1. The code has been written and explained and uploaded to GitHub titled "problem_1.py". The solution relies on the ability of our function to store the previously calculated values of $f(x)$ to use in our next recursion as we already have the function at the bounds and midpoint calculated by the previous recursion. Therefore, only the two points on either side of the middle must be calculated. If we run the program for a few typical functions:

$$y_1 = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} \quad (-5 \leq x \leq 5), \tag{1}$$

$$y_2 = \sqrt{x} \quad (0 \leq x \leq 5), \tag{2}$$

$$y_3 = |x| \quad (-5 \leq x \leq 5), \tag{3}$$

$$y_4 = x \ln x \quad (0.5 \leq x \leq 5), \tag{4}$$

$$y_5 = \sqrt{1 - x^2} \quad (-1 \leq x \leq 1), \tag{5}$$

we can determine the difference between the number of function calls made by the original integrator and our new improved one with code (as shown in the python file) or analytically. Analytically we can predict that for each time after the first recursion that the old integrator will call the function 5 times, and the new integrator will only call the function twice as the other 3 points have been calculated previously. As a result, if the old integrator calls the function n times, then the new one will call the function $5 + 2(n-5)/5$ times. The output of the code is shown below.

```
[Running] python -u "/Users/jehandastoor/Phys512/assignment2/problem_1.py"
Gaussian number of reduced calls = 66
√x number of reduced calls = 96
|x| number of reduced calls = 6
xln(x) number of reduced calls = 24
Semi-circle of radius 1 number of reduced calls = 150

[Done] exited with code=0 in 0.923 seconds
```

Figure 1: Python output of the difference between the number of calls made by the old integrator and our new integrator.

We see that there has been a significant reduction in the number of calls made to $f(x)$ as predicted.

2. The code and explanation for the Chebyshev polynomial fit has been written and uploaded to GitHub under "chebyshev.py". The number of terms required can be determined using a while loop in python with the termination condition being once the maximum error drops below 10^{-6} (as done in "problem_2.py"). The order of the truncated Chebyshev polynomial was chosen to be 20 and the number of terms required was determined to be 8 (order 7) in order to have a maximum error of 10^{-6}. We can plot the residuals for the truncated and untruncated Chebyshev fits and the np.polynomial.legendre.legfit fit,

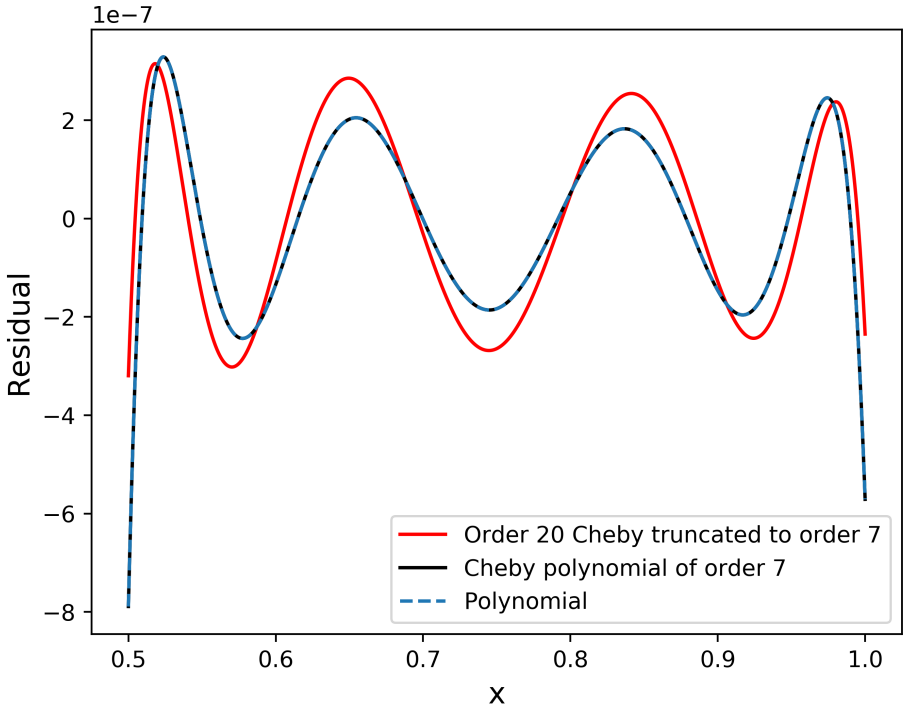

Figure 2: Python plot of the residuals associated with the truncated and untruncated Chebyshev polynomial fit as well as the np.polynomial.legendre.legfit fit for $\log_2(x)$ between $0.5 \leq x \leq 1$.

Comparing the RMS and maximum errors between the truncated and untruncated Chebyshev fits and the np.polynomial.legendre.legfit fit, we get the following output:

```
[Running] python -u "/Users/jehandastoor/Phys512/assignment2/problem_2.py"
Truncated Chebyshev Polynomial fit: truncated order = 7 , max error = 3.196978302089093e-07 , RMS error = 1.9201623254698075e-07
Untruncated Chebyshev Polynomial of order = 7 , max error = 7.888699555813616e-07 , RMS error = 1.6860950830770662e-07
Polynomial fit: max error = 7.888700004343718e-07 , RMS error = 1.6860950824547397e-07

[Done] exited with code=0 in 12.57 seconds
```

Figure 3: Python output of the maximum and RMS errors associated with the truncated and untruncated Chebyshev polynomial fit as well as the np.polynomial.legendre.legfit fit for $\log_2(x)$ between $0.5 \leq x \leq 1$.

As we can see from the output, the truncated chebyshev polynomial has the lowest max error while the np.polynomial fit has the lowest RMS error (although the np.polynomial fit and the untruncated chebyshev polynomial fit both have similar errors).

3. a) The code to solve for the decay products of ^{238}U with respect to time has been uploaded to GitHub under the title "problem_3.py". The implicit ODE solver should be and was used to solve this stiff ODE as otherwise a large number of time-steps would be required to calculate the result (i.e. the program would be less efficient/take more time). As a result, we use the "Radau" method associated with scipy.integrate.solve_ivp.

b) We can plot the ratio of ^{206}Pb to ^{238}U over the range $[0, 10]$ billion years. As most of the half-lives are relatively small compared to the half-life of ^{238}U, we can approximate the increase in ^{206}Pb to have a doubling time that is approximately the same as the half-life of ^{238}U. Hence, we expect the ratio of ^{206}Pb to ^{238}U after a time equivalent to the half-life of ^{238}U (i.e. 4.468 billion years) should be 1. The plot can be seen below.

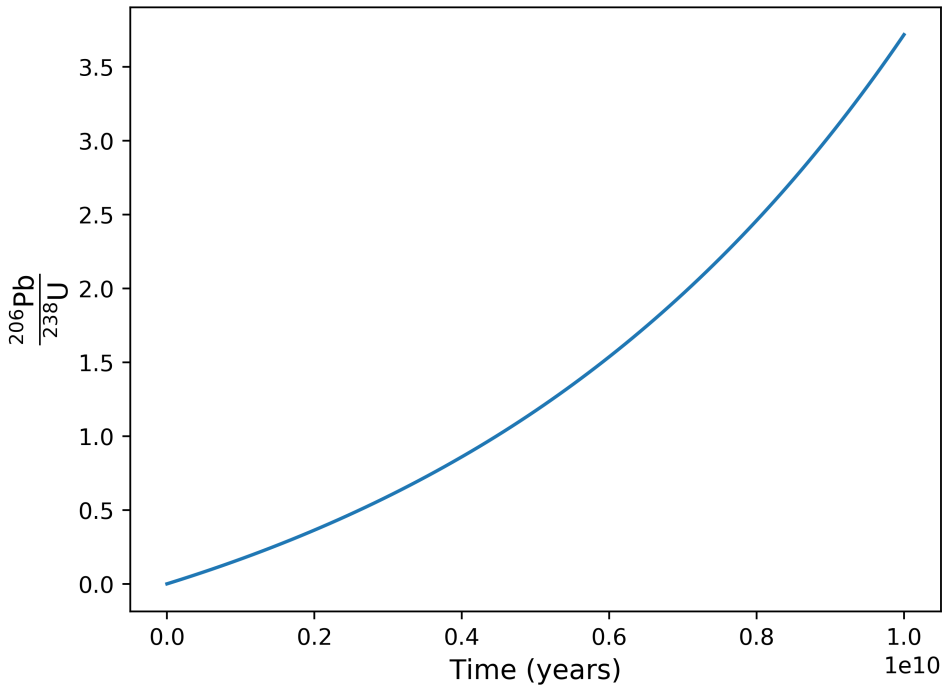

Figure 4: The ratio of ^{206}Pb to ^{238}U plotted as a function of time, assuming that we started with a pure source of ^{238}U, over 10 billion years.

Looking at the plot, we see that the ratio is 1 after a time approximately equal to the half-life of ^{238}U as expected. We can also plot the ratio of ^{230}Th to ^{234}U over the region [0,50] million years to get an interesting result. The plot has been shown below.

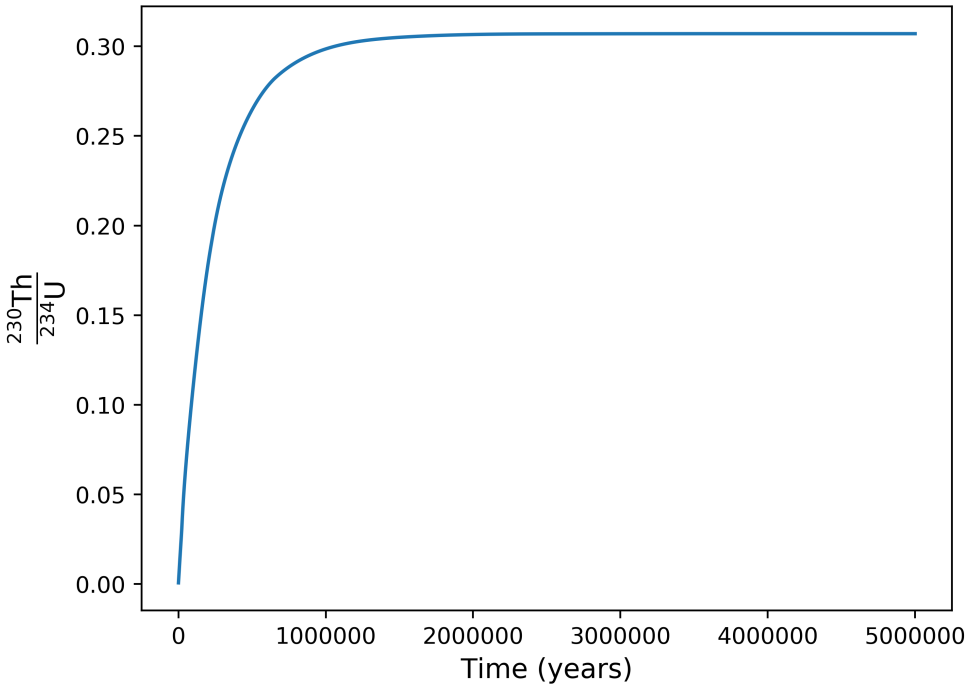

Figure 5: The ratio of ^{230}Th to ^{234}U plotted as a function of time, assuming that we started with a pure source of ^{238}U, over 50 million years.

We see that the ratio of both elements increases till around 15 million years where it plateaus around 0.3. When dating different rocks we can use either of these two methods. If we find that the ratio of ^{230}Th to ^{234}U is approximately 0.3, then we know that the rock is likely greater than 15 million years old and we can then compare the ratio of ^{206}Pb to ^{238}U, using Figure 4, to determine its actual age (assuming the quantity of ^{206}Pb is exponentially increasing with the equivalent half life of ^{238}U). However, if the ratio of ^{230}Th to ^{234}U is less than 0.3, we know that the rock is likely younger than 20 million years old and we can use Figure 5 to figure out its exact age.