In [116]:
```
"""
arj1
Arjun Srivastava
AMATH 301 B
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg
import scipy.integrate
import pandas as pd
```

In [117]:
```
# Problem 1

a = 10
b = 2
theta0 = 1
theta_dot0 = 0
x_true = lambda t : (1/3)*np.exp(-t)*np.sin(3*t) + np.exp(-t)*np.cos(3*t)

# a)

"""
ϑ"(t) + bϑ'(t) + aϑ(t) = 0
v' = (ϑ'
      y')

y = ϑ'

v = (ϑ
     y)

v' = Av

ϑ'   =   0   1    ϑ
y'   =  -10 -2    y

A = [ 0   1

    -10  -2 ]

"""
A = np.array([[0, 1], [-10, -2]])
v0 = np.array([theta0, theta_dot0])

# b)

# Forward Euler Method

f = lambda t, v : A @ v
T = 8

max_err_forward = []
for dt in range(4, 11):
    dt = 2**(-dt)
    t = np.arange(0, T + dt, dt)
    n = t.size
    V = np.zeros((2, n))
    V[:, 0] = v0
    for k in range(n - 1):
        V[:, k + 1] = V[:, k] + dt * f(t[k], V[:, k])
    theta = V[0, :]
    err = np.max(np.abs(x_true(t) - theta))
    max_err_forward.append(err)

# c)

# Backwards Euler
```

```
"""
vk+1 = vk + dt * f(tk+1, vk+1)

vk+1 = vk + dt * A * vk+1

vk+1 - dt * A * vk+1 = vk

vk+1(I - dt * A) = vk

vk+1([[1 0] [0 1]] - dt * [[0 1] [-10 -2]]) = vk

vk+1 = [[1 -dt] [10dt 2dt]]^-1 * vk
"""

max_err_backward = []
for dt in range(4, 11):
    dt = 2**(-dt)
    t = np.arange(0, T + dt, dt)
    n = t.size
    V = np.zeros((2, n))
    V[:, 0] = v0
    A1 = np.array([[1, -dt], [10*dt, 1+2*dt]])
    P, L, U = scipy.linalg.lu(A1)
    for k in range(n - 1):
        y = scipy.linalg.solve_triangular(L, P @ V[:, k], lower=True)
        x = scipy.linalg.solve_triangular(U, y)
        V[:, k + 1] = x
    theta = V[0, :]
    err = np.max(np.abs(x_true(t) - theta))
    max_err_backward.append(err)

# d)

# RK2

max_err_RK2 = []
for dt in range(4, 11):
    dt = 2**(-dt)
    t = np.arange(0, T + dt, dt)
    n = t.size
    V = np.zeros((2, n))
    V[:, 0] = v0
    for k in range(n - 1):
        f1 = f(t[k], V[:, k])
        V[:, k + 1] = V[:, k] + dt * f(t[k] + dt / 2, V[:, k] + (dt / 2) * f1)
    theta = V[0, :]
    err = np.max(np.abs(x_true(t) - theta))
    max_err_RK2.append(err)

# e)

# RK4

max_err_RK4 = []
for dt in range(4, 11):
    dt = 2**(-dt)
```

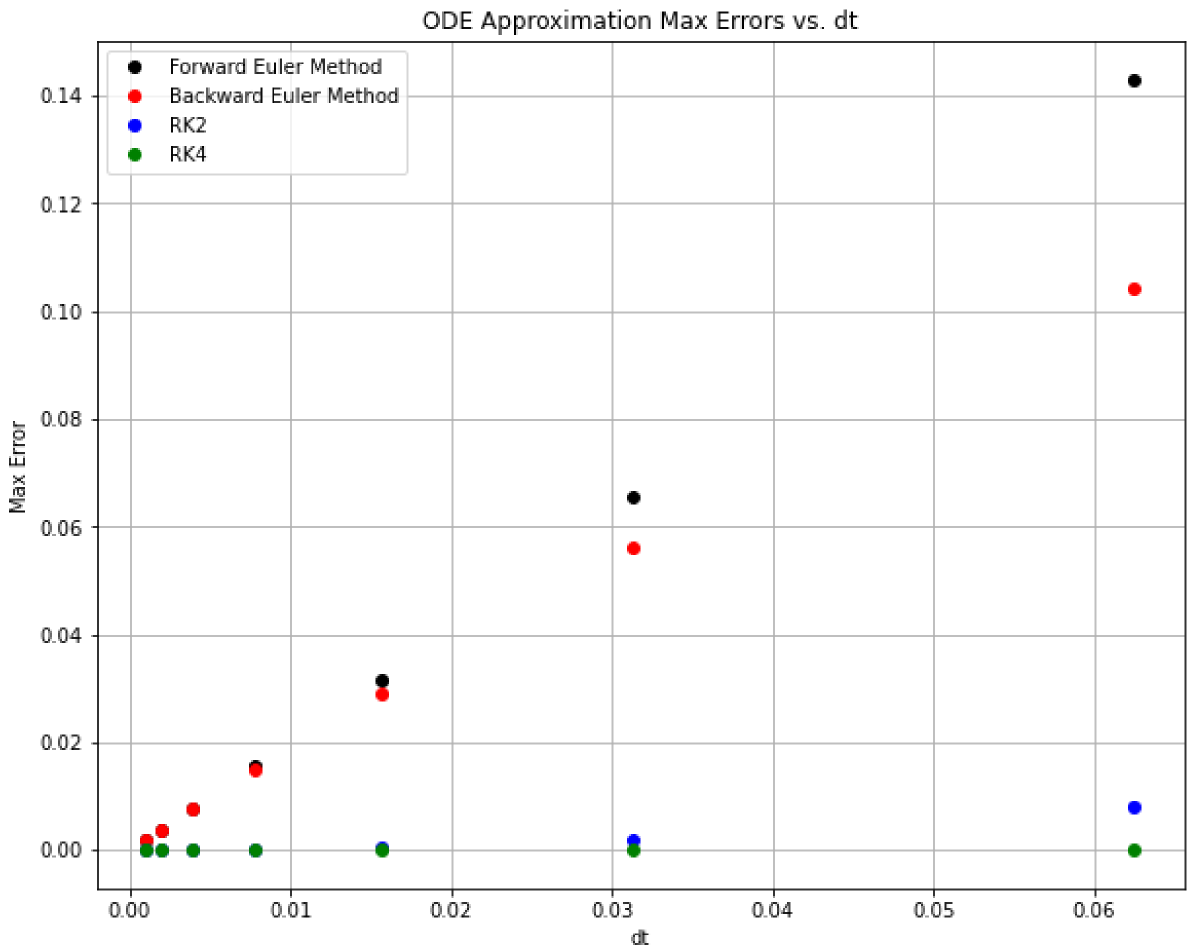
```
    t = np.arange(0, T + dt, dt)
    n = t.size
    V = np.zeros((2, n))
    V[:, 0] = v0
    for k in range(n - 1):
        f1 = f(t[k], V[:, k])
        f2 = f(t[k] + dt / 2, V[:, k] + (dt / 2) * f1)
        f3 = f(t[k] + dt / 2, V[:, k] + (dt / 2) * f2)
        f4 = f(t[k] + dt, V[:, k] + dt * f3)
        V[:, k + 1] = V[:, k] + (dt / 6) * (f1 + 2 * f2 + 2 * f3 + f4)
    theta = V[0, :]
    err = np.max(np.abs(x_true(t) - theta))
    max_err_RK4.append(err)

# d)

# Display results

dts = [2**-dt for dt in range(4, 11)]
data = {'dt': dts}

methods, errs = ['F.E.', 'B.E.', 'RK2', 'RK4'], [max_err_forward, max_err_back
ward, max_err_RK2, max_err_RK4]

for k in range(len(methods)):
    data[methods[k]] = errs[k]

table = pd.DataFrame(data)
display(table)

"""
These errors show us the rate of change of each ODE Solver Method as dt shrink
s exponentially (2^n). Both the Forward and
Backward Euler methods shrink at approximately the same rate, proving that bot
h methods are first order accurate. RK2 shrinks
by double this rate, at 2^2n, proving that it is second order accurate. RK4 sh
rinks by approximately 2^4n, proving that it is
fourth order accurate. I calculated the rates of change between the first and
 second observation for each method to demonstrate
this:

As shown, F.E. and B.E. are approximately equal, RK2 is approximately double,
 and RK4 is approximately four times the factor
of dt.
"""
print("dt factor:", dts[0]/dts[1])
print("F.E. factor:", max_err_forward[0]/max_err_forward[1])
print("B.E. factor:", max_err_backward[0]/max_err_backward[1])
print("RK2 factor:", max_err_RK2[0]/max_err_RK2[1])
print("RK4 factor:", max_err_RK4[0]/max_err_RK4[1])
```

	dt	F.E.	B.E.	RK2	RK4
0	0.062500	0.143069	0.104068	0.007798	1.616373e-05
1	0.031250	0.065689	0.055968	0.001930	9.885757e-07
2	0.015625	0.031505	0.029072	0.000479	6.109503e-08
3	0.007812	0.015430	0.014824	0.000119	3.796622e-09
4	0.003906	0.007638	0.007486	0.000030	2.366148e-10
5	0.001953	0.003800	0.003762	0.000007	1.476708e-11
6	0.000977	0.001895	0.001886	0.000002	9.218737e-13

```
dt factor: 2.0
F.E. factor: 2.1779688608503607
B.E. factor: 1.8594096314063433
RK2 factor: 4.0395139374991205
RK4 factor: 16.35052561077577
```

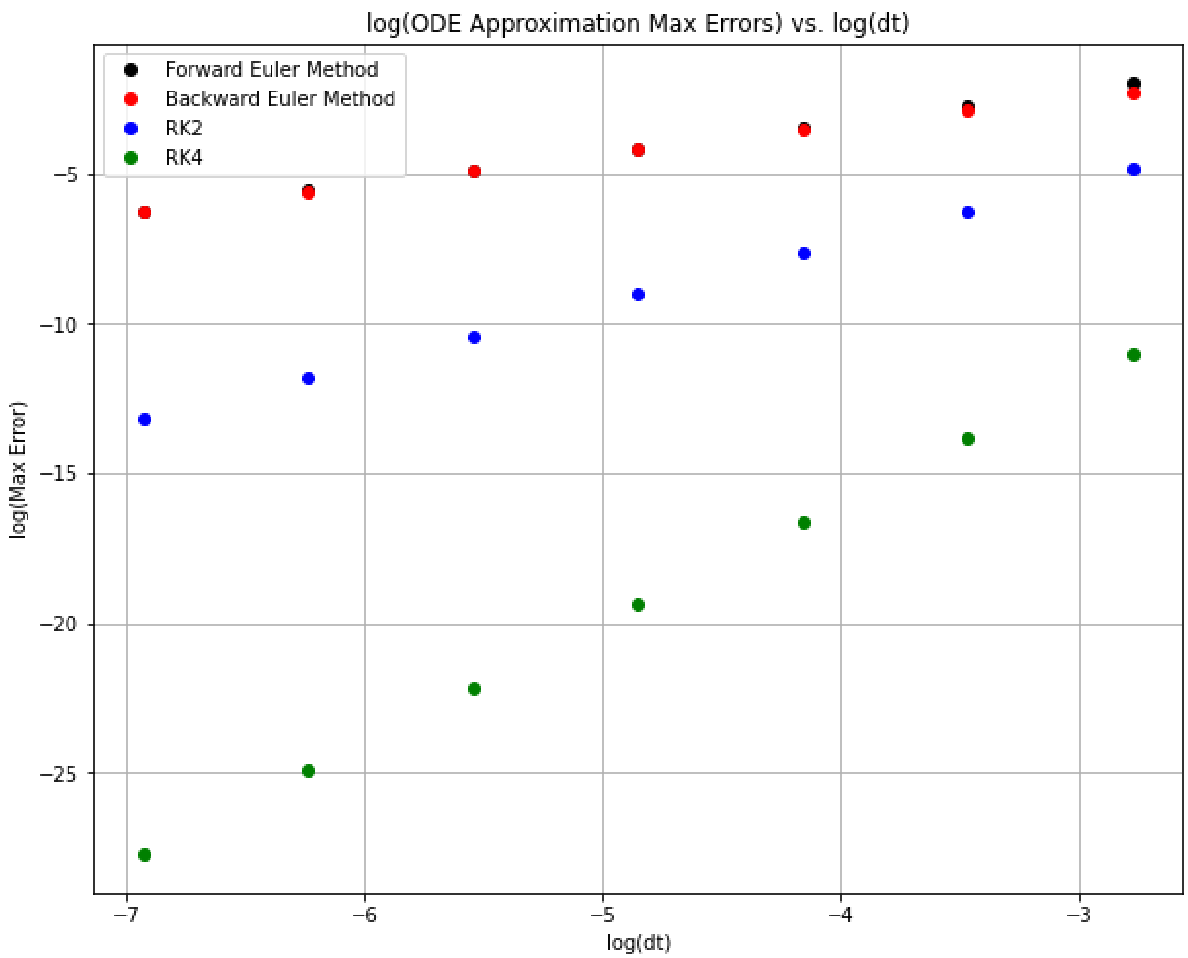
In [118]:
```
# Problem 2

# a)

plt.figure(figsize=(10, 8))
plt.title('ODE Approximation Max Errors vs. dt')
plt.xlabel('dt')
plt.ylabel('Max Error')
plt.grid()
plt.plot(dts, max_err_forward, 'ko', dts, max_err_backward, 'ro', dts, max_err
_RK2, 'bo', dts, max_err_RK4, 'go')
plt.legend(['Forward Euler Method', 'Backward Euler Method', 'RK2', 'RK4'])
```

Out[118]: `<matplotlib.legend.Legend at 0x1e6f66bf548>`



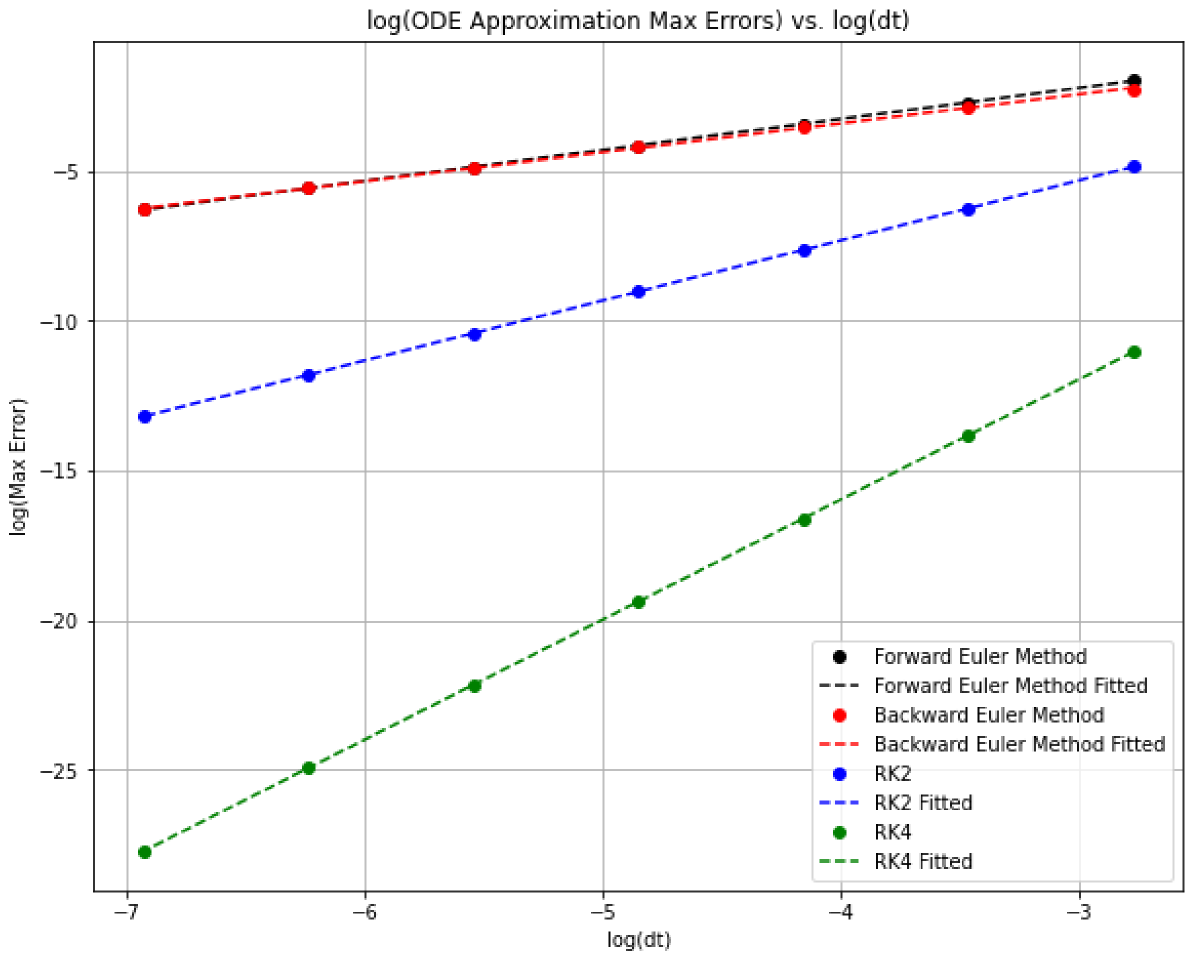
In [119]:
```
# b)

# Log data

log_dts, log_forward, log_backward, log_RK2, log_RK4 = np.log(dts), np.log(max
_err_forward), np.log(max_err_backward), np.log(max_err_RK2), np.log(max_err_R
K4)

plt.figure(figsize=(10, 8))
plt.title('log(ODE Approximation Max Errors) vs. log(dt)')
plt.xlabel('log(dt)')
plt.ylabel('log(Max Error)')
plt.grid()
plt.plot(log_dts, log_forward, 'ko', log_dts, log_backward, 'ro', log_dts, log
_RK2, 'bo', log_dts, log_RK4, 'go')
plt.legend(['Forward Euler Method', 'Backward Euler Method', 'RK2', 'RK4'])
```

Out[119]: <matplotlib.legend.Legend at 0x1e6f6b0b888>



In [120]:
```
# c)

# Linear fits

# Forward Euler
coeffs_forward = np.polyfit(log_dts, log_forward, 1)
forward_hat = np.polyval(coeffs_forward, log_dts)

# Backward Euler
coeffs_backward = np.polyfit(log_dts, log_backward, 1)
backward_hat = np.polyval(coeffs_backward, log_dts)

# RK2
coeffs_RK2 = np.polyfit(log_dts, log_RK2, 1)
RK2_hat = np.polyval(coeffs_RK2, log_dts)

# RK4
coeffs_RK4 = np.polyfit(log_dts, log_RK4, 1)
RK4_hat = np.polyval(coeffs_RK4, log_dts)

# Plot

plt.figure(figsize=(10, 8))
plt.title('log(ODE Approximation Max Errors) vs. log(dt)')
plt.xlabel('log(dt)')
plt.ylabel('log(Max Error)')
plt.grid()
plt.plot(log_dts, log_forward, 'ko', log_dts, forward_hat, 'k--',
         log_dts, log_backward, 'ro', log_dts, backward_hat, 'r--',
         log_dts, log_RK2, 'bo', log_dts, RK2_hat, 'b--',
         log_dts, log_RK4, 'go', log_dts, RK4_hat, 'g--', )
plt.legend(['Forward Euler Method', 'Forward Euler Method Fitted',
            'Backward Euler Method', 'Backward Euler Method Fitted',
            'RK2', 'RK2 Fitted',
            'RK4', 'RK4 Fitted'])
```

Out[120]: <matplotlib.legend.Legend at 0x1e6f6b8ed08>



In [121]:
```
# d)

print('Forward Euler Slope:', coeffs_forward[0])
print('Backward Euler Slope:', coeffs_backward[0])
print('RK2 Slope:', coeffs_RK2[0])
print('RK4 Slope:', coeffs_RK4[0])

"""
These four slopes represent the order of accuracy of each ODE Solver. By plott
ing the max errors for each method over dt, we
can verify their orders of accuracy as dt shrinks. The slopes of the Foward an
d Backward Euler methods are approximately
1.035 and 0.968 respectively. At dt changes by a factor of log(dt), both of th
ese methods' log(errors) are growing at a
proportional rate of about 1, proving that Forward and Backward Euler are firs
t order accurate. The RK2 slope is approximately
2.005, proving that it is second order accurate for the same reason. The RK4 s
lope is approximately 4.009, proving that it is
fourth order accurate.
"""
```

```
Forward Euler Slope: 1.0351025691513667
Backward Euler Slope: 0.9681028710257773
RK2 Slope: 2.0053911381429557
RK4 Slope: 4.009449203409709
```

Out[121]:
```
"\nThese four slopes represent the order of accuracy of each ODE Solver. By p
lotting the max errors for each method over dt, we\ncan verify their orders o
f accuracy as dt shrinks. The slopes of the Foward and Backward Euler methods
are approximately\n1.035 and 0.968 respectively. At dt changes by a factor of
log(dt), both of these methods' log(errors) are growing at a\nproportional ra
te of about 1, proving that Forward and Backward Euler are first order accura
te. The RK2 slope is approximately\n2.005, proving that it is second order ac
curate for the same reason. The RK4 slope is approximately 4.009, proving tha
t it is\nfourth order accurate.\n"
```