In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Problem 1

Plot

$P(x) = (x-2)^9$

Also represented by:

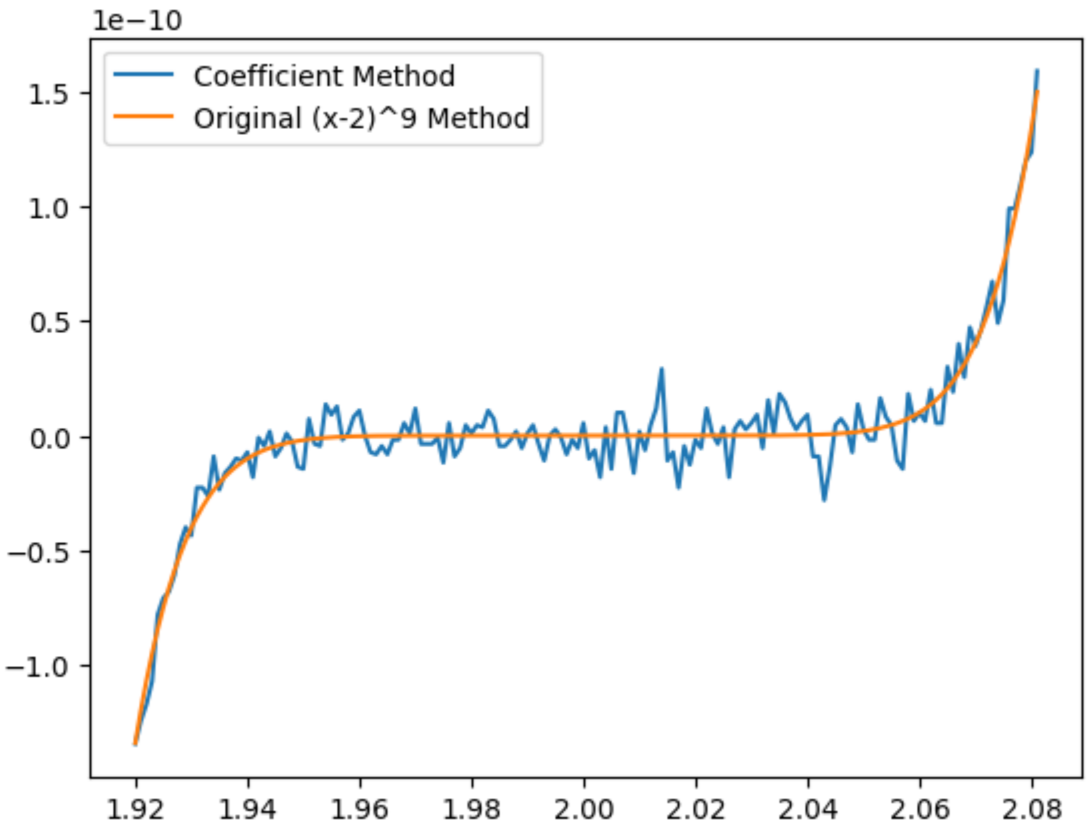
$$P(x) = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 51$$

In [3]:
```
#setting up x for problem 1
x=np.arange(1.920, 2.081, 0.001)
```

In [6]:
```
#inputting the two versions of P(x)

p_coef = x**9-18*x**8+144*x**7-672*x**6+2016*x**5-4032*x**4+5376*x**3-4608*x**2+230
p_9 = (x-2)**9
```

In [7]:
```
plt.plot(x,p_coef, label = 'Coefficient Method')
plt.plot(x,p_9, label = "Original (x-2)^9 Method")
plt.legend()
```

Out[7]:
```
<matplotlib.legend.Legend at 0x1f477ffb950>
```



This discrepancy is due to the number of subtraction operations that are done. In the coefficient method, we do 5 subtractions, each losing some information. For the original method, we only do one, losing much less information. Therefore the original method is better and more "correct" (although it also still has some error, just much less than the coefficient method).

Problem 2

Part i:

In [38]:
```
#Establishing a "small" number as a
a = 0.00000001

#calculating the output with the original function
# and a newer optimized version
original1 = np.sqrt(a+1)-1
new1 = (a)/(np.sqrt(a+1)+1)
```

In [39]:
```
print(original1)
print(new1)
```

```
4.999999969612645e-09
4.9999999875e-09
```

Here the original method led to floating point errors but the optimized version did not. This was done by multiplying the top and bottom of the original equation by the conjugate of the binomial to remove any subtraction.

In [40]:
```
#Establishing a "small" number as b
b = 0.00000002

#calculating the output with the original function
# and a newer optimized version
original2= np.sin(a)-np.sin(b)
new2 = 2*np.cos((a+b)/2)*np.sin((a-b)/2)
```

In [41]:
```
print(original2)
print(new2)
```

```
-1e-08
-9.999999999999999e-09
```

Here the original method once again led to errors, as the "correct" answer here is the second, which was found by a modified function taking advantage of trigonometric functions.

In [30]:
```
#calculating the output with the original function
# and a newer optimized version

original3 = (1-np.cos(a))/np.sin(a)
new3 = np.sin(x)
```

In [31]:
```
print(original3)
print(new3)
```

```
0.0
1e-16
```

Here the original method actually gave zero, which is not correct for the input, but the modified version, which took advantage of conjugates and trigonometric identities worked and gave the correct output.

Problem 3

This question asks us to use the taylow poynomial approximation of a function to find and investigate error and the bounds on it.

a)

Here the taylor polynomial

$P_2(x) = 1 + x - (x^2)/2$

was used to approximate $f(0.5)$ to compare the values and find their error.

In [47]:
```
#This is where the different functions
#are defined for use later

c=0.5
P2=1+c-(c**2)/2
f=(1+c+c**3)*np.cos(c)
#here the max error is calculated using
max_error = (0.5*(-1.625)*np.sin(0.5)+0.5*1.75*np.cos(0.5))/(1.625*np.cos(0.5))
```

In [48]:
```
print("P2(0.5)=",P2)
print("f(0.5)=",f)
print("The error is:",f-P2)
print("The max error is:", max_error)
```

```
P2(0.5)= 1.375
f(0.5)= 1.4260716630718557
The error is: 0.05107166307185573
The max error is: 0.2653102935396432
```

These errors work, as the found error is within the bound of the maximum calculated.

b)Find a bound for the error in general as a function of x.
In this case, the bound for the error is:

$$|f(x) - P_2(x)| \leq -(x^3 - x - 1)sin(x) + (3x^2 + 1)cos(x)$$

Which was found from:

$$|Error| \leq f'(x)(x/f(x))$$

c)
Approximate the integral of $f(x)$ from 0 to 1 using P_2.

The integral $\int_0^1 P_2(x)dx = 1 + 0.5 - 1/6 = 4/3$

d)
The approximate error should be roughly equal to the magnitude of the integral of the next term of the taylor series. So:

$$\int_0^1 0.5x^3 dx = 0.125 = Error$$

Problem 4

In [55]:
```python
a = 1
b = -56
c = 1

#r1 is the negative root, r2 is the positive root


#from calculator
r1 = 55.98213716
r2 = 0.017862841


#With 3 decimal point assumption
r1rounded = (-b+round(np.sqrt(b**2-4*a*c),3))/2*a
r2rounded = (-b-round(np.sqrt(b**2-4*a*c),3))/2*a

relError1 = abs(r1-r1rounded)/abs(r1)
relError2 = abs(r2-r2rounded)/abs(r2)


print("The calculated first root is:",r1rounded)
print("The calculated second root is:",r2rounded)
print("The relative error for the first root is:", relError1)
print("The relative error for the second root is:", relError2)
```

```
The calculated first root is: 55.982
The calculated second root is: 0.018000000000000682
The relative error for the first root is: 2.450067235005535e-06
The relative error for the second root is: 0.007678453836132857
```

Here you can see that the errors have vastly different orders of magnitude.

b)

Find and apply a "better" approximation using $(x - r_1)(x - r_2)$

The better relations are:

$r_1 + r_2 = -b/a$

&

$r_1 r_2 = c/a$

These are obtained by dividing the entire polynomial by a, then noting that when foiling out $(x - r_1)(x - r_2)$ these relations occur.

```
In [56]: #Applying the better approximations:

         r2new = c/(a*r1rounded)

         r2newerror = (r2-r2new)/r2

         print("The new second root is:", r2new)
         print("The new relative error for the second root is:", r2newerror)
```

```
The new second root is: 0.017862884498588832
The new relative error for the second root is: -2.4351439298820114e-06
```

Problem 5:

Consider two inputs of the form $\tilde{x} = x + \Delta x$

When added or subtracted, their respecive errors Δx also add or subtract.

a) Find the upper bounds on the absolute error:
$|\Delta y|$

And relative error:

$|\Delta y|/|y|$

The upper bound on the absolute error would be:

$|\Delta y| \leq |\Delta x_1| + |\Delta x_2|$

And the upper bound on the relative error is:

$|\Delta y|/|y| \leq (|\Delta x_1| + |\Delta x_2|)/|y|$

This shows that the relative error will be large when $y << |\Delta x_1| + |\Delta x_2|$

b)
We find through trig identities that

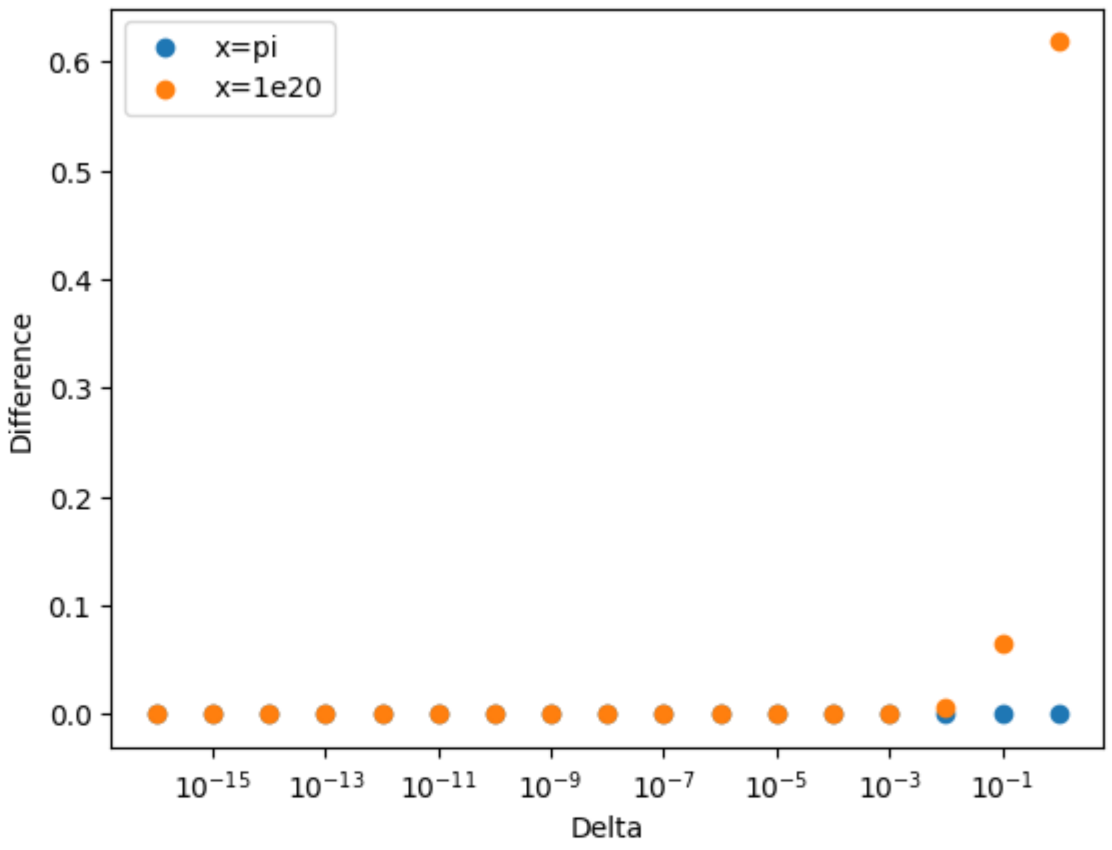
$$cos(x+\delta)-cos(x)$$

turns into

$$-2sin((2x+\delta)/2)sin(\delta/2)$$

In [90]:
```
#establishing the two values for x
# (I found that 10^20 had a more interesting graph than 10^6)
x1 = np.pi
x2 = 1e20

#Establishing delta values
delta = np.array([1e-16,1e-15,1e-14,1e-13,1e-12,1e-11,1e-10,1e-9,1e-8,1e-7,1e-6,1e-

#the difference for x1
newx1 = (-2*np.sin((2*x1+delta)/2)*np.sin(delta/2))
cosOnlyx1 = np.cos(x1+delta)-np.cos(x1)
newx2 = (-2*np.sin((2*x2+delta)/2)*np.sin(delta/2))
cosOnlyx2 = np.cos(x2+delta)-np.cos(x2)
differencePi = newx1-cosOnlyx1
difference6 = newx2-cosOnlyx2
#the difference for x2



print(differencePi)
print(difference6)
plt.scatter(delta,differencePi, label = "x=pi")
plt.scatter(delta,difference6, label = "x=1e20")
plt.legend()
plt.xlabel("Delta")
plt.ylabel("Difference")
plt.xscale('log')
```

```
[-1.22464680e-32  3.21624530e-31  4.76251663e-29  5.00596160e-27
  4.99921986e-25  4.99987795e-23  4.99998817e-21  4.99999919e-19
  4.99999985e-17  3.99639096e-18 -4.44503438e-17 -4.14055274e-18
  2.61967893e-17  7.87602998e-18 -1.67170422e-17  2.51534904e-17
 -5.55111512e-17]
[6.45251285e-17 6.45251285e-16 6.45251285e-15 6.45251285e-14
 6.45251285e-13 6.45251285e-12 6.45251285e-11 6.45251285e-10
 6.45251285e-09 6.45251285e-08 6.45251285e-07 6.45251285e-06
 6.45251285e-05 6.45251258e-04 6.45248597e-03 6.44982464e-02
 6.18699890e-01]
```



c) Create a taylor expansion based algorithm to approximate the previous cosine function.

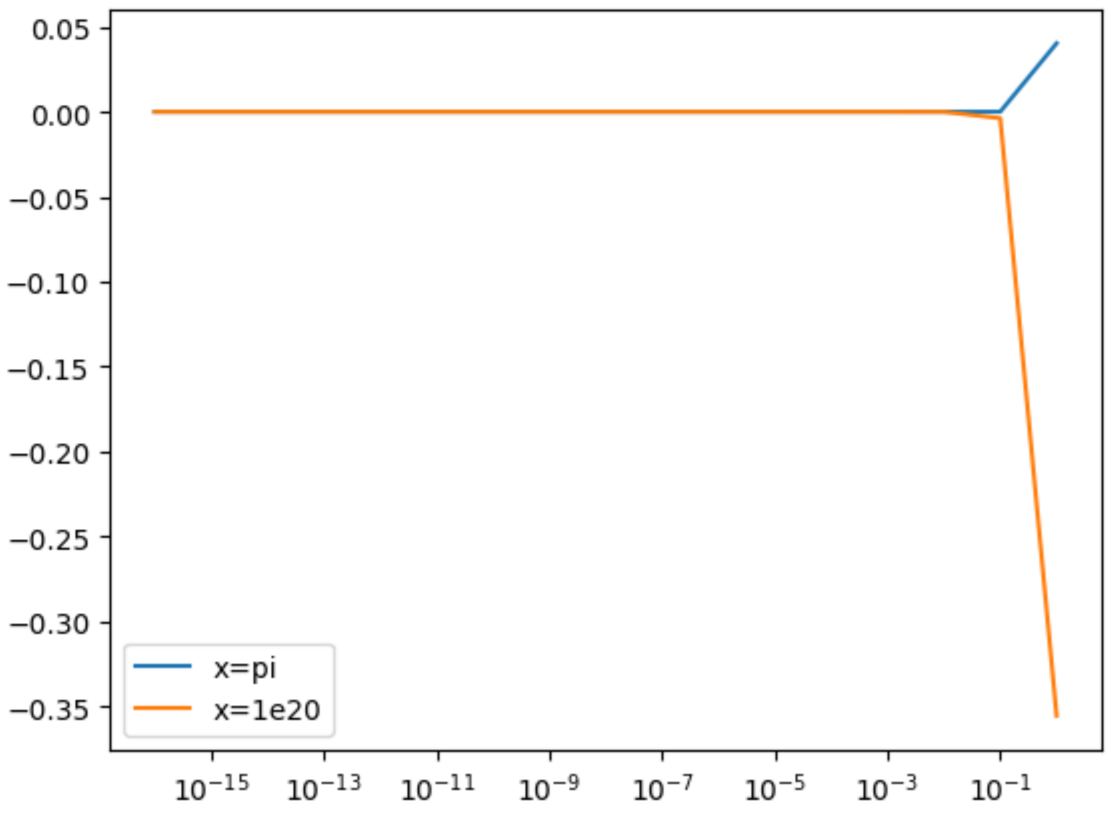
I decided to approximate it with:

$$f(x+\delta) - f(x) = \delta f'(x) + (\delta^2 f''(x))/2$$

As it keeps the delta squared values but not any past. Past this the amount of data lost should be around the same scale as the next taylor series component, which will have a δ^3 out front, which is extremely small.

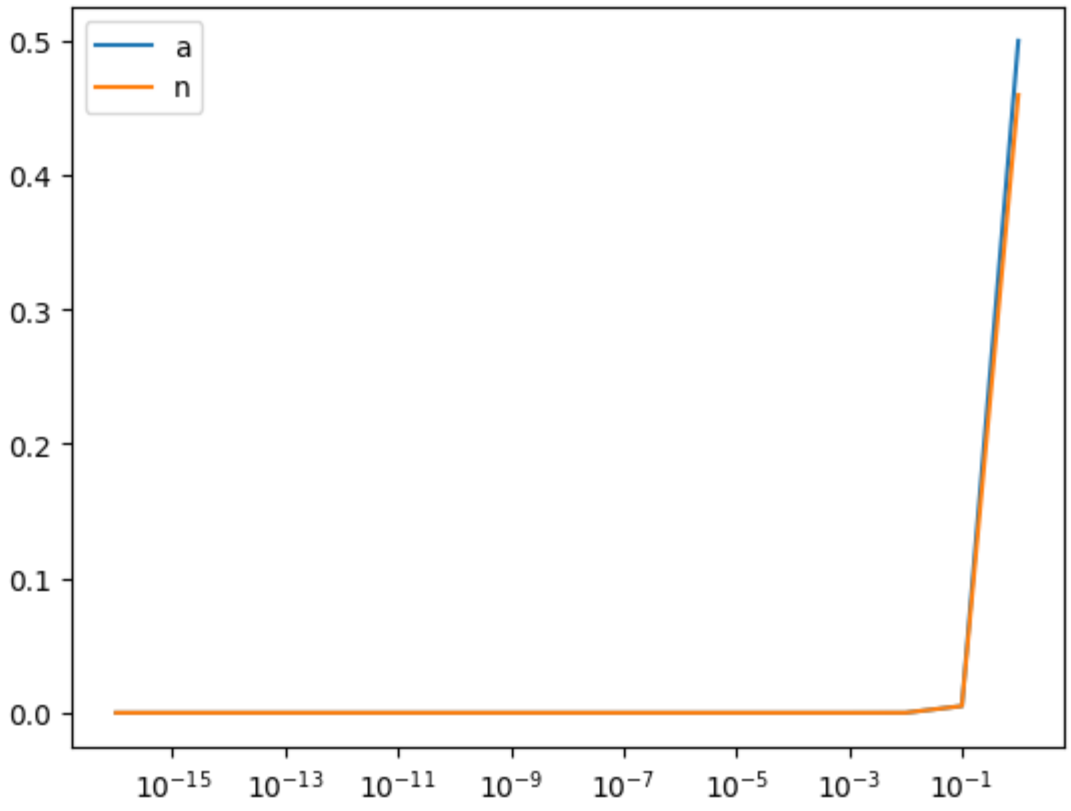
In [92]:
```
#These are the two new calculations using the new algorithm

newAlgo1 = -delta*np.sin(x1)-(delta**2/2)*np.cos(x1)
newAlgo2 = -delta*np.sin(x2)-(delta**2/2)*np.cos(x2)
```

In [ ]:
```
#here the new algorithm is tested against the old one to see how well they agree

plt.plot(delta,newAlgo1-newx1, label = 'x=pi')
plt.plot(delta,newAlgo2-newx2, label = "x=1e20")
plt.legend()
plt.xscale('log')
```



In [ ]:

```
#Here I just plotted the functions themselves as
#  I wanted to see what shapes they were


plt.plot(delta, newAlgo1, label = 'a')
plt.plot(delta,newx1, label = 'n')
plt.legend()
plt.xscale("log")
```



```
In [ ]:
```