In [348]:
```
from IPython.display import Latex
import numpy as np
from matplotlib import pyplot as plt
```

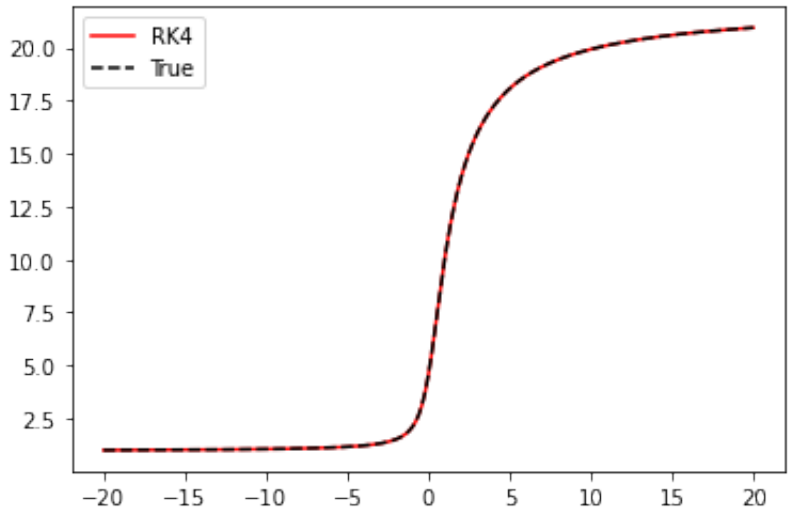
Problem 1

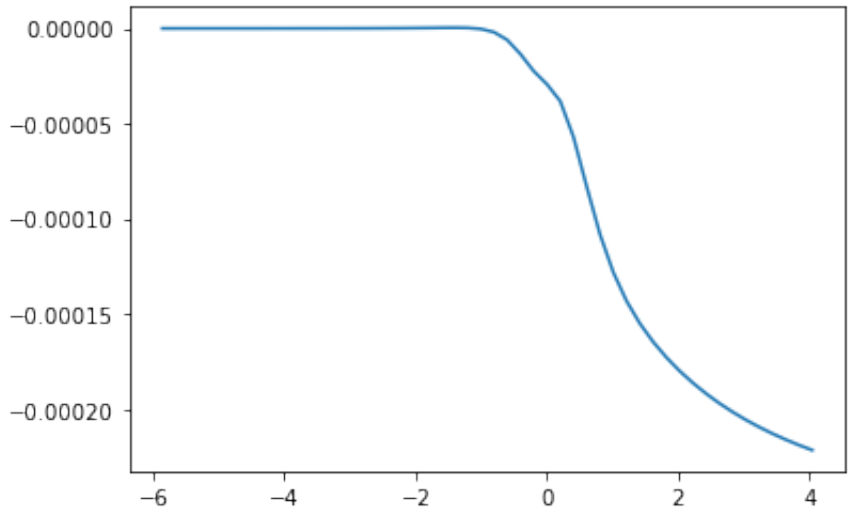
Part (a)

Integrate the initial value problem with a RK4 integrator.

In [349]:
```
def f(x,y):
    f.counter += 1
    dydx = y / (1 + x**2)
    return dydx
f.counter = 0


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 dy

npt = 199
x_range = np.linspace(-20, 20, npt)
yy=np.zeros([2,npt])

yy[0,0]=1 #starting conditions
yy[1,0]=0
for i in range(npt-1):
    h=x_range[i+1]-x_range[i]
    yy[:,i+1]=yy[:,i]+rk4_step(f,x_range[i],yy[:,i],h)

y_true = (1/(np.e**(np.arctan(-20))))*np.e**(np.arctan(x_range))
plt.plot(x_range,yy[0,:],'r',label='RK4')
plt.plot(x_range,y_true, 'k',linestyle='--', label='True')
plt.legend()
print(np.std(yy[0,:]-y_true))
plt.show()

plt.plot(x_range[70:120],yy[0,70:120]-y_true[70:120] )
plt.show()

print('number of func eval:', f.counter)
```

```
0.00012264179233945375
```






```
number of func eval: 792
```

Part (b)

Here we use the following equations to show how comparing the two methods of a single step and two half steps can cancel out the 5th order error term:

$$y(x+h) = y_1 + h^5\phi + O(h^6) + \ldots$$

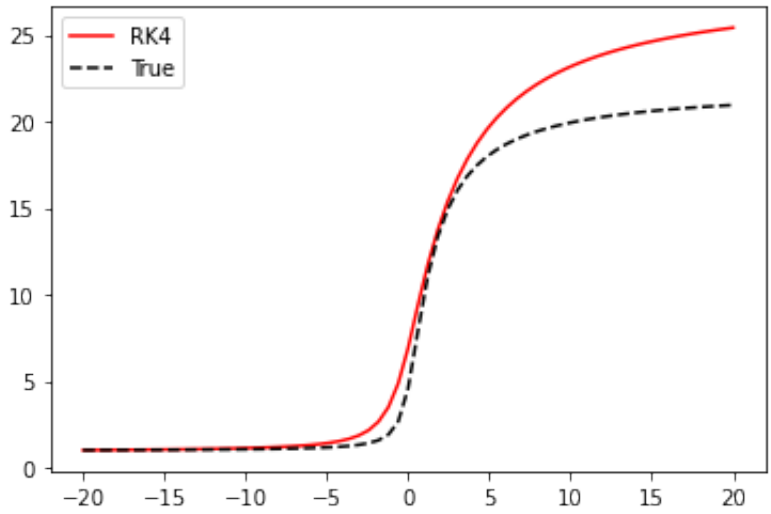
$$y(x+h) = y_2 + 2(h/2)^5\phi + O(h^6) + \ldots$$

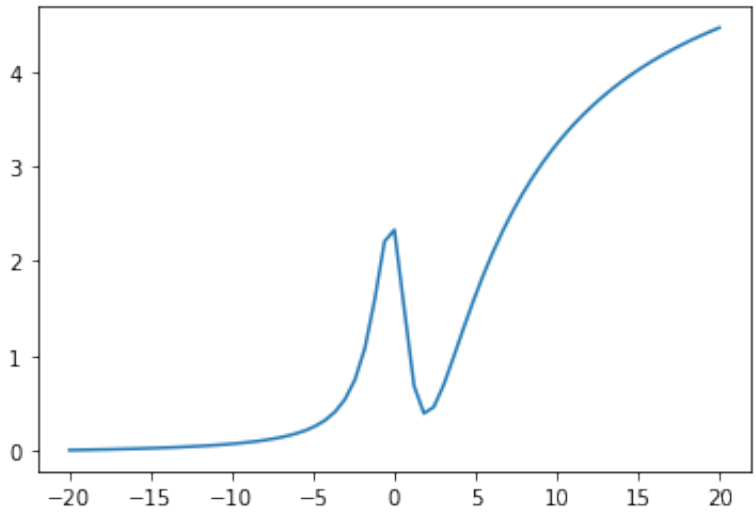
$$\Delta = y_2 - y_1$$

$$y(x+h) = y_2 + \Delta/15 + O(h^6)$$

In [350]:
```
def f(x,y):
    f.counter += 1
    dydx = y / (1 + x**2)
    return dydx
f.counter = 0

def rk4_stepd(fun,x,y,h):

    x2 =  (x + h)/ 2

    #evaluate for the full h step
    dy_h = rk4_step(f,x,y,h)

    #evaluate twice for half the h step
    dy_h_2= rk4_step(f,x,y,h/2)
    dy_h_2 = rk4_step(f,x2,y+dy_h_2,h/2)

    y1 = dy_h - (h)**5
    y2 = dy_h_2 - 2*(h/2)**5

    delta = y2 - y1

    #calculate final dy by comparing methods
    dy = y2 + delta/ 15

    return dy


npt = 67
x_range = np.linspace(-20, 20, npt)
yy=np.zeros([2,npt])

yy[0,0]=1 #starting conditions
yy[1,0]=0
for i in range(npt-1):
    h=x_range[i+1]-x_range[i]
    h_half = (x_range[i+1]-x_range[i])/2
    yy[:,i+1]=yy[:,i]+rk4_stepd(f,x_range[i],yy[:,i],h)
y_true = (1/(np.e**(np.arctan(-20))))*np.e**(np.arctan(x_range))
plt.plot(x_range,yy[0,:],'r',label='RK4')
plt.plot(x_range,y_true, 'k',linestyle='--', label='True')
plt.legend()
print(np.std(yy[0,:]-y_true))
plt.show()

plt.plot(x_range,yy[0,:]-y_true )
plt.show()

print('number of func eval:', f.counter)
```

```
1.6395136518708804
```






```
number of func eval: 792
```

The original rk4 step does 4 function evaluations for every step. The updated version takes 11 new function evaluations since there are 4 from the full step of h, 3 from the first half step of h (the first one is shared with the full stepper) and 4 from the second half step. Therefore we need to take approximately 1/3 the steps in the newer version to get the same number of function evaluations as the original integrator. However, I was struggling with saving the shared point for each step in the second method so there are still 12 function evaluations occuring which is why it doesn't quite work for 200 points. I'm showing above for 199 points because that shows the same number of function evaluations but I see why that would be incorrect and what needs to happen to fix it. Even with 199 points we can see that the second rk4 method (rk4_stepd) is less accurate because this method depends on a high number of steps.

Problem 2

In order to set up the chain we need to write each half life in a common time unit, I picked years. Instead of writing the Bateman equations out like Jon did in class I converted the halflifes to decay times and put them in a matrix. This matrix has the negative decay times on the diagonal and the positive ones below on the off diagonal. Dotting this matrix with the amounts of the elements (starting with just U238) allows you to calculate the derivative of the change of the element amount with respect to time. Putting this function through an integrator we can calculate the entire chain. I used the Radau method for solving the IVP. This is because the differential equations in the chain are stiff and require an extremely small step size to remain stable since the decay times vary so much.

I picked a time period of 7 billion years to capture the entire chain.

In [351]:
```
import numpy as np
from scipy import integrate
import time
from matplotlib import pyplot as plt

#half lifes of each element in years (14)
half_life = [4.47E+09,0.066027,7.65E-04,245500,75380,1600,0.010475,5.90E-06,5.10E-05,3.79E-05,5.21E-12,22.3,5.015,0.37911]
elem = np.array(['U238', 'T234','P234','U234','T230','R226','R222','P218','Po214','B214','Pl214','Po210','B210','Pl210','P206'])

#matrix will be 15 by 15, with P206 never having a half life as it is stable
#convert the half-lifes into lambda
m = np.zeros([15,15])
for i in range(len(half_life)):
    m[i,i] = - np.log(2)/half_life[i]
    m[i+1,i] = np.log(2)/half_life[i]
print(m)
```

```
[[-1.55066483e-10  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.55066483e-10 -1.04979354e+01  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.04979354e+01 -9.06074746e+02  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
```

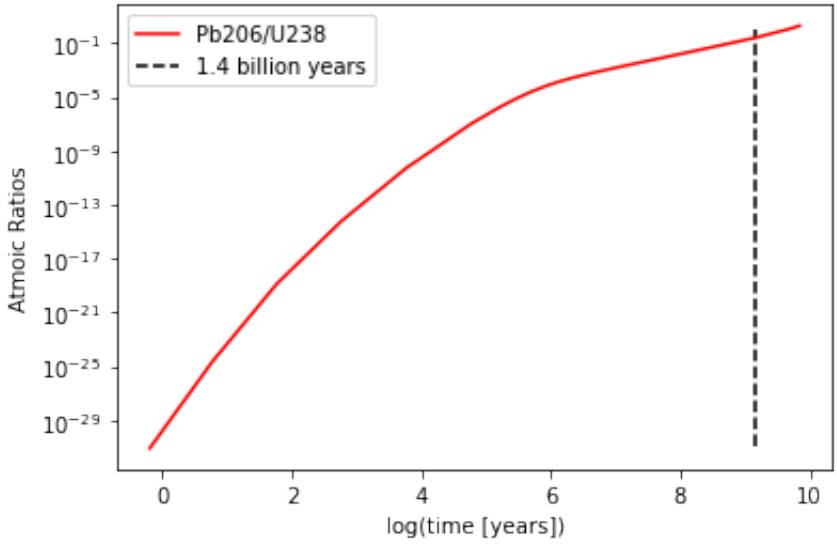
```
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  9.06074746e+02 -2.82341010e-06
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  2.82341010e-06
 -9.19537252e-06  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  9.19537252e-06 -4.33216988e-04  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  4.33216988e-04 -6.61715685e+01  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  6.61715685e+01 -1.17482573e+05
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  1.17482573e+05
 -1.35911212e+04  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  1.35911212e+04 -1.82888438e+04  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  1.82888438e+04 -1.33041685e+11  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  1.33041685e+11 -3.10828332e-02
  0.00000000e+00  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  3.10828332e-02
 -1.38214792e-01  0.00000000e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  1.38214792e-01 -1.82835372e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  1.82835372e+00  0.00000000e+00]]
```

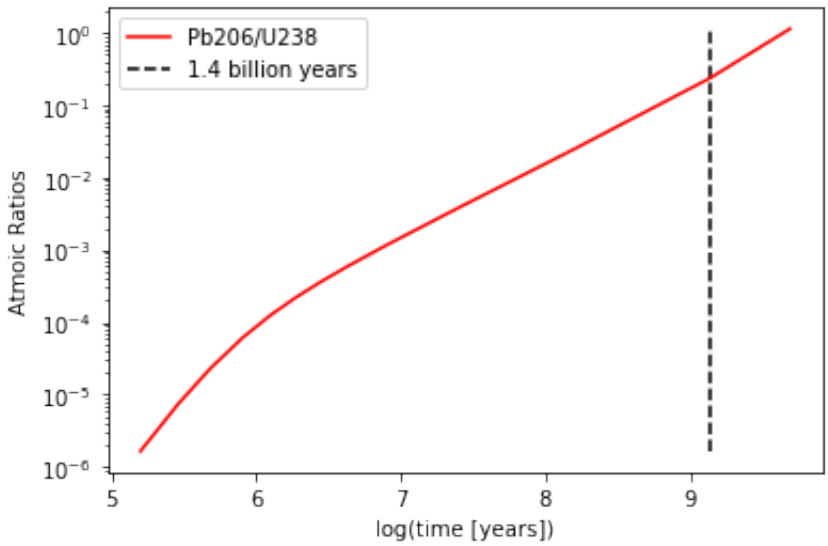
In [352]:
```
def m_fun(x,y,half_life=m):
    dydx = np.zeros(len(half_life))
    dydx[:] = half_life@y
    return dydx

#fill first entry of y0 with U238 amount
y0=np.zeros(15)
y0[0] = 100
x0=0
x1=7e9

ans_stiff=integrate.solve_ivp(m_fun,(x0,x1),y0,method='Radau')
y = ans_stiff.y
t = ans_stiff.t
```

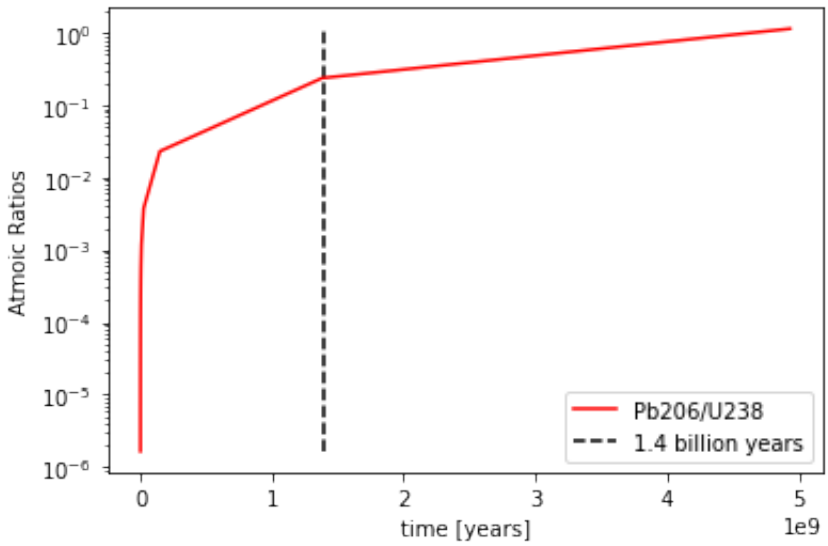
U238/ Pb206 Ratio Plots

In [353]:
```
#select the amounts for each element from the integrator
U238 = y[0,:]
Pb206 = y[-1,:]

plt.plot(np.log10(t),Pb206/U238,color='r',label='Pb206/U238')
plt.vlines(x=np.log10(1.4e9), ymin=1e-31, ymax = 10e-1 , colors='k', l
s='--',label='1.4 billion years')
plt.yscale('log')
plt.xlabel('log(time [years])')
plt.ylabel('Atmoic Ratios')
plt.legend()
plt.show()

ratio = Pb206[7:-1]/U238[7:-1]
plt.plot(np.log10(t[7:-1]),ratio,color='r',label='Pb206/U238')
plt.vlines(x=np.log10(1.4e9), ymin=ratio.min(), ymax =ratio.max() , co
lors='k', ls='--',label='1.4 billion years')
plt.yscale('log')
plt.xlabel('log(time [years])')
plt.ylabel('Atmoic Ratios')
plt.legend()
plt.show()

plt.plot((t[7:-1]),ratio,color='r',label='Pb206/U238')
plt.yscale('log')
plt.vlines(x=1.4e9, ymin=ratio.min(), ymax =ratio.max() , colors='k',
ls='--',label='1.4 billion years')
plt.xlabel('time [years]')
plt.ylabel('Atmoic Ratios')
plt.legend()
plt.show()
```

```
<ipython-input-353-6ecb08ba53cd>:5: RuntimeWarning: divide by zero e
ncountered in log10
  plt.plot(np.log10(t),Pb206/U238,color='r',label='Pb206/U238')
```


Pb206/U238
1.4 billion years
Atmoic Ratios
log(time [years])


Pb206/U238
1.4 billion years
Atmoic Ratios
log(time [years])

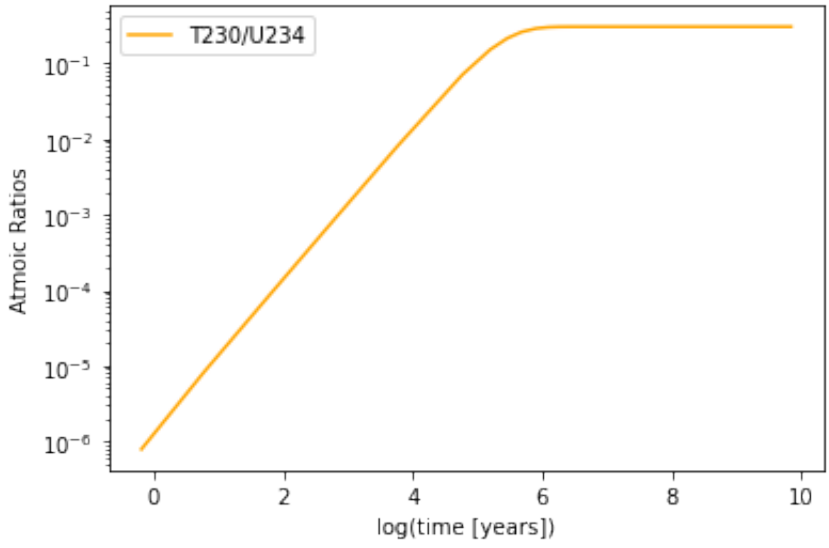

Pb206/U238
1.4 billion years
Atmoic Ratios
time [years]
1e9

We can see from the above graphs of the Pb206/U238 ratio that it changes very quickly at the beginning of the time period (first 1.4 billion years) and slowly approaches 1 in the last 3 billion years. This behavior is easied to see versus linear time, however the logged time alows us to see what is going in in the early time frame where the change is occuring very rapidly. This behavior is a product of all the elements following U238 in the chain having much shorter relative half lifes. We can see why it's realistic to estimate the U238 decaying instantly to lead since relatively speaking, the ratio approaches converging on it's final value very quickly.

T230/U234 Ratio Plots

In [354]:
```
U234 = y[3,:]
T230 = y[4,:]

plt.plot(np.log10(t),T230/U234,color='orange',label='T230/U234')
plt.yscale('log')
plt.xlabel('log(time [years])')
plt.ylabel('Atmoic Ratios')
plt.legend()
plt.show()
```

```
<ipython-input-354-ef718d1cb148>:4: RuntimeWarning: divide by zero e
ncountered in log10
  plt.plot(np.log10(t),T230/U234,color='orange',label='T230/U234')
<ipython-input-354-ef718d1cb148>:4: RuntimeWarning: invalid value en
countered in true_divide
  plt.plot(np.log10(t),T230/U234,color='orange',label='T230/U234')
```

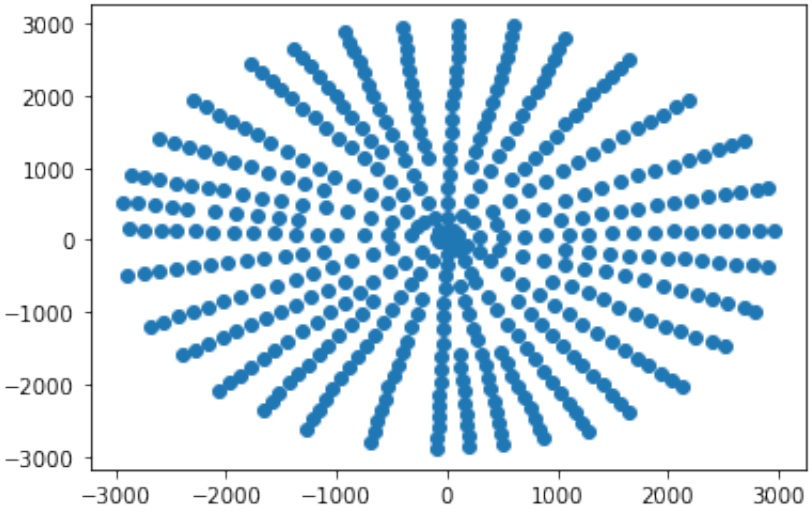


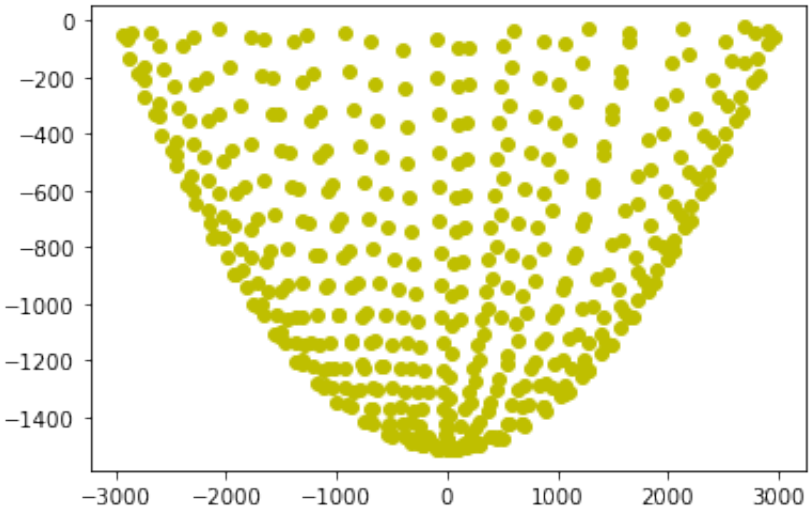
Problem 3

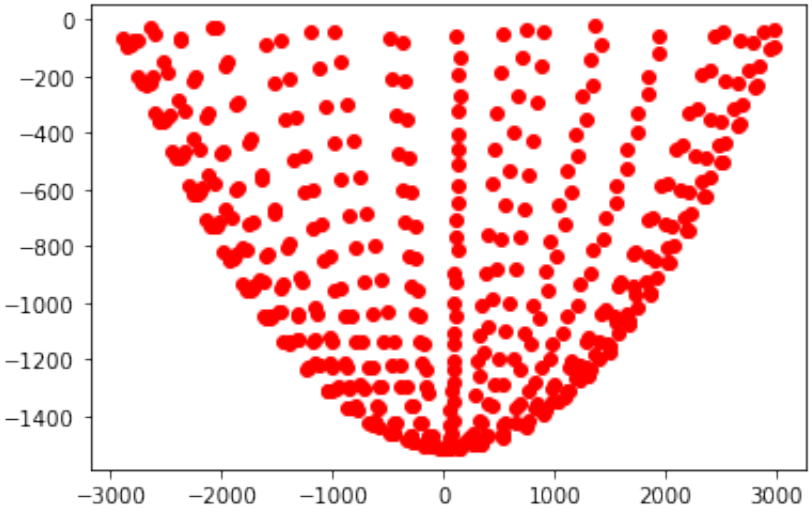
In [355]:
```
#load data
data = np.loadtxt('dish_zenith.txt')
x = data[:,0] ; y = data[:,1] ; z=data[:,2]
```

We can take a look at the data in the different planes:

In [356]:
```
plt.scatter(x,y)
plt.show()
plt.scatter(x,z, color='y')
plt.show()
plt.scatter(y,z, color='r')
plt.show()
```


3000
2000
1000
0
−1000
−2000
−3000
−3000 −2000 −1000 0 1000 2000 3000


0
−200
−400
−600
−800
−1000
−1200
−1400
−3000 −2000 −1000 0 1000 2000 3000


0
−200
−400
−600
−800
−1000
−1200
−1400
−3000 −2000 −1000 0 1000 2000 3000

Part (a)

In order to linearlize the paraboloid we can expand out the equation:

$$z - z_0 = a(x^2 - 2xx_0 + x_0^2 + y^2 - 2yy_0 + y_0^2)$$

$$z = ax^2 + ay^2 - 2x_0 ax - 2y_0 ay + (ax_0^2 + ay_0^2 + z_0)$$

We can identify the new parameters:

$$C_1 = a$$

$$C_2 = -2x_0 a$$

$$C_3 = -2y_0 a$$

$$C_4 = (ax_0^2 + ay_0^2 + z_0)$$

Our new linear equation is then:

$$z = C_1(x^2 + y^2) + C_2 x + C_3 y + C_4$$

We can write this as a matrix equation to make our lives easier:

$$\begin{bmatrix} z \\ . \\ . \\ z_n \end{bmatrix} = \begin{bmatrix} (x^2 + y^2) & x & y & 1 \\ . & . & . & . \\ . & . & . & . \\ (x_n^2 + y_n^2) & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix}$$

Part (b)

Writing our matrix equation in terms of linear -least squared fit:

$\langle d \rangle = Am$

Where d is the data, A is what maps the model parameters to the data and m is the model parameters. Using the data we have we can create these matrices and do a fit using SVD to calculate m. We want to use the pseudoinverse because the problem isn't too large and we can avoid the inverse blowing up with a factor of S^{-2} in the SVD:

$m = A^{\dagger} d$

In [357]:
```
d = z

A = np.zeros(shape=(x.shape[0],4))
A[:,0] = (x**2+ y**2)
A[:,1] = x
A[:,2] = y
A[:,3] = 1

m = np.linalg.pinv(A)@d
c1 = m[0] ; c2 = m[1] ; c3 = m[2] ; c4 = m[3]


print('The best fit parameters are: C1:{0:.5f},C2:{1:.5f},C3:{2:.5f},C
4:{3:.5f}'.format(c1,c2,c3,c4))
```

```
The best fit parameters are: C1:0.00017,C2:0.00045,C3:-0.01941,C4:-1
512.31182
```

We have to solve the system of equations we made to define the new parameters to get the true values for a, x_0, y_0 and z_0:

$a = C_1$

$x_0 = \frac{C_2}{-2C_1}$

$y_0 = \frac{C_3}{-2C_1}$

$z_0 = C_4 - \frac{C_2^2}{4C_1} - \frac{C_3^2}{4C_1}$

In [358]:
```
a = c1
x0 = c2 /(-2*c1)
y0 = c3 /(-2*c1)
z0 = c4 - ((c2**2)/(4*c1))- ((c3**2)/(4*c1))

print('a:{0:.5f},x0:{1:.5f},y0:{2:.5f},z0:{3:.5f}'.format(a,x0,y0,z0))
```

```
a:0.00017,x0:-1.36049,y0:58.22148,z0:-1512.87721
```

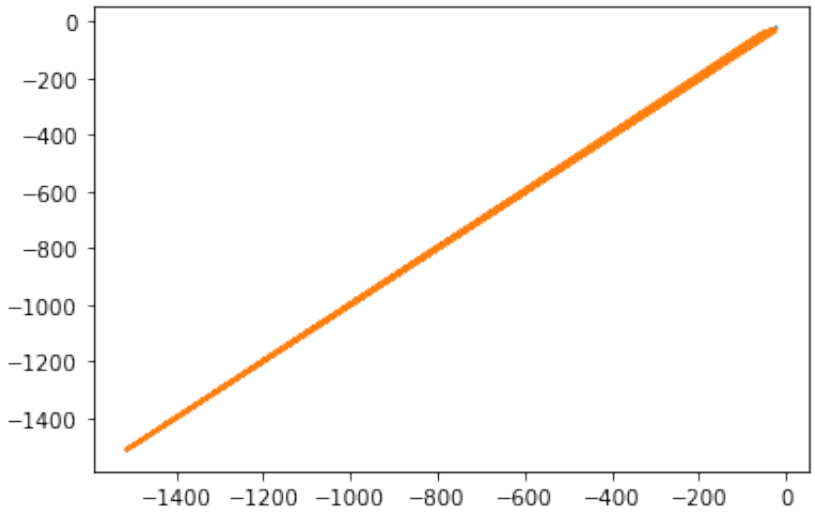
Part (c)

Below we estimate the noise from the data and the uncertainties in the parameters using the following equation:

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

In [359]:
```
d_true = A@m
mynoise=np.std(d-d_true)
N=np.eye(z.shape[0])*mynoise**2
Ninv=np.eye(z.shape[0])*mynoise**-2
mat=A.T@Ninv@A
errs=np.linalg.inv(mat)
print('parameter errors are ',np.sqrt(np.diag(errs)))
a_err = np.sqrt(np.diag(errs))[0]
plt.plot(z,d)
plt.plot(z,d_true)
plt.show()

#noise from the data
derrs=A@errs@A.T
model_sig=np.sqrt(np.diag(derrs))
```

```
parameter errors are  [6.45189976e-08 1.25061100e-04 1.19249564e-04
3.12018436e-01]
```



If we take a slice of the paraboloid through the $y = 0$ plane then we get a parabola in the center of the paraboloid. According to the equation for focal length of a parabola:

$$f = x^2/4d$$

Where x is the radius and d is the depth of the parabola. In our case the depth is z_0. The radius can be calculated taking the inverse of a since this parameter dictates the diameter of the circle in the $x - y$ plane. Halfing this value gives the radius.

We can calculate the focal length we get from our model fit and convert it into meters.

In [360]:
```
d = 1/a
r = d/2

f = r**2/(4*np.abs(z0)) * 1e-3
print('Focal length from fit: {0:.3f} m'.format(f))
```

```
Focal length from fit: 1.487 m
```

Another perhaps more accurate way to measure the focal length is assuming that the parabola is centered at $(0, 0, 0)$ which would yield the equation:

$z = ax^2 = x^2/(4f)$

Which would give us $f = 1/(4a)$

In [361]:
```
f1= 1/ (4*a)
print('Focal length from fit: {0:.5f} m'.format(f1* (1e-3)))
```

```
Focal length from fit: 1.49966 m
```

It makes sense that the second method for measuring the focal length would be more accurate since it only takes into account one parameter so there are less sources of error. I will then use this definition to propagate the errors. As we saw above the error on a is 6.45189976×10^8. To propagate the error we can use:

$$\frac{\delta f}{f} = \frac{\delta a}{a}$$

In [362]:
```
f_err = f1*(a_err/a)

print('Focal length from fit: {0:.5f} m +/- {1:.5f} m'.format(f1* (1e-3),f_err*1e-3))
```

```
Focal length from fit: 1.49966 m +/- 0.00058 m
```

This is within our target focal length.