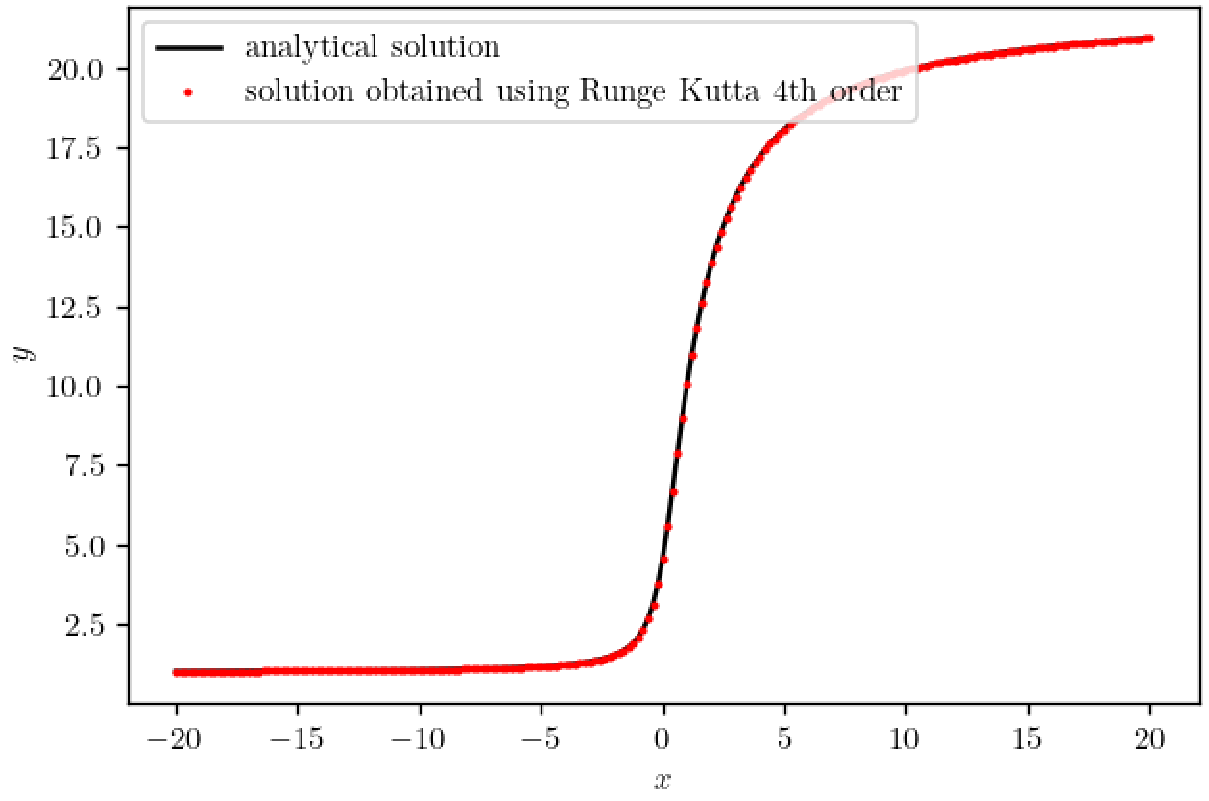
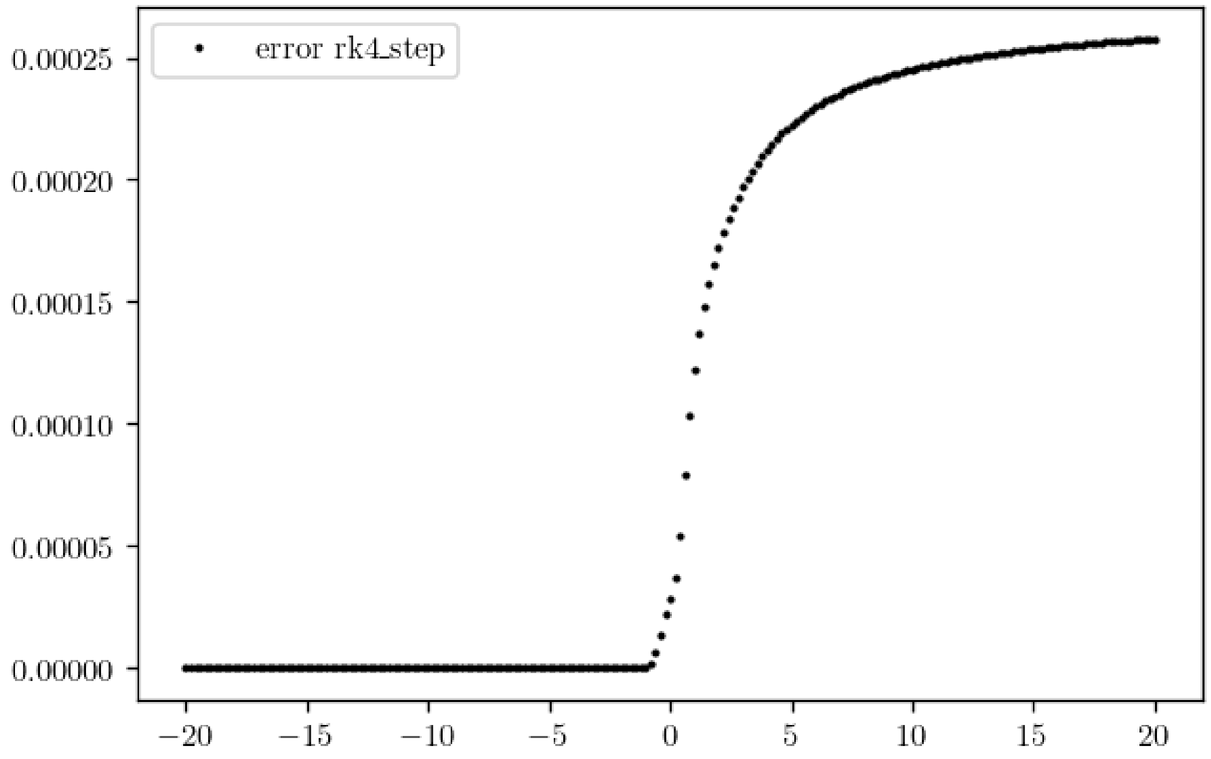
Question 1

In [ ]:
```
# RK4 stepper
import numpy as np
import matplotlib.pyplot as plt
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rcParams['figure.dpi'] = 120

# solving dy/dx = y/(1+x**2) using RK4

# ODE to solve
def my_fun(x,y):
    dydx=y/(1+x**2)
    return dydx

# analytical solution
def truth(x):
    return (np.exp(np.arctan(x)))/(np.exp(np.arctan(-20)))

# Runge Kutta 4th order stepper
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

# advancing the solution using rk4_step
y0=1
x=np.linspace(-20,20,201)
h=np.median(np.diff(x))
y=np.zeros(len(x))
y[0]=y0
steps = 0
for i in range(len(x)-1):
    y[i+1]=rk4_step(my_fun,x[i],y[i],h)
    steps += 1

# error
err_step = np.abs(y-truth(x))
rmse_step = np.sqrt(np.mean(err_step**2))
print('Number of steps is', steps)
plt.plot(x, truth(x), 'k', label = 'analytical solution')
plt.plot(x,y, 'r.', markersize=3, label = 'solution obtained using Runge
Kutta 4th order')
plt.legend(loc = 'upper left')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()

plt.plot(x, err_step, 'k.',markersize=3, label = 'error rk4_step')
plt.legend()
plt.show()
print('RK4 stepper root mean square error is', rmse_step)
```

```
Number of steps is 200
```





```
RK4 stepper root mean square error is 0.0001637665315667728
```

Now let's do two steps of $h/2$ and one step of h and compare them to cancel out the leading order error term (which is h^5). The two numerical approximations are related by:

$$f(x+(h)) = y = y_1 + (h)^5 + O(h^6)$$
$$f(x+2(h/2)) = y = y_2 + 2(h/2)^5 + O(h^6).$$

The first equation is the error for the big step of $2h$ and the second equation is the error for two small stepts of h. If we drop the $O(h^6)$, we can try to solve for a combination of y_1 and y_2 that will cancel the leading order error in h^5.

We have,

$$y_2 - \frac{y_1}{16} = y - \frac{h^5}{16} - (\frac{y}{16} - \frac{h^5}{16}) = \frac{15y}{16}.$$

Now rearanging to find y in terms of y_1 and y_2 we have that

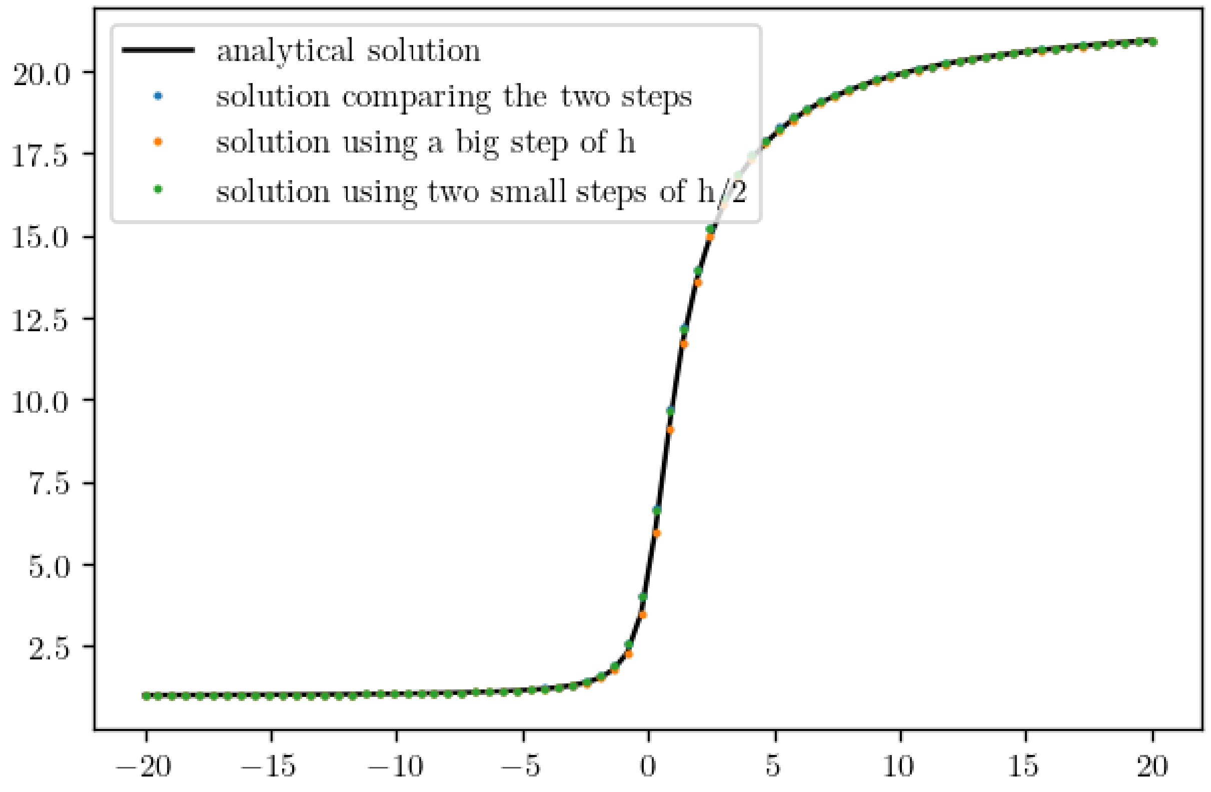
$$y = \frac{16y_2 - y_1}{15}. \quad (1)$$

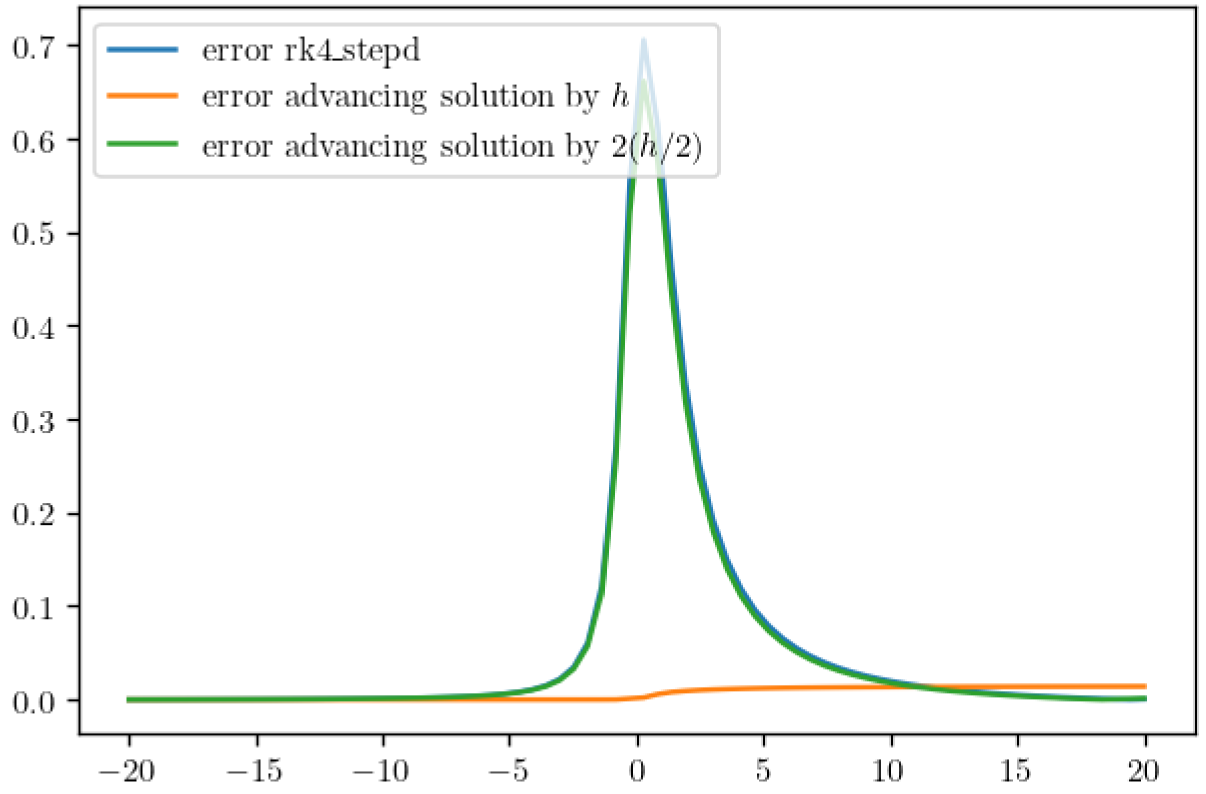
We can use Equation (1) to get a better numerical integration of our ODEs using RK4.

The normal RK4 stepper uses 4 function evaluations per step, so 800 evaluations for 200 steps. This new adaptive stepper takes 4 functions evaluations for the 3 calls, every step, but they all share the same starting point, so 11 evaluations per step. For fairness, we want to use the same amount of function evaluations to compare both methods, so with the adaptive method we need 73 steps to reach 800 function evaluations.

In [ ]:
```
# adaptive RK4 stepper
y0=1
x=np.linspace(-20,20,74)
h=40/73
yd=np.zeros(len(x))
y1=np.zeros(len(x))
y2=np.zeros(len(x))
yd[0]=y0
y1[0]=y0
y2[0]=y0

steps = 0
for i in range(len(x)-1):
    y2[i+1] = rk4_step(my_fun,x[i],y2[i],h/2) # two smaller steps of RK4
    y2[i+1] = rk4_step(my_fun,x[i+1],y2[i+1],h/2)
    y1[i+1] = rk4_step(my_fun,x[i], y1[i], h) # normal step of RK4

    yd[i+1] = (16*y2[i+1]-y1[i+1])/15
    steps += 1

err_stepd = np.abs(yd-truth(x))
rmse_stepd = np.sqrt(np.mean(err_stepd**2))
err_y2 = np.abs(y2-truth(x))
rmse_y2 = np.sqrt(np.mean(err_y2**2))
err_y1 = np.abs(y1-truth(x))
rmse_y1 = np.sqrt(np.mean(err_y1**2))



print('The number of is', steps)
plt.plot(x, truth(x), 'k', label = 'analytical solution')
plt.plot(x,yd,'.', markersize=3, label ='solution comparing the two
steps')
plt.plot(x,y1, '.', markersize=3, label = 'solution using a big step of
h')
plt.plot(x,y2, '.', markersize=3, label = 'solution using two small steps
of h/2')
plt.legend()
plt.show()
plt.plot(x, err_stepd, '-',markersize=3, label = 'error rk4_stepd')
plt.plot(x, err_y1, '-',markersize=3, label = 'error advancing solution by
$h$')
plt.plot(x, err_y2, '-',markersize=3, label = 'error advancing solution by
$2(h/2)$')
plt.legend(loc = 'upper left')
plt.show()
print('The rms error for comparing the two steps is:', rmse_stepd)
print('The rms error for using a big step of h is:', rmse_y1)
print('the rms error for using two small steps of h/2', rmse_y2)
```

```
The number of is 73
```





```
The rms error for comparing the two steps is: 0.15503707680896503
The rms error for using a big step of h is: 0.008972594144552778
the rms error for using two small steps of h/2 0.1451947612230162
```

This is much worse, for the same number of function evaluations, than just using a step of length h. I have a feeling that this should not be the case, and there is something wrong with my code. I think that something goes wrong when I take the two steps of h/2, because a solution using these small steps has a rmse of 0.14, while a solution using only big steps of h has a rmse of 0.009. I have not been able to resolve that.

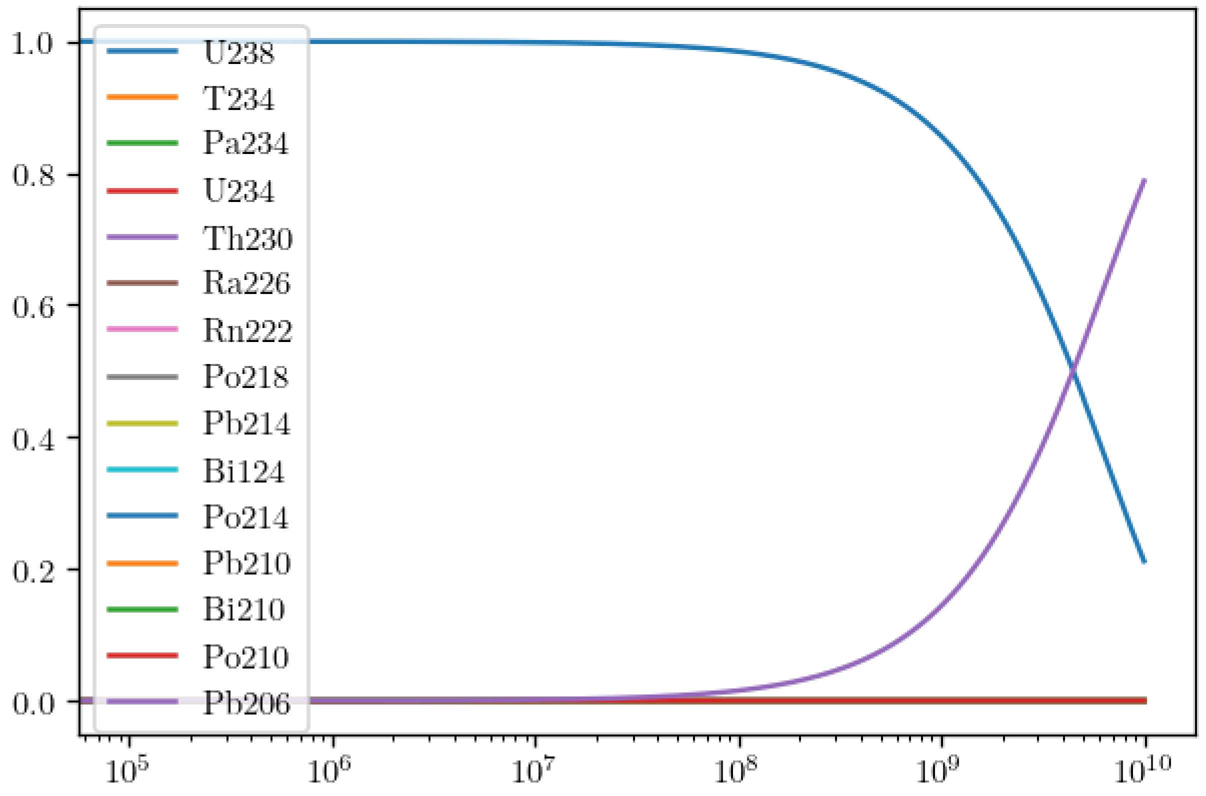
I've also checked for the same number of steps (200) and the adaptive method of comparing the step size still gives an error that is worse than just using a big step of h for some reason.

Question 2

In [ ]:
```
# Question 2 a)

import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rcParams['figure.dpi'] = 120
```

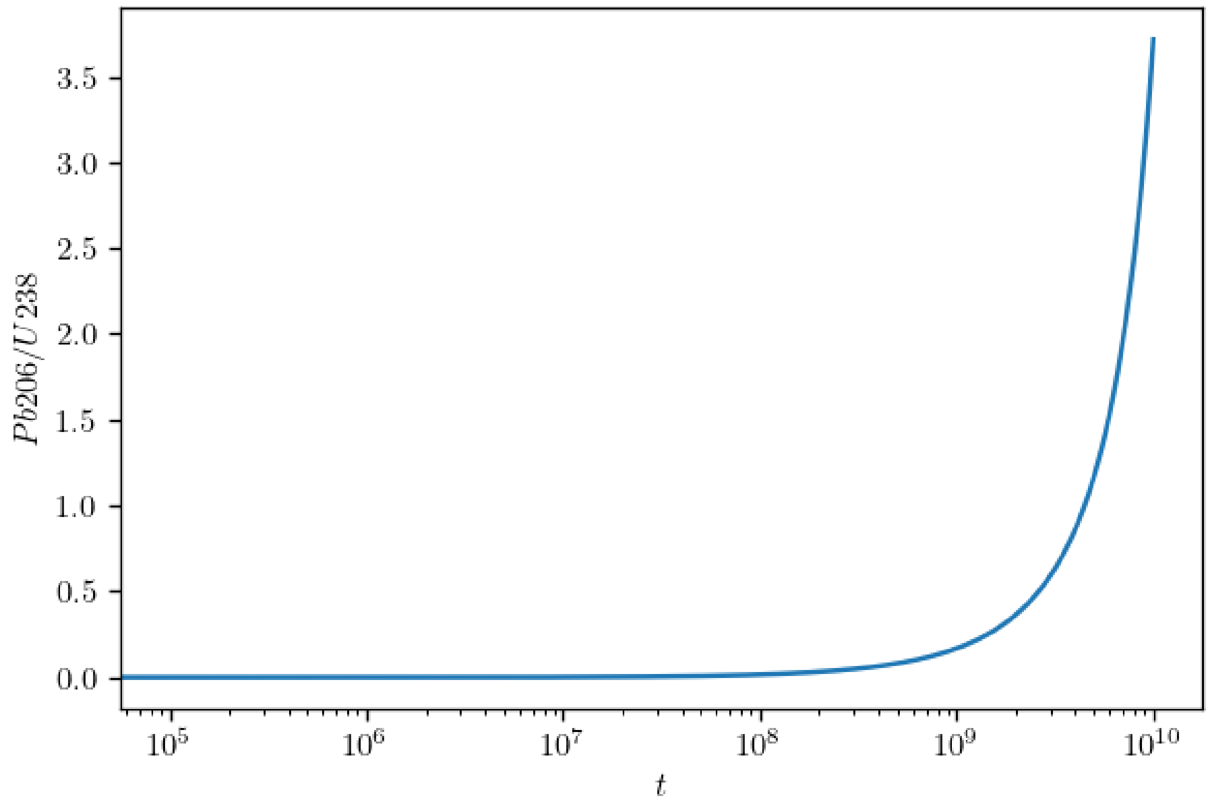
```
# elements of the decay chain of U238 (except the final, stable elemtent)
elements = ['U238', 'T234', 'Pa234', 'U234', 'Th230', 'Ra226', 'Rn222',
'Po218', 'Pb214', 'Bi124', 'Po214', 'Pb210', 'Bi210', 'Po210']
# half life, in years, of the elements of the decay chain of U238
half_life = [4.468e9,0.06598,0.0007643,
245500,75380,1600,0.010468,0.00000589,0.00005095,0.00003784,5.206e-
12,22.3,5.015,0.37885]

# the system of ODEs associated with nuclear decay for k steps is
# dN_1(t)/dt = -ln(2)/t_1,1/2 N_1(t)
# dN_i(t)/dt = -ln(2)/t_i,1/2 N_i(t) + ln(2)/t_i-1,1/2 N_i-1(t)
# dN_k(t) = ln(2)/t_k-1,1/2 N_k-1(t)
# where N is the quantity of element and t_1/2 is the half life

def decay_ODEs(t, N, k):

    dNdt = np.zeros(len(k)+1)  # change of quantity for all elements
including the last one (which is not in the dict)

    dNdt[0] = -N[0]*np.log(2)/k[0] # first element (U238) decay

    for i in range(0,len(k)-1):
        dNdt[i+1]=(N[i]/k[i]-N[i+1]/k[i+1])*np.log(2) # intermediate
elements of the chain



    dNdt[-1]=N[len(k)-1]/k[len(k)-1]*np.log(2) # last element of chain:
stable so does not decay

    return dNdt

t = np.linspace(0, 1e10, 100000)
N0 = np.zeros(len(half_life) + 1)
N0[0] = 1 # start with pure U238

# use solve_ivp from scipy to take care of this. Radau method because we
have stiff equations
sol = integrate.solve_ivp(fun= lambda t,y: decay_ODEs(t, y, half_life),
t_span=[t[0], t[-1]],t_eval = t,y0 = N0, method='Radau')
```

```
elements.append('Pb206')

for i in range(np.shape(sol.y)[0]):
    plt.semilogx(sol.t, sol.y[i], label = elements[i])
plt.legend()
plt.show()
```



I had to use the Radau method which is implicit and is much better to handle stiff equations (like we have here). We can see that all intermediate elements of the decay chain have proportions that remain really close to 0 at all times. This is because they all have short half life compared to U238 and Pb206 (which is stable). We can really just approximate the whole chain as decay of U238 directly to Pb206.

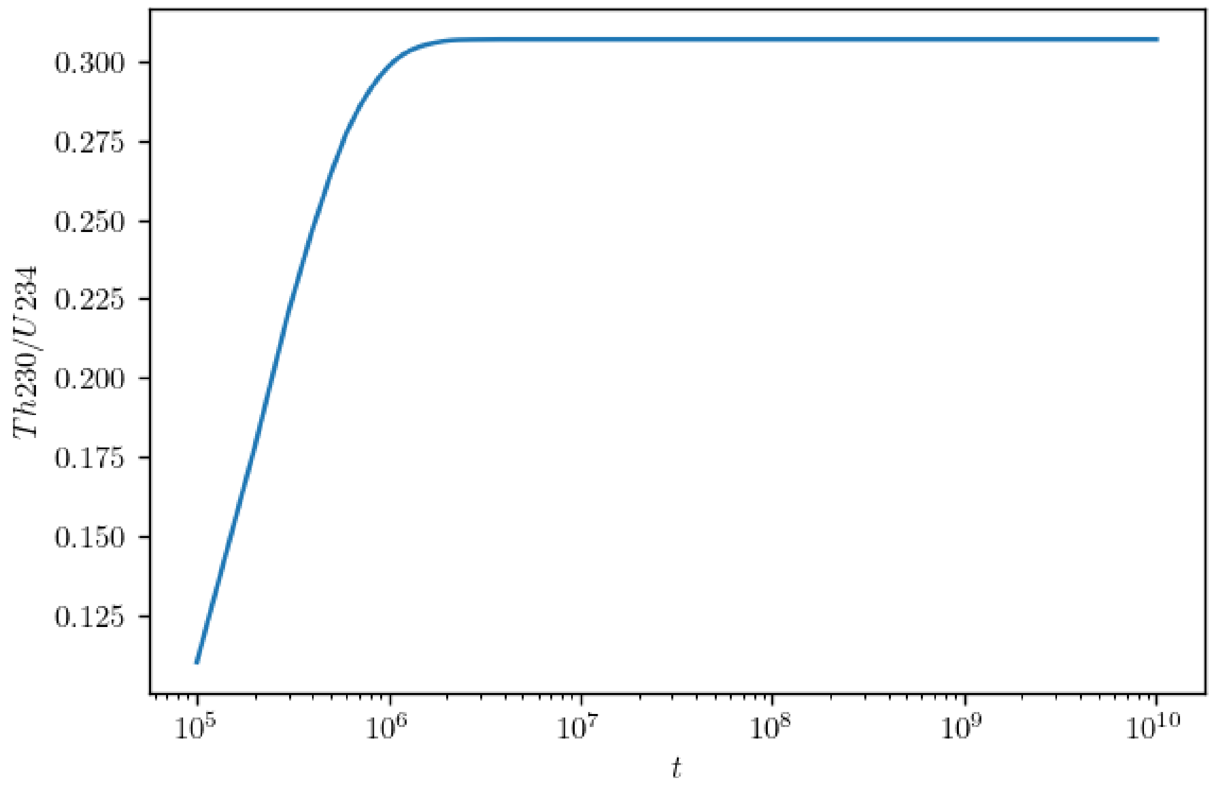
In [ ]:
```
# Question 2 b)

# ratio of Pb206 to U238
plt.semilogx(sol.t, sol.y[-1]/sol.y[0])
plt.xlabel('$t$')
plt.ylabel('$Pb206/U238$')
plt.show()
```



This makes sense. We start with a ratio of Pb206:U238 of 0:1, and towards the end of the process, we should have a ratio of 1:0, which goes to infinity exponentially as the decay rates are exponential.

```
In [ ]: # ratio of Th230 to Th234
        plt.semilogx(sol.t, sol.y[4]/sol.y[3], '-')
        plt.xlabel('$t$')
        plt.ylabel('$Th230/U234$')
        plt.show()
```

```
C:\Users\Greg\AppData\Local\Temp\ipykernel_2604\2558304246.py:2: RuntimeWarning: inva
lid value encountered in true_divide
  plt.semilogx(sol.t, sol.y[4]/sol.y[3], '-')
```



This also makes sense. The amount of Th230 grows exponentially until its growth rate matches the growth rate of U234. Then, they will have similar decay rate, which is why we see the ratio being constant at around 0.3. I assume that the growth/decay rate of these two elements can match up because they have similar half life (245 500 years for U234 and 75 380 years for Th230).

Question 3

a) The equation for a rotationally symmetric paraboloid is

$$z - z_0 = a((x - x_0)^2 + (y - y_0)^2).$$

Let's expand this:

$$z = a(x^2 + y^2) - 2axx_0 - 2ayy_0 + a(x_0^2 + y_0^2).$$

To do some linear least-squares fit we want $z =$ (linear combinations of powers of x and y), where the constants in the linear combinations are linear. That's not quite what we have (we have some $x_0^2, some$ax_0$, etc.), solet'spicksomenewconstants$b,c,d,e $suchthatz = b(x^2 + y^2) + cx + dy + e. Thiswillallowustodetermineb,c,d,e using least-squares, because they are linear. We can also directly read off what these constants should be in terms of the old parameters:

$$\begin{aligned} b &= a, \\ c &= -2ax_0, \\ d &= -2ay_0, \\ e &= a(x_0^2 + y_0^2) + z_0, \end{aligned}$$

and so,

$$\begin{aligned} a &= b, \\ x_0 &= -c/2b, \\ y_0 &= -d/2b, \\ z_0 &= e - \frac{1}{4b}(c^2 + d^2). \end{aligned}$$

In [ ]:

```
# Question 3 b)

import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rcParams['figure.dpi'] = 120

data = np.loadtxt('dish_zenith.txt')
x_data = data[:,0]; y_data = data[:,1]; z_data = data[:,2]

# set up array of values
A = np.empty([len(x_data), 4])

# first column should be x**2+y**2
A[:, 0] = x_data**2 + y_data**2
# second column is just x
A[:,1] = x_data
# third column is y
A[:,2] = y_data
# fourth column is constant
A[:,3] = 1

# use svd decomposition
u,s,v=np.linalg.svd(A,0)
mfit=v.T@np.diag(1/s)@u.T@z_data
pred = A@mfit
```

```
b,c,d,e = mfit

# old parameters
a = b; x0 = -c/(2*b); y0 = -d/(2*b); z0 = e -(c**2+d**2)/(4*b)

print('The best fit parameters are:')
print('a = ', a, 'mm')
print('x0 = ', x0, 'mm')
print('y0 = ', y0, 'mm')
print('z0 = ', z0, 'mm')
```

```
The best fit parameters are:
a =  0.00016670445477399477 mm
x0 =  -1.3604886161321565 mm
y0 =  58.22147612332249 mm
z0 =  -1512.8772100375948 mm
```

In [ ]:

```
# Question 3 c)

# let's estimate the noise
N = np.mean((z_data-pred)**2)

# from our estimate in the noise, we can find the error on the parameters
and the model
parameter_error = np.sqrt(N*np.diag(np.linalg.inv(A.T@A)))

B = A@np.linalg.inv(A.T@A)@A.T
model_error=np.sqrt(N*np.diag(B))

print('The uncertainty on "a" is:',parameter_error[0])
print('The uncertainty on "x0" is:',parameter_error[1])
print('The uncertainty on "y0" is:',parameter_error[2])
print('The uncertainty on "z0" is:',parameter_error[3])
# focal length is 1/(4a)
f = 1/(4*a)

# using uncertainty propagation
f_err = f*(parameter_error[0]/a )
```

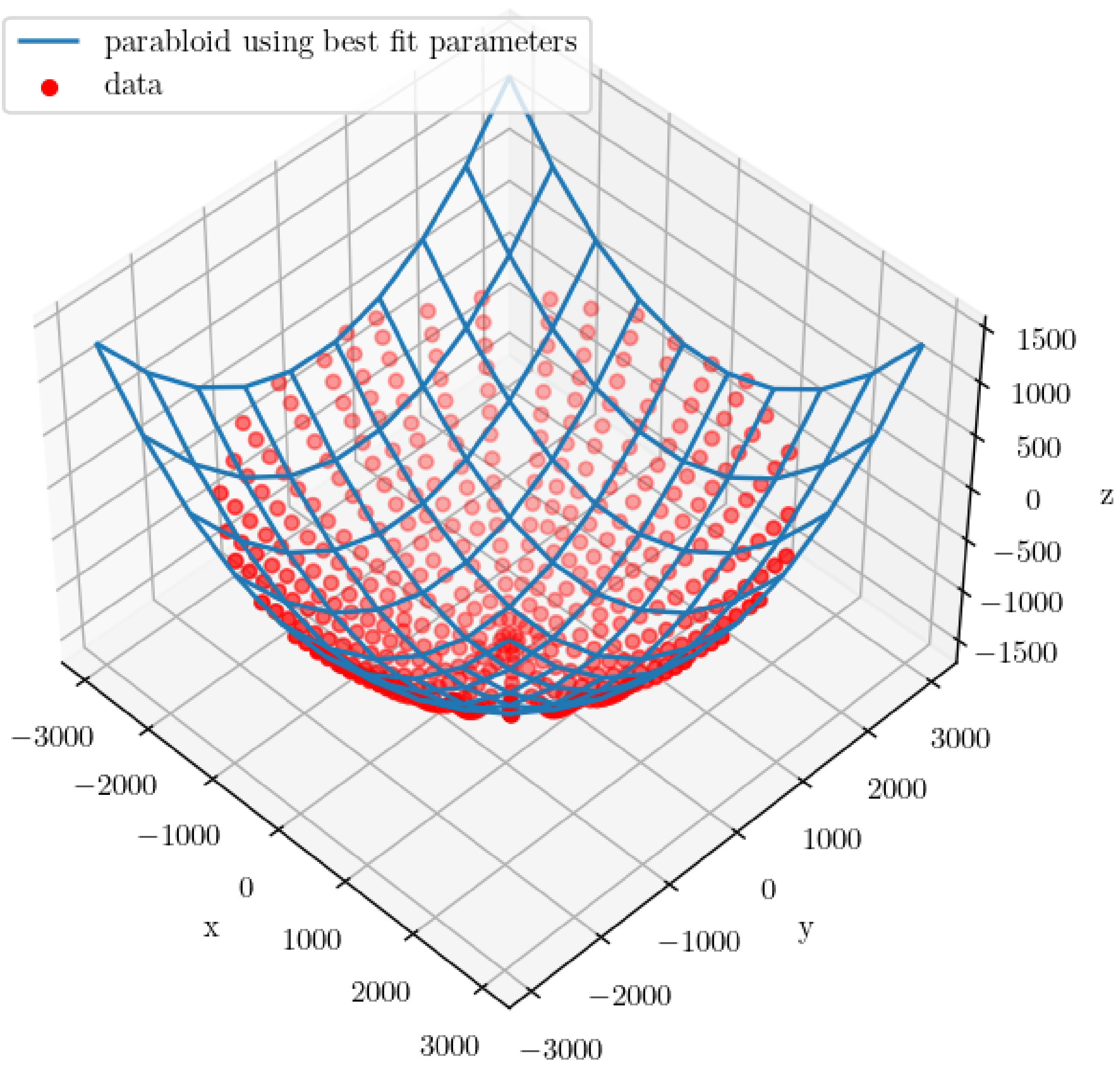
```
print('The focal length measured using our best fit paremeters is:',f,
',with uncertainty of:', f_err)
```

```
The uncertainty on "a" is: 6.451899757263453e-08
The uncertainty on "x0" is: 0.0001250610995127078
The uncertainty on "y0" is: 0.00011924956427610117
The uncertainty on "z0" is: 0.3120184362020165
The focal length measured using our best fit paremeters is: 1499.6599841253853 ,with
uncertainty of: 0.5804077581894141
```

True result is 1500 mm so we are quite close (actually the value we got with our fit parameters is equal to the real value considering uncertainties). Now let's plot the paraobloid using our fit parameters to compare it to the real data.

In [ ]:

```
# plotting stuff

# equation of paraboloid is
def paraboloid(x,y):
    return a*((x-x0)**2+(y-y0)**2)+z0

xx, yy = np.meshgrid(np.linspace(np.min(x_data), np.max(x_data),10),
np.linspace(np.min(y_data),np.max(y_data), 10))
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(xx, yy, paraboloid(xx,yy), label = 'parabloid using best
fit parameters')
ax.scatter(x_data, y_data, z_data, color='r', label = 'data')
ax.view_init(45, -45)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")
ax.legend(loc= 'upper left')
plt.show()
```



Doesn't look too bad!