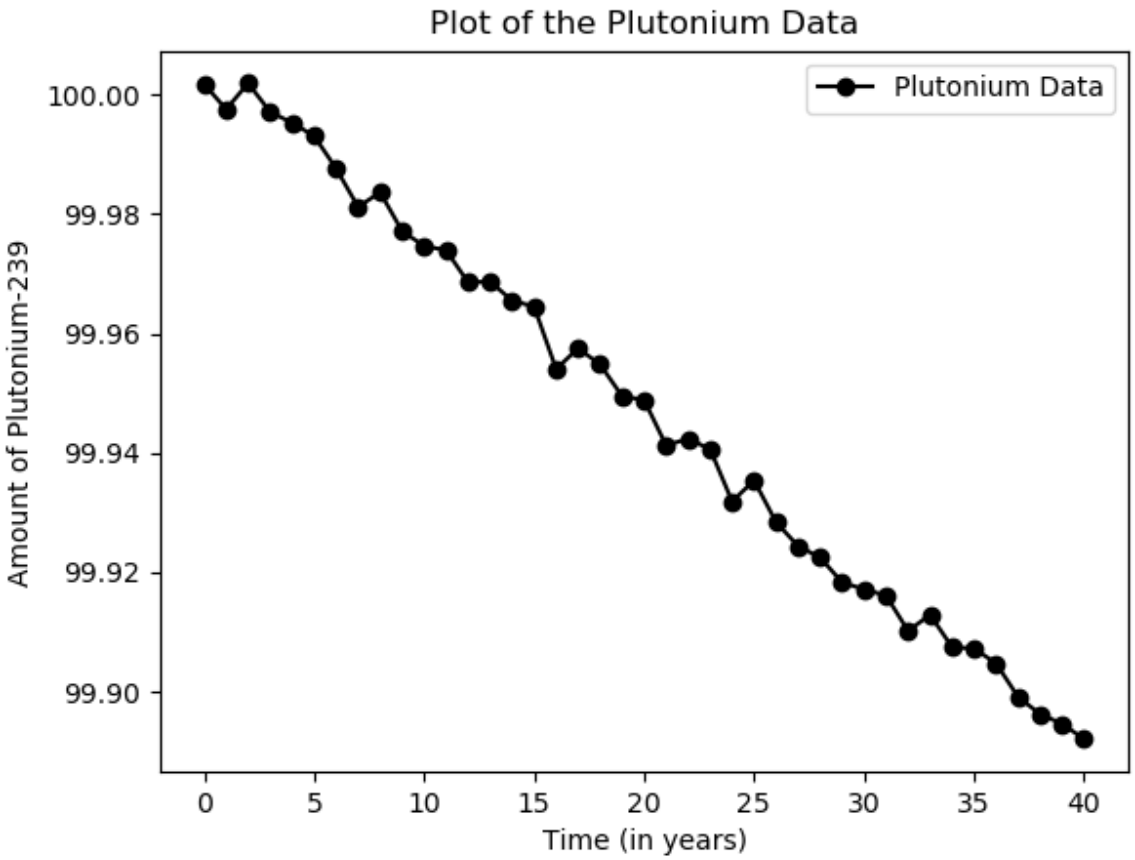
AMATH 301 Homework 3 Writeup

Jaiden Atterbury – Section B

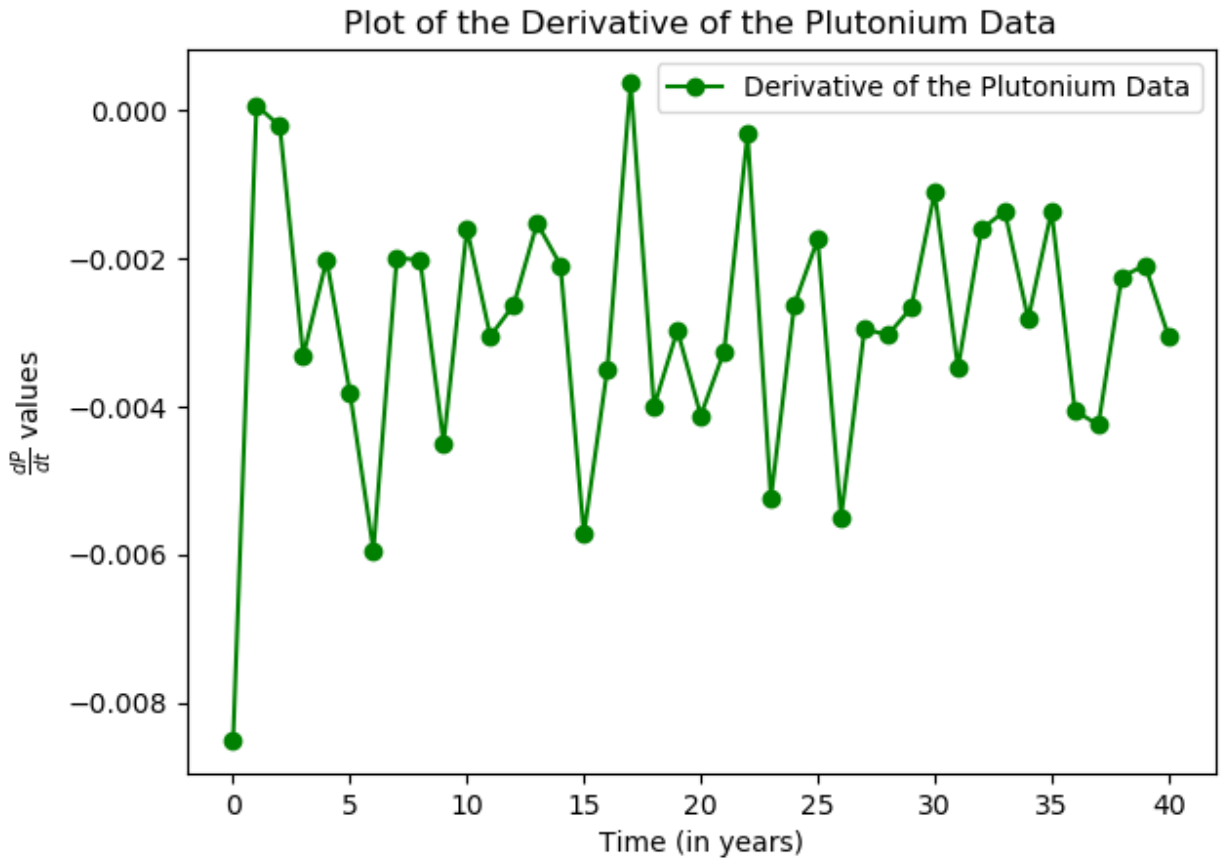
1/22/2023

Problem 1

Part a:
Load in the data and plot the data as black markers with black lines in between ('-ok'). Add axis labels, legends, and a title.



Part b:
Load in the data and plot the data as black markers with black lines in between ('-ok'). Add axis labels, legends, and a title.Create a new figure (you can use plt.figure()). On it, plot the derivative that you calculated using the second order methods, A6. Plot this using green markers with green lines between the markers ('-og'). Include axis labels, legends, and a title.



Part c:

Q: Comment on what you see in the plot of the derivative. You should see something that looks strange/confusing. Why is that? Why isn't it a smooth curve?

A: As can be seen from the plot of the derivative, the resulting curve is not smooth. Instead there are sharp edges with no consistent constant, increasing or decreasing trend from t = 0 to t = 40. This strange/confusing curve occurs because even though there is a clear decreasing and linear trend in the data plot, this linear trend is not "perfectly" linear. Instead, there are small increases and decreases in the amount of plutonium while in general the plutonium amount is decreasing linearly with respect to time. Since these small changes in the amount of plutonium happen throughout the entirety of the data set, the derivative thus shows no meaningful trend when plotted.

Part d:

Q: Based on what you see, why is it a good idea to use the mean of the calculated decay rate to calculate the half life.

A: As mentioned in the explanation to part (c), the plot of the plutonium data shows a clear decreasing linear relationship between the amount of plutonium left and time elapsed in years. However, since this relationship isn't perfectly linear we don't see a distinct trend in our derivative data. However, since we know this relationship is linear we'd expect the derivative to be a horizontal line at the slope value of the plot of the plutonium data. Thus, in order to get a constant decay rate we must take an average of all of these increases and decreases that appear in the derivative plot. Once we take the average of these rates of change, we will have a better estimate of the "expected" derivative value, thus our half life calculation will be more accurate than if we didn't take the average and just use the calculated derivative data.

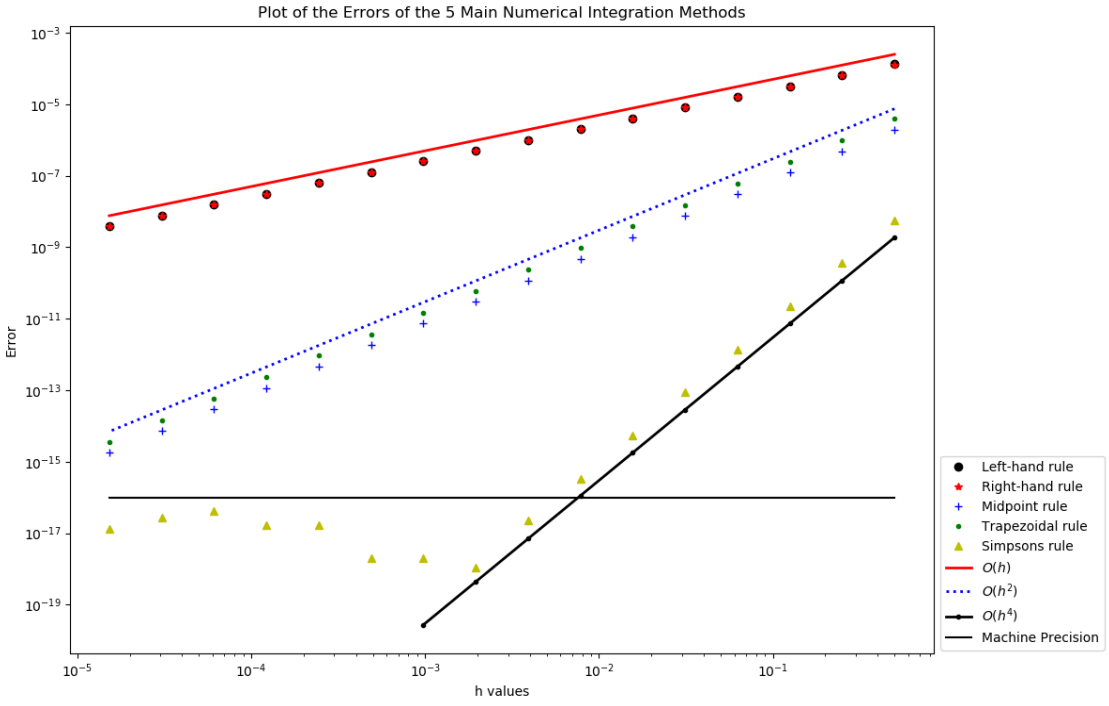
Code used:

```
# Problem 1:
M = np.genfromtxt('Plutonium.csv', delimiter=',')
xdata = M[0, :]
ydata = M[1, :]

h = xdata[1] - xdata[0]
sofd = 1/2*h * (-3*ydata[0] + 4*ydata[1] - ydata[2])
sobd = 1/2*h * (3*ydata[40] - 4*ydata[39] + ydata[38])

fp_centered = np.zeros(ydata.shape)
fp_centered[0] = sofd
for k in range(1, len(xdata)-1):
   fp_centered[k] = 1/(2*h)*(ydata[k+1] - ydata[k-1])
fp_centered[-1] = sobd

# Part (a): Load in the data and plot the data as black markers with black
# lines in between ('-ok'). Add axis labels, legends, and a title.
plt.plot(xdata, ydata, "-ok", label="Plutonium Data")
plt.xlabel("Time (in years)")
plt.ylabel("Amount of Plutonium-239")
plt.title("Plot of the Plutonium Data")
plt.legend()
plt.show()

# Part (b): Create a new figure (you can use plt.figure()). On it, plot the
```

```
# derivative that you calculated using the second order methods, A6. Plot this
# using green markers with green lines between the markers ('-og'). Include
# axis labels, legends, and a title.
plt.figure()
plt.plot(xdata, fp_centered, "-og", label="Derivative of the Plutonium Data")
plt.xlabel("Time (in years)")
plt.ylabel(r'$\frac{dP}{dt}$ values')
plt.title("Plot of the Derivative of the Plutonium Data")
plt.legend()
plt.show()
```

Problem 2

Part a-f:
Plot the errors versus the step size for each of the five methods
using a log-log plot. Use a different color and marker type for each method.
On the same figure, plot a trend line that represents O(h^n). Add a black horizontal line at 10−16 which is machine precision. Add appropriate labels
to the x and y axes, a legend, and a title.



Part g – discussion

(i)
Q: Which method has the highest order of accuracy? How can you tell from the plot?

A: The method that has the highest order of accuracy out of the 5 techniques of the numerical integration used in this problem is Simpson's Rule. We can see this is true by looking at the plot of the errors of all of the 5 methods used. In particular, we want to look at the slope of these different methods because the slope of the line shows which order of error the method is. Furthermore, Simpsons rule is order 4, while the other methods are of order 2 and 1. Thus, Simpson's rule has the highest order of accuracy.

(ii)
Q: What is happening to the error for Simpson's rule with a very small step size? Why does the error stop decreasing here?

A: For Simpson's rule with a very small step size, the error diverges from the trend line and begins increasing with each subsequent decrease of step size. This confusing/counterintuitive trend can be explained through the lens of machine accuracy. Since computers can only store so much information, when doing arithmetic with floating point numbers, there is an associated error that comes with each operation done to one of these numbers. In particular, since these very small step size values lead to very large values of n when approximating the integral, just as we saw in homework 2, the more iterations of operations on floating point numbers, the higher the error is going to be (unless the number is stored in base 2). Hence, when the value of the step size exceeds this machine precision threshold, the error associated with each term begins to increase. Thus when deciding a step size, we need to find the number that optimizes the error in the sense that we get the lowest amount of error without exceeding this precision threshold.

Code used:

```
# Problem 2:
coef = 1 / np.sqrt(2 * np.pi * (8.3) ** 2)
f = lambda x: coef * np.exp((-(x-85) ** 2) / (137.78))
h_values = np.logspace(-1, -16, 16, base=2.0)

# True value:
true_value = scint.quad(f, 110, 130)[0]

# Left-hand rule:
approx_left = np.zeros(16)
for index, h in enumerate(h_values):
   x_vals = np.arange(110, 130 + h, h)
```

```
    y_vals = f(x_vals)
    approx_left[index] = h * sum(y_vals[:-1])

# Right-hand rule:
approx_right = np.zeros(16)
for index, h in enumerate(h_values):
    x_vals = np.arange(110, 130 + h, h)
    y_vals = f(x_vals)
    approx_right[index] = h * sum(y_vals[1:])

# Midpoint rule:
approx_mid = np.zeros(16)
for index, h in enumerate(h_values):
    x_vals = np.arange(110, 130, h) + h/2
    y_vals = f(x_vals)
    approx_mid[index] = h * sum(y_vals)

# Trapezoidal rule:
approx_trap = np.zeros(16)
for index, h in enumerate(h_values):
    x_vals = np.arange(110, 130 + h, h)
    y_vals = f(x_vals)
    approx_trap[index] = h/2 * (y_vals[0] + 2 * sum(y_vals[1:-1]) + y_vals[-1])

# Simpson's rule:
approx_simp = np.zeros(16)
for index, h in enumerate(h_values):
    x_vals = np.arange(110, 130 + h, h)
    y_vals = f(x_vals)
    approx_simp[index] = h/3 * (y_vals[0] + 4 * sum(y_vals[1:-1:2]) + 2 *
sum(y_vals[2:-2:2]) + y_vals[-1])

# Part (a): For each of the five methods used to calculate S, create an array
# of length 16 containing the absolute value of the error for the different
# step sizes.
left_error = np.abs(approx_left - true_value)
right_error = np.abs(approx_right - true_value)
mid_error = np.abs(approx_mid - true_value)
trap_error = np.abs(approx_trap - true_value)
simp_error = np.abs(approx_simp - true_value)

# Part (b): Plot the errors versus the step size for each of the five methods
```

```
# using a log-log plot. Use a different color and marker type for each method.
plt.figure()
plt.loglog(h_values, left_error, "ok", label="Left-hand rule")
plt.loglog(h_values, right_error, "*r", label="Right-hand rule")
plt.loglog(h_values, mid_error, "+b", label="Midpoint rule")
plt.loglog(h_values, trap_error, ".g", label="Trapezoidal rule")
plt.loglog(h_values, simp_error, "^y", label="Simpsons rule")

# Part(c):
plt.loglog(h_values, (0.0005) * h_values, "-r", label=r'$O(h)$', linewidth=2)
plt.loglog(h_values, (0.00003) * h_values ** 2, ":b", label=r'$O(h^2)$', linewidth=2)
plt.loglog(h_values[0:10], (0.00000003) * h_values[0:10] ** 4, ".-k",
label=r'$O(h^4)$', linewidth=2)
plt.loglog(h_values, np.array([10 ** -16] * 16), "k", label="Machine Precision")
plt.xlabel("h values")
plt.ylabel("Error")
plt.title("Plot of the Errors of the 5 Main Numerical Integration Methods")
plt.legend(loc="bottom right", bbox_to_anchor=(1.00001, 0.325))
plt.show()
```