631e-10 2.23353057e-11 1.39598554e-12
 8.72481341e-14 5.45201904e-15 3.39355280e-16 1.99493200e-17
 2.16840434e-19 1.08420217e-18 1.51788304e-18 1.08420217e-18
 1.51788304e-18 1.51788304e-18 1.30104261e-18 6.50521303e-19]
```



1.3.3 Part g - discussion

(i) From the plot above, it's very clear Simpson's Rule has the best accuracy (more specifically, to the forth order). The plot show's the error for Simpson's Rule is even less than "Machine Precision," 10^{-16}.

(ii) As the step size for Simpson's Rule gets smaller (specifically $2^{-9} \approx 0.002$) the error simply stops decreasing regularly and takes a plateau pattern. I assume the error stops decreasing due to the computer's inability to calculate that far down (below 10^{-19}) with Python.