PHYS 512 Assignment 2

Liam Fitzpatrick

September 2022

1 Question 1

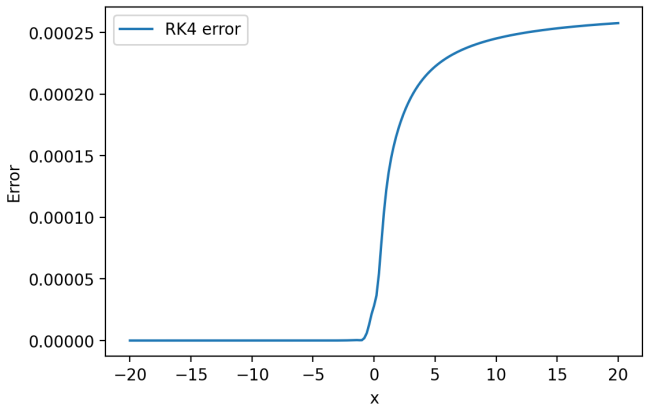
Using the same RK4 method outlined in the lecture notes, the function `rk4_step(fun,x,y,h)` was coded and the number of function calls was counted (for later):

```
def rk4_step(fun,x,y,h):                #Start count of fun calls:
    k1=h*fun(x,y)                       #+1
    k2=h*fun(x+h/2,y+k1/2)              #+1
    k3=h*fun(x+h/2,y+k2/2)              #+1
    k4=h*fun(x+h,y+k3)                  #+1
    return y+(k1+2*k2+2*k3+k4)/6        #Total 4 fun calls
```

This stepper function was then iterated over each interval (x from -20 to 20 with 200 intervals):

```
nsteps=200
npt=nsteps+1
x=np.linspace(-20,20,npt)
y=np.zeros(npt)
y[0]=1  #Initial condition
for i in range(nsteps):     #Iterate stepper over each interval
    h=x[i+1]-x[i]
    y[i+1]=rk4_step(fun,x[i],y[i],h)
```

Here is the plot of the resulting error from the true solution (of order 10^{-4} for h=0.1):



In order to improve this method, we will compare 1 step of length h to 2 steps of length h/2. Since RK4 uses a Taylor expansion up to the fourth order, the error terms will be $O(h^5)$. Using $RK4_h(x)$ to define the RK4 computed step with step size h:

$$y(x+h) = RK4_h(x) + O(h^5) = RK4_{h/2}(x) + 2O((h/2)^5) = RK4_{h/2}(x) + \frac{1}{16}O(h^5)$$

Where $O((h/2)^5)$ is multiplied by 2 in the h/2 RK4 stepper since 2 steps are needed for every 1 step of h.

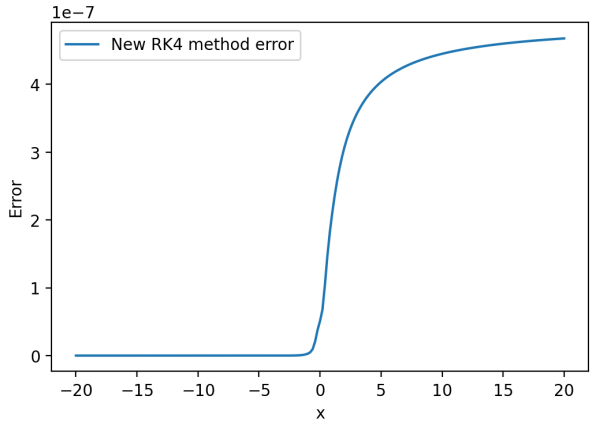
$$\Rightarrow 16\,RK4_{h/2}(x) - RK4_h(x) = 15y(x+h) + O(h^6)$$

$$\Rightarrow y(x+h) = \frac{16\,RK4_{h/2}(x) - RK4_h(x)}{15} + O(h^6)$$

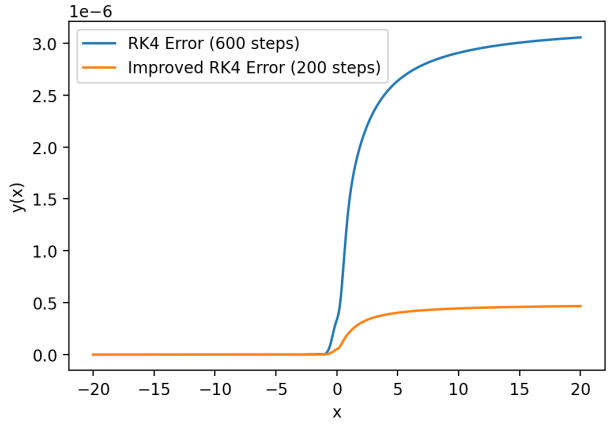
By combining step size h and h/2 in this ratio, we can obtain an error of $O(h^6)$. Writing the function `rk4_stepd(fun,x,y,h)`:

```
def rk4_stepd(fun,x,y,h):              #Start count of fun calls:
    y1=rk4_step(fun,x,y,h)             #+4
    y2=rk4_step(fun,x,y,h/2)           #+4
    y2=rk4_step(fun,x+h/2,y2,h/2)      #+4
    return (16*y2-y1)/15               #Total 12 fun calls
                                       # => use 1/3 the intervals for same call count
```

Iterating over the intervals and plotting results in:



As seen in the above code blocks, `rk4_step` calls `fun` 4 times for each step, `rk4_stepd` calls `fun` 12 times for each step. Therefore for the same number of function calls, the interval will be split into 1/3 the amount of sub-intervals for evaluating with `rk4_stepd` when compared to `rk4_step`. Using 600 sub-intervals for `rk4_step` and 200 for `rk4_stepd`, their errors were plotted:



Even when using 1/3 the amount of intervals to evaluate the stepper on, the improved RK4 method is still greater than 6 times more accurate than the original method.

2 Question 2

For this problem, a system of ODE's modelling the decay chain of Uranium-238 was solved using `scipy.integrate.solve_ivp(fun,t_span,y0)`. The system of ODE's in question is an example of the Bateman Equation: (`https://en.wikipedia.org/wiki/Bateman_equation`)

$$\frac{dN_1(t)}{dt} = -\lambda_1 N_1(t)$$

$$\frac{dN_i(t)}{dt} = \lambda_{i-1} N_{i-1}(t) - \lambda_1 N_i(t)$$

$$\frac{dN_k(t)}{dt} = \lambda_{k-1} N_{k-1}(t)$$

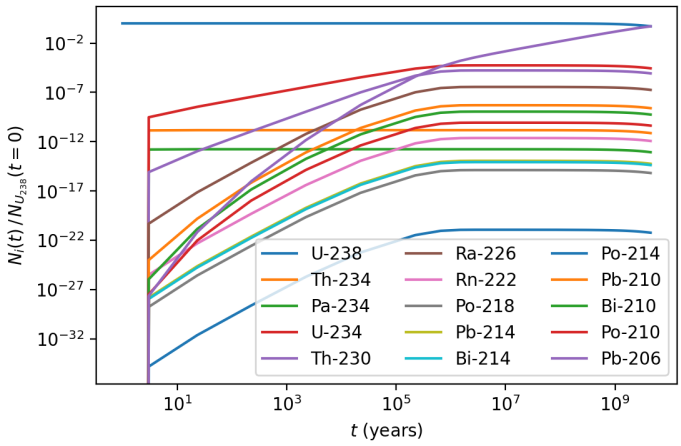
Where there are k isotopes and where $N_i(t)$ is the concentration of the i^{th} isotope with a decay rate λ_i. In this case, `y0` is the initial conditions vector and `fun` is the vector of functions on the RHS of each equation in the system. :

```
decay_rates=np.log(2)/half_lives
n=len(decay_rates)+1

def fun(t,y):
    dydt=np.zeros(n)
    dydt[0]=-decay_rates[0]*y[0]  #First isotope is only decaying
    for i in range(1,n-1):
        #Middle isotopes decay and are produced by previous isotopes
        dydt[i]=-decay_rates[i]*y[i]+decay_rates[i-1]*y[i-1]
    dydt[n-1]=decay_rates[n-2]*y[n-2]  #Last isotope is only being produced
    return dydt
```

Where `decay_rates` is defined using a list `half_lives` which are the half lives on the lecture slides, not including Pb-206 which is stable.

Here is a loglog plot of all the isotopes' quantities divided by the initial amount of U-238, over the time span of 1 U-238 half-life:



Next is the plot for the ratio of of Pb-206 to U-238. When the half lives of all intermediate isotopes are approximated to 0 in relation to the half life of U-238, the system of differential equations becomes:

$$\frac{dN_1(t)}{dt} = -\lambda_1 N_1(t)$$

$$\frac{dN_2(t)}{dt} = \lambda_1 N_1(t)$$

Which is easy to solve analytically:

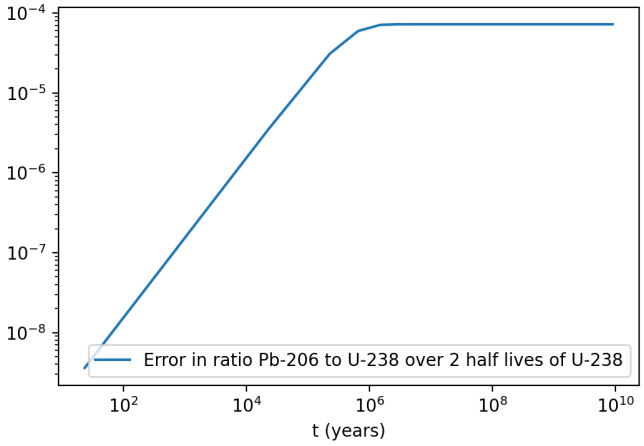
$$\frac{N_1(t)}{N_1(t_0)} = e^{-\lambda_1 t}$$

$$N_1(t)) + N_2(t) = N_1(t_0) \Rightarrow N_2(t) = N_1(t_0)(1 - e^{-\lambda_1 t})$$

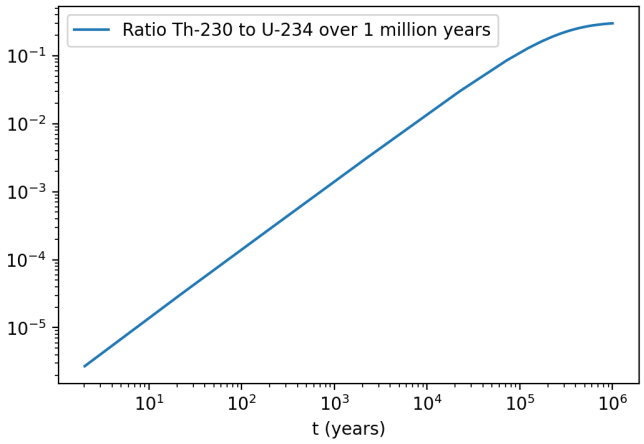
Therefore the ratio of Pb-206 to U-238 is:

$$\frac{N_2(t)}{N_1(t)} = \frac{1 - e^{-\lambda_1 t}}{e^{-\lambda_1 t}} = e^{\lambda_1 t} - 1$$

Comparing this to the numerically computed solution, the error is less than 10^{-4}:



As for the ratio of Th-230 to U-234:



3 Question 3

a)

$$z - z_0 = a((x - x_0)^2 + (y - y_0)^2)$$
$$= a(x^2 - 2x_0x + x_0^2 + y^2 - 2y_0y + y_0^2)$$
$$= a(x^2 + y^2) - 2ax_0x - 2ay_0y + a(x_0^2 + y_0^2)$$

This can be re-parameterized:

$$z = a(x^2 + y^2) + bx + cy + d$$

Where $b = -2ax_0$, $c = -2ay_0$, $d = a(x_0^2 + y_0^2) + z_0$. As a matrix equation:

$$z = \begin{bmatrix} x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ x_1^2 + y_1^2 & x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n^2 + y_n^2 & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \mathbf{A}m \tag{1}$$

We can recover x_0, y_0, z_0 and a from the fit parameters by:

$$a = a \rightarrow x_0 = \frac{b}{a},\ y_0 = \frac{c}{a} \rightarrow z_0 = d - a(x_0^2 + y_0^2)$$

To fit the data, first the matrix A was coded, then assuming the noise matrix is close to the identity, the Singular Value Decomposition (SVD) method was used to decompose the matrix A then rearannge the original matrix equation for A to get m. As derived in the lecture slides:

$$\mathbf{A}m = z\,,\ \mathbf{A} = \mathbf{USV}^T$$

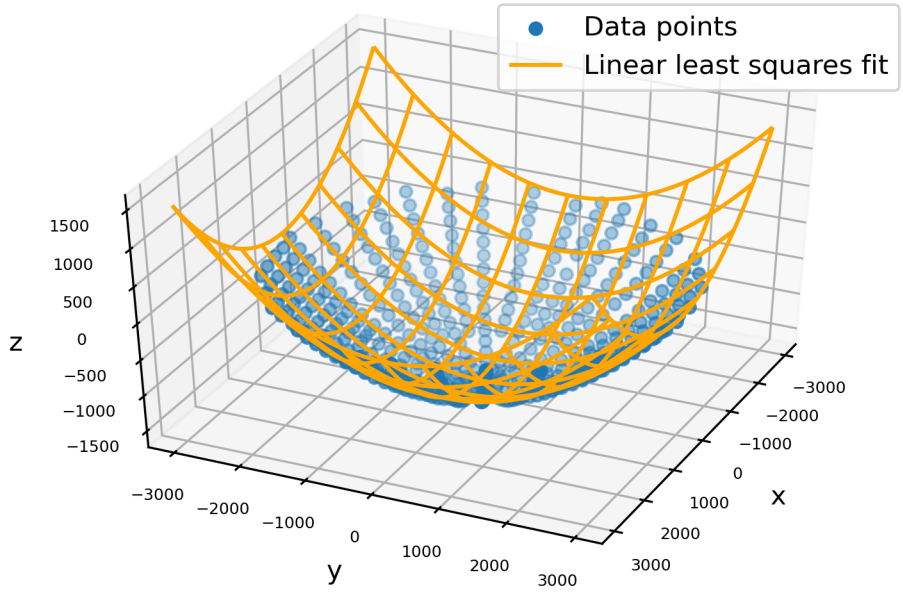
$$\Rightarrow m = \mathbf{VS}^{-1}\mathbf{U}^T z$$

```
A=np.empty([len(x_data),4])
A[:,0]=x_data**2+y_data**2    #Defining A matrix based on parametrization
A[:,1]=x_data
A[:,2]=y_data
A[:,3]=1

u,s,vt=np.linalg.svd(A,False)  #A=USV^T , Singular Value Decompositon
s_inv=np.diag(1/s)   #Since s is diagonal
m=vt.T@s_inv@u.T@z_data # m=VS^-1U^Tx  , model parameters using SVD
```

Plot of data and best fit surface on next page

The data points and the surface resulting from the best fit parameters were then plotted:



The best fit parameters were:

$$a = 0.00016670445477399507$$

$$x_0 = 2.7209772323619767$$

$$y_0 = -116.44295224646957$$

$$z_0 = -1514.5733901286528$$

For the error in the focal point, if $f(a)$ is the the focal length as a function of a and da is the error of a, then the error is:

$$f(a + da) = f(a) + f'(a)da + ...$$

$$\Rightarrow err(f) = f'(a)da = \frac{1}{4a^2}da$$

To get the error in a, the matrix formula for the error in the model parameters was used:

$$(m - m_t) = (\mathbf{A}^T\mathbf{N}^{-1}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{N}^{-1}n$$

Finally, this was coded to return the error of a then calculate the error of f:

```
#Determination of error in focal point
n=z_data-A@m  #Estimation of noise
N=np.outer(n,n)
N_inv=np.linalg.inv(N)
m_err=np.linalg.inv(A.T@N_inv@A)@A.T@N_inv@n  # m-m_t=(A^TN^-1A)^-1A^TN^-1n
m_err=np.sqrt(np.abs(m_err))

f=1/(4*m[0])   #f=1/(4a)
f_err=1/(4*m[0]**2)*m_err[0]   #f(a+err)=f(a)+f'(a)*err
print('Focal length =',f,'+/-',f_err)
```

This returned:

$$f = 1499.6600 \pm 0.0004$$