Assignment 3

Q1.

rk4_Step can be copied from class repo

for rk4_stepd, first recall that rk4 is accurate to 4^{th} order, meaning that the error goes like dx^4. when we use step size of h, we can write

$$f(x+h)_h = f(x+h)_{true} + a h^4 + \mathcal{O}(h^5)$$

when we use two step with step size of $\frac{h}{2}$, we can write

$$f(x+h)_{\frac{h}{2}} = f(x+h)_{true} + a\left(\frac{h}{2}\right)^4 + \mathcal{O}(h^5)$$

manipulate to get

$$\frac{f(x+h)_{\frac{h}{2}} \cdot 16 - f(x+h)_h}{15} = f(x+h)_{true} + \mathcal{O}(h^5)$$

so we see that we can cancel leading error by using above equation.

The disadvantage is that we called the function 2 more times per step when we calculate $f(x+h)\frac{h}{2}$. which means the number of function calls for rk4.stepd is 3 times that for rk4.step

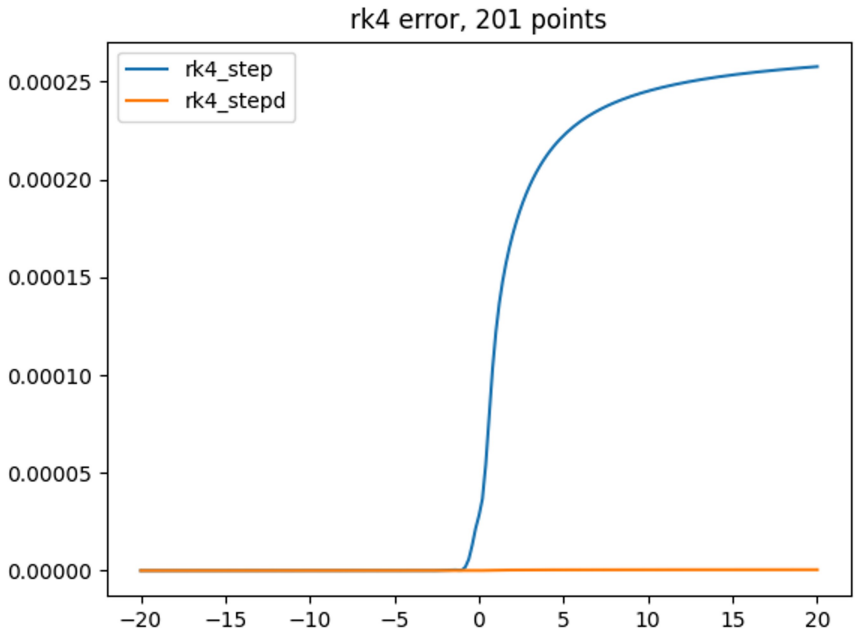
```
import numpy as np
from matplotlib import pyplot as plt

#define dydx = y/(1+x^2)
def myfun(x,y):
    return y/(1+x**2)

#copied from rk4.py in class repo
#equations can be found in class ppt page 5
def rk4_step(fun,x,y,h):
    k1=fun(x,y)*h
    k2=h*fun(x+h/2,y+k1/2)
    k3=h*fun(x+h/2,y+k2/2)
    k4=h*fun(x+h,y+k3)
    dy=(k1+2*k2+2*k3+k4)/6
    return y+dy

def rk4_stepd(fun,x,y,h):
    rk4_h = rk4_step(fun,x,y,h)
    #run rk4 twice with step h/2
    #note we called the function 3 times per step
    rk4_h2 = rk4_step(fun,x+h/2,rk4_step(fun,x,y,h/2),h/2)
    #from our previous calculation
    return (rk4_h2*16-rk4_h)/15

npt=201 #use 200 steps
```

```
x0=-20
x1=20
y0=1
x=np.linspace(x0,x1,npt)
y1=np.zeros(npt) #y1 uses rk4_step
y2=np.zeros(npt) #y2 uses rk4_stepd
y1[0]=y0 #starting conditions
y2[0]=y0

for i in range(npt-1):
    h=x[i+1]-x[i]
    y1[i+1]=rk4_step(myfun,x[i],y1[i],h)
    y2[i+1]=rk4_stepd(myfun,x[i],y2[i],h)

#calculate the constant term in real solution
c0 = 1/(np.exp(np.arctan(-20)))
#real solution
truth=c0*np.exp(np.arctan(x))
print(np.std(truth-y1))
print(np.std(truth-y2))

plt.clf()
# plt.plot(x,np.abs(y1-truth))
plt.plot(x,np.abs(y2-truth))
# plt.legend(['rk4_step','rk4_stepd'])
plt.title('rk4_stepd error, ' + repr(npt)+ ' points')
plt.show()
plt.savefig('rk4_err_stepd.png')
```

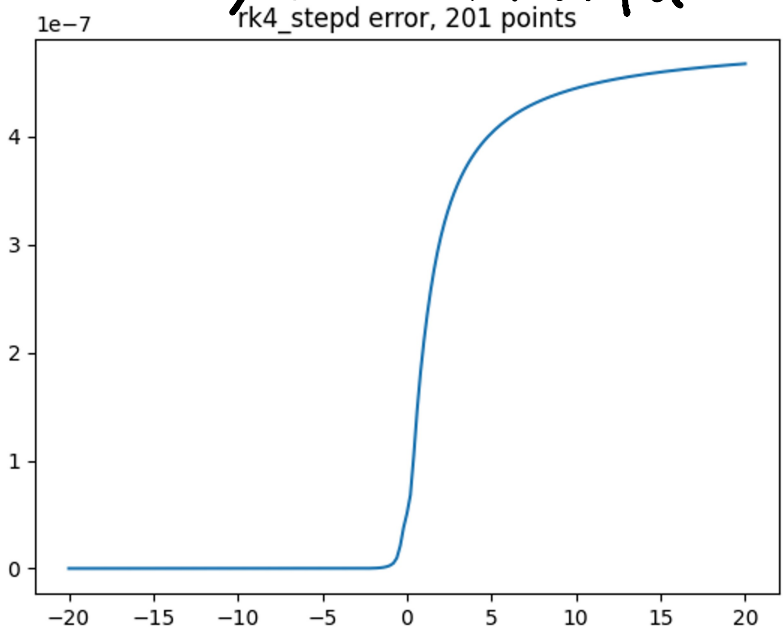
comment & change different part of this to get different graphs

The output is

Error for rk4_step is: 0.00011776140522522912
Error for rk4_stepd is: 2.1342639258687148e-07



only show rk4-stepd



we can see that rk4.stepd indeed has a much lower error than rk4.step, as to be expected because

we eliminated leading error.

Q2.

(a) As we discussed in class, the decay problem is a stiff equations. Therefore it needs to be solved implicitly.

I will use scipy.integrate.solve_ivp with Radau method.

```
import numpy as np
from scipy import integrate
from matplotlib import pyplot as plt

#get time translation coeffs
day_to_year = 1/365.25 #radioactivity uses Julian year
hour_to_day = 1/24
min_to_hour = 1/60
sec_to_min = 1/60
hour_to_year = hour_to_day * day_to_year
min_to_year = min_to_hour * hour_to_year
sec_to_year = sec_to_min * min_to_year

#create life time of years
lifetime=np.zeros(14)
lifetime[0] = 4.468e+9
lifetime[1] = 24.10*day_to_year
lifetime[2] = 6.70*hour_to_year
lifetime[3] = 245500
lifetime[4] = 75380
lifetime[5] = 1600
lifetime[6] = 3.8235*day_to_year
lifetime[7] = 3.10*min_to_year
```

```
lifetime[8] = 26.8*min_to_year
lifetime[9] = 19.9*min_to_year
lifetime[10] = (164.3e-6)*sec_to_year
lifetime[11] = 22.3
lifetime[12] = 5.015
lifetime[13] = 138.376*day_to_year

def fun(x,y):
    #set up the ODE system
    global lifetime
    dydx=np.zeros(len(lifetime)+1)
    #U238 decay
    dydx[0]=-y[0]/lifetime[0]
    #decay into middle decay products, then themselves decay
    for i in np.arange(1,len(lifetime)):
        dydx[i]=y[i-1]/lifetime[i-1]-y[i]/lifetime[i]
    #decay into Pb206
    dydx[14]=y[13]/lifetime[13]
    return dydx

#use integrate.solve_ivp with Radau method to solve the ODE system implicitly
def solve_decay(x0,x1):
    y0 = np.zeros(15)
    y0[0] = 1 #start with pure U238
    ans_stiff=integrate.solve_ivp(fun,[x0,x1],y0,method='Radau')
    return ans_stiff
```

(b)

For Pb206/U238, if we use the hint to assume that U238 decays instantly into Pb206, then we have

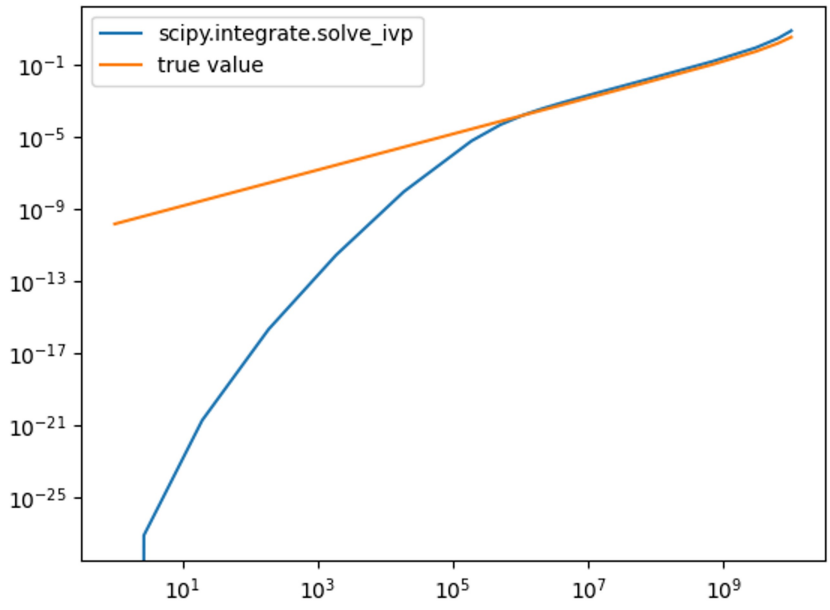
$$N_{U238} = \left(\frac{1}{2}\right)^{\frac{t}{t_{\frac{1}{2}}}}$$

$$N_{Pb206} = 1 - N_{U238}$$

plot it against what we have with time from 0 to 10^{10} year

```
ans_utopb = solve_decay(1,1e+10)
```

```
t = ans_utopb.t

y_U238 = ans_utopb.y[0,:]
y_Pb206 = ans_utopb.y[-1,:]
y_U234 = ans_utopb.y[3,:]
y_Th230 = ans_utopb.y[4,:]

#true value if U238 decays instantly into Pb206
y_U238_true = 0.5**(t/lifetime[0])
y_Pb206_true = 1-y_U238_true

plt.clf()
plt.loglog(t,y_Pb206/y_U238)
plt.plot(t,y_Pb206_true/y_U238_true)
plt.legend(['scipy.integrate.solve_ivp','true value'])
plt.savefig('U238_to_Pb206.png')
plt.show()

plt.clf()
plt.loglog(t,y_U234/y_Th230)
plt.savefig('U234_to_Th230.png')
plt.show()
```

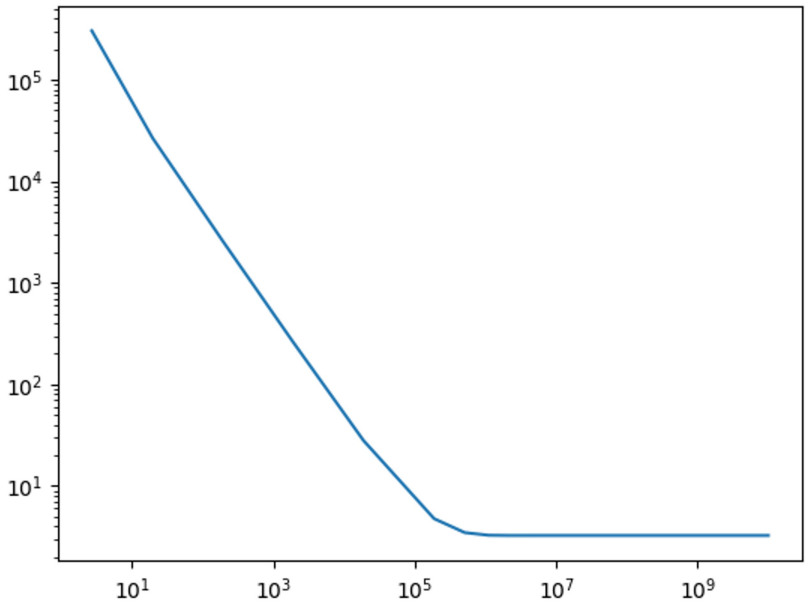


At early time they do not match.

But this is to be expected, as

But this is to be expected, as our expected true value is NOT accurate at low t, when the half-life of intermediate products are not neglectable. We see that our expected value and scipy agrees pretty well at large t. So we can say that our answer makes sense analytically.

For U234/Th230 ratio we have



Q3

(a) To make it linear:

$$z - z_0 = a\left((x-x_0)^2 + (y-y_0)^2\right)$$

$$z - z_0 = a\left(x^2 - 2x_0x + x_0^2 + y^2 - 2y_0y + y_0^2\right)$$

$$z = ax^2 - 2ax_0x + ay^2 - 2ay_0y + \left(z_0 + ax_0^2 + ay_0^2\right)$$

$$z = a(x^2+y^2) - 2ax_0x - 2ay_0y + (z_0 + ax_0^2 + ay_0^2)\cdot 1$$

$$\equiv \alpha(x^2+y^2) + \beta x + \gamma y + \delta$$

where $a = \alpha$

$$x_0 = \frac{-\beta}{2\alpha}$$

$$y_0 = \frac{-\gamma}{2\alpha}$$

$$z_0 = \delta - \frac{\beta^2}{4\alpha} - \frac{\gamma^2}{4\alpha}$$

written in matrix form,

$$A = \begin{pmatrix} (x+y)_0^2 & x_0 & y_0 & 1 \\ (x+y)_1^2 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$$M = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix}$$

(b)

```
import numpy as np
from matplotlib import pyplot as plt

#load data from txt file
dat = np.loadtxt('dish_zenith.txt')
x = dat[:,0]
y = dat[:,1]
z = dat[:,2]

ndata = len(x)

#set up our A matrix
A=np.empty([ndata,4])
A[:,0] = ((x**2)+(y**2))
A[:,1] = x
A[:,2] = y
A[:,3] = 1

#same as in class. Ignore noise for now.
#m = (A^T*A)^{-1}*(A^T*z)
lhs=A.T@A
rhs=A.T@z
m=np.linalg.inv(lhs)@rhs
pred=A@m

#estimate parameter error
#assume noise is naively the mean between real z and predicted z
#variance = (A^T*N^{-1}*A)^{-1}
N=np.mean((z-pred)**2)
par_errs=np.sqrt(N*np.diag(np.linalg.inv(lhs)))
a = m[0]
```

```
x0 = -m[1]/(2*a)
y0 = -m[2]/(2*a)
z0 = m[3]-a*(x0**2)-a*(y0**2)
param = [a,x0,y0,z0]

print('Predicted parameters of the new model are: ', m)
print('Error bars are: ', par_errs)
print('Predicted parameters of the old model are: ', param)
```

Output is:

Predicted parameters of the new model are: [1.66704455e-04
4.53599028e-04 -1.94115589e-02 -1.51231182e+03]
Error bars are: [6.45189976e-08 1.25061100e-04 1.19249564e-04 3.12018436e-01]
Predicted parameters of the old model are: [0.00016670445477401358, -1.360488622197728,
58.22147608157934, -1512.8772100367878]

$\Rightarrow a \simeq 1.667 \times 10^{-4}$

$x_0 \simeq -1.36$

$y_0 \simeq 58.22$

$z_0 \simeq -1512.88$

(c) The error estimate for the new model is done in part (b) above.

We see that the uncertainty of a is 6.45×10^{-8}

Recall that the focal length of a symmetrical paraboloidal dish is

is

$4FD = R^2$.

In our case $D = z - z_0$

$$R^2 = (x-x_0)^2 + (y-y_0)^2$$

we have

$$z - z_0 = \frac{1}{4F}\left[(x-x_0)^2 + (y-y_0)^2\right]$$

but we also have

$$z - z_0 = a\left[(x-x_0)^2 + (y-y_0)^2\right]$$

$\Rightarrow F = \frac{1}{4a} \cong 1499.66$ mm

is very close to what we want with 1.5 m.

To calculate the error bar of F,

$F = \frac{1}{4a}$

Recall from error propagation or simply from

https://en.wikipedia.org/wiki/Propagation_of_uncertainty

we have

$$\sigma_F \simeq \left|\frac{1}{4}\right| \cdot |F| \cdot \left|\frac{\sigma_a}{A}\right|$$

$$\Rightarrow \sigma_F \cong 0.1451$$