In [351]:

```python
import scipy.stats as si
from yahoo_fin.stock_info import get_data
from yahoo_fin import options
from pandas_datareader import data
import yfinance as yf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from scipy.stats import norm
from datetime import datetime
from datetime import date

def newton(f, df, x0, times = 100000):
    """
    Finding root using linear approximation
    At any point in f(x), we have df/dx = f(x)' = (f(x+a) - f(x))/a
    f(x) = (x-a)*f(a)' + f(a)
    when x is the root x0, we have 0 = (x0-a)*f(a)' + f(a)
    newton's method converge fast but not nessarily gives you an answer
    """
    xn = x0
    error = pow(10, -5)
    for n in range(0,times):
        fxn = f(xn)
        if abs(fxn) <= error:
            return xn
        dfxn = df(xn)
        if dfxn == 0:
            print("Zero derivative. Change your starting point.")
            return None
        xn = xn - fxn/dfxn
    print("So many iterations, over ",times, ".")
    return None
def bisection(x1,x2,f, times = 10000):
    """
    Assumption: f(x) is continous and start with x1 and x2 such that f(x1)*f(x2) < 0
    With the starting interval [x1, x2], we check if the mid point x0 = (x1+x2)/2 is the root
    To determine the next interval [x3, x4], if f(x0)*f(x1) <0 then [x0, x1] becomes the new interval and repeat
    Tolarance = 10^-6
    """
    error = pow(10, -6)
    if f(x1)*f(x2) > 0:
        print("Bad Input, please rechoose x1 and x2.")
        return None
    x0 = 0.5*(x1 + x2)
    counter  = 1
    while abs(f(x0)) > error:
        if f(x1)*f(x0) < 0:
            x2 = x0
        else:
            x1 = x0
        counter += 1
        x0 = 0.5*(x1 + x2)
        if counter > times:
            print("So many iterations, over ",times, ".")
            return None
    return x0
def get_IV(bid, ask, S0, K, T, r,type = "call", method = "Bisection"):
    option_price = 0.5 * (bid + ask)
    if type == "call" and method == "Bisection":
        def f(x):
            return vanilla_option(S0, K, T,r,x) - option_price
        return bisection(-0.1,1.5,f)
    elif type == "put" and method == "Bisection":
        def f(x):
            return vanilla_option(S0, K, T,r,x, option = "put") - option_price
        return bisection(-1,2,f)
    elif type == "call" and method == "Newton":
        def g(x):
            return vanilla_option(S0, K, T,r,x, option = "call") - option_price
        def dg(x):
            return vega(S0,K,T,r,x)
        return newton(g, dg, 0.3, times = 10000)
    elif type == "put" and method == "Newton":
        def g(x):
            return vanilla_option(S0, K, T,r,x, option = "put") - option_price
        def dg(x):
            return vega(S0,K,T,r,x)
        return newton(g, dg,0.3, times = 10000)

def vanilla_Euro_option(S, K, T, r, vol, option = 'call'):

    #S: spot price
    #K: strike price
    #T: time to maturity
    #r: interest rate
    #sigma: volatility of underlying asset

    d1 = (np.log(S/K)+(0.5 * vol**2 + r)*T)/(vol*np.sqrt(T))
    d2 = d1 - vol * np.sqrt(T)

    if option == 'call':
        result = (S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.cdf(d2, 0.0, 1.0))
    if option == 'put':
        result = (K * np.exp(-r * T) * si.norm.cdf(-d2, 0.0, 1.0) - S * si.norm.cdf(-d1, 0.0, 1.0))
    return result
```

In [377]:

```python
def Binomial_Tree (S0, K, r, T, vol, N, CP = "call", AE = "E"):
    """
    S0: Stock Price
    K: Strike Price
    r: Risk Free Rate
    T: Time to Maturity
    vol: Volatility
    N: How many time partitions we took
    Assumption: No dividends
    """
    dt = T/N
    u = np.exp(vol*np.sqrt(dt))
    d = 1.0/u
    p = (np.exp(r*dt)-d)/(u-d)

    stock = np.zeros((N+1,N+1))
    option = np.zeros((N+1,N+1))
    stock[0,0] = S0

    for i in range(0, N):
        stock[i+1,0] = stock[i,0]*u
        for j in range (0,i+1):
            stock[i+1, j+1] = stock[i, j]*d

    if CP == "call":
        for x in range(0, N+1):
            option[N,x] = max(stock[N,x] - K, 0)
        for y in range(N-1, -1, -1):
            for x in range(0, y+1):
                option[y, x] = np.exp(-r*dt) * ((1-p)*option[y+1, x+1]+(p*option[y+1,x]))
    elif CP == "put":
        for x in range(0, N+1):
            option[N,x] = max(K - stock[N,x], 0)
        if AE == "E":
            for y in range(N-1, -1, -1):
                for x in range(0, y+1):
                    option[y, x] = np.exp(-r*dt) * ((1-p)*option[y+1, x+1]+(p*option[y+1,x]))
        elif AE == "A":
            for y in range(N-1, -1, -1):
                for x in range(0, y+1):
                    option[y,x] = max(K-stock[y,x], np.exp(-r*dt) * ((1-p)*option[y+1, x+1]+(p*option[y+1,x])))
    return option[0,0]
def error_bio_euro (S, K, T, r, vol, option, N):
    actual = vanilla_Euro_option(S, K, T, r, vol, option)
    calculate = Binomial_Tree (S, K, r, T, vol, N, option, AE = "E")
    error = calculate - actual
    return error

Tree = Binomial_Tree (100, 120, 0.06, 1, 0.2, 1000, CP = "put", AE = "A")
vanilla_Euro_option(100, 120, 1, 0.06, 0.2, option = 'put')
for N in [10,20,30,40,50,100,150,200,250,300,350,400]:
    print("With {:3d} steps, the price is {:.2f}, the absolute error is {:.2f}".format(N,Binomial_Tree (100, 120, 0.06, 1, 0.2, N, CP = "put", AE = "E"), error_bio_euro (100, 120, 1, 0.06, 0.2,"put",N)))
```

```
With  10 steps, the price is 16.59, the absolute error is 0.07
With  20 steps, the price is 16.41, the absolute error is -0.11
With  30 steps, the price is 16.55, the absolute error is 0.03
With  40 steps, the price is 16.49, the absolute error is -0.03
With  50 steps, the price is 16.51, the absolute error is -0.01
With 100 steps, the price is 16.53, the absolute error is 0.01
With 150 steps, the price is 16.53, the absolute error is 0.00
With 200 steps, the price is 16.52, the absolute error is 0.00
With 250 steps, the price is 16.52, the absolute error is -0.00
With 300 steps, the price is 16.52, the absolute error is -0.00
With 350 steps, the price is 16.52, the absolute error is 0.00
With 400 steps, the price is 16.52, the absolute error is -0.00
```

In [382]:

```python
Binomial_Tree (100, 120, 0.06, 1, 0.2, 1000, CP = "put", AE = "E")
```

Out[382]:

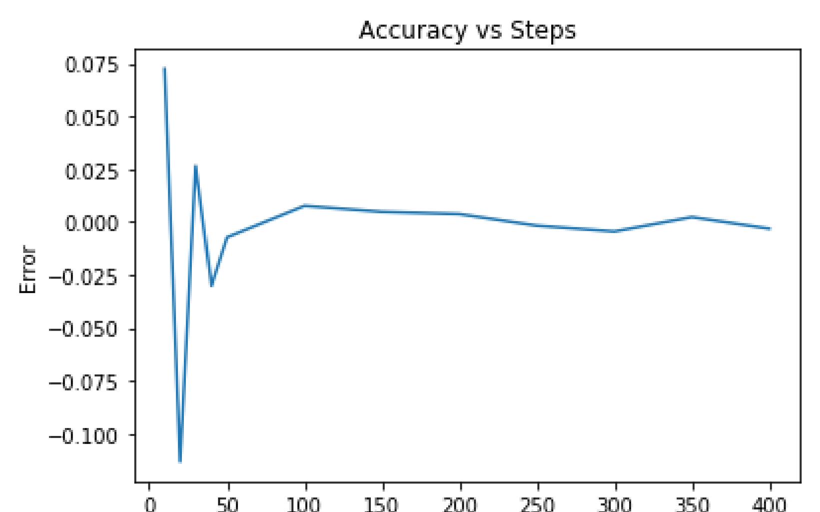
```
16.521908970714083
```

In [392]:

```python
li = [10,20,30,40,50,100,150,200,250,300,350,400]
plt.plot(li,[error_bio_euro (100, 120, 1, 0.06, 0.2,"put",N) for N in li])
plt.ylabel('Error')
plt.xlabel('Number of Steps')
plt.title("Accuracy vs Steps")
```

Out[392]:

```
Text(0.5, 1.0, 'Accuracy vs Steps')
```



In [348]:

```python
def IV_Binomial(bid, ask, S0, K, T, r,type = "call"):
    option_price = 0.5 * (bid + ask)
    if type == "call":
        def f(x):
            return Binomial_Tree(S0, K, r, T, x, 200, CP = "call", AE = "E") - option_price
        return bisection(-0.1,1.5,f)
    elif type == "put":
        def f(x):
            return Binomial_Tree(S0, K, r, T, x, 200, CP = "put", AE = "E") - option_price
        return bisection(-0.1,1.5,f)

def IV_BS(bid, ask, S0, K, T, r,type = "call", method = "Bisection"):
    option_price = 0.5 * (bid + ask)
    if type == "call" and method == "Bisection":
        def f(x):
            return vanilla_option(S0, K, T,r,x) - option_price
        return bisection(-0.1,1.5,f)
    elif type == "put" and method == "Bisection":
        def f(x):
            return vanilla_option(S0, K, T,r,x, option = "put") - option_price
        return bisection(-1,2,f)
    elif type == "call" and method == "Newton":
        def g(x):
            return vanilla_option(S0, K, T,r,x, option = "call") - option_price
        def dg(x):
            return vega(S0,K,T,r,x)
        return newton(g, dg, 0.3, times = 10000)
    elif type == "put" and method == "Newton":
        def g(x):
            return vanilla_option(S0, K, T,r,x, option = "put") - option_price
        def dg(x):
            return vega(S0,K,T,r,x)
        return newton(g, dg,0.3, times = 10000)
```

In [263]:

```python
# Data pulled on Oct 6th
ZM = yf.Ticker("ZM")
ZM.options
DATA1 = ZM.option_chain("2020-11-05")
tau1 = (datetime(2020,11,5) - datetime.today()).days/365
DATA2 = ZM.option_chain("2020-12-17")
tau2 = (datetime(2020,12,17) - datetime.today()).days/365
DATA3 = ZM.option_chain("2021-01-14")
tau3 = (datetime(2021,1,14) - datetime.today()).days/365
x = ZM.history("period = 1d")["Close"]
S0 = float(x)
rf = 0.09 * 0.01 # Fed funds rate
```

In [383]:

```python
def get_full_table(df, S0, Tau, CP):
    rf = 0.09 * 0.01
    df = df.dropna()
    SD = 0.9 * S0
    SU = 1.1 * S0
    # Stock Price is 477.81, taking ratio [0.9, 1.1], the upper and lower bound of Strike Price should be [430,526]
    df = df[df["strike"] >=SD]
    df = df[df["strike"] <=SU]
    df["Price_Binomial_Euro"] = df.apply(lambda row:Binomial_Tree(S0, row["strike"], rf, Tau, row["impliedVolatility"], 500, CP, AE = "E"), axis = 1)
    df["Price_Binomial_Amer"] = df.apply(lambda row:Binomial_Tree(S0, row["strike"], rf, Tau, row["impliedVolatility"], 500, CP, AE = "A"), axis = 1)
    df["BS_Euro"] = df.apply(lambda row: vanilla_Euro_option(S0, row["strike"], Tau, rf, row["impliedVolatility"], CP), axis = 1)
    df["IV_Binomial"] = df.apply(lambda row:IV_Binomial(row["bid"], row["ask"], S0, row["strike"], Tau, rf, CP) ,axis = 1)
    df["IV_BS"] = df.apply(lambda row: IV_BS(row["bid"], row["ask"], S0, row["strike"], Tau, rf,CP, "Bisection"), axis =1)

    return df
```

In [384]:

```python
DATA1_C = get_full_table(DATA1.calls, S0, tau1, "call")
DATA1_P = get_full_table(DATA1.puts, S0, tau1, "put")
```

In [370]:

```python
DATA2_C = get_full_table(DATA2.calls, S0, tau2, "call")
DATA2_P = get_full_table(DATA2.puts, S0, tau2, "put")
DATA3_C = get_full_table(DATA3.calls, S0, tau3, "call")
DATA3_P = get_full_table(DATA3.puts, S0, tau3, "put")
```

In [387]:

```python
DATA3_P[["strike","bid","ask","impliedVolatility","Price_Binomial_Amer","Price_Binomial_Euro","BS_Euro","IV_Binomial","IV_BS"]]
```

Out[387]:

	strike	bid	ask	impliedVolatility	Price_Binomial_Amer	Price_Binomial_Euro	BS_Euro	IV_Binomial	IV_BS
57	440.0	56.3	57.0	0.756396	55.559694	55.556242	55.564428	0.767604	0.768274
58	450.0	61.1	62.0	0.753573	60.482184	60.478327	60.443959	0.765843	0.765431
59	460.0	66.3	67.1	0.751132	65.578674	65.574306	65.575197	0.762059	0.762989
60	470.0	71.7	73.3	0.753222	71.376227	71.371477	71.355282	0.765126	0.765130
61	480.0	77.2	78.1	0.746524	76.534823	76.529531	76.499188	0.758003	0.758368
62	490.0	82.9	83.8	0.743441	82.174056	82.168217	82.193903	0.754395	0.755253
63	500.0	88.8	89.8	0.740878	88.179803	88.173424	88.138335	0.753384	0.752694
64	510.0	95.1	96.0	0.739413	94.387079	94.379976	94.385213	0.750560	0.751241