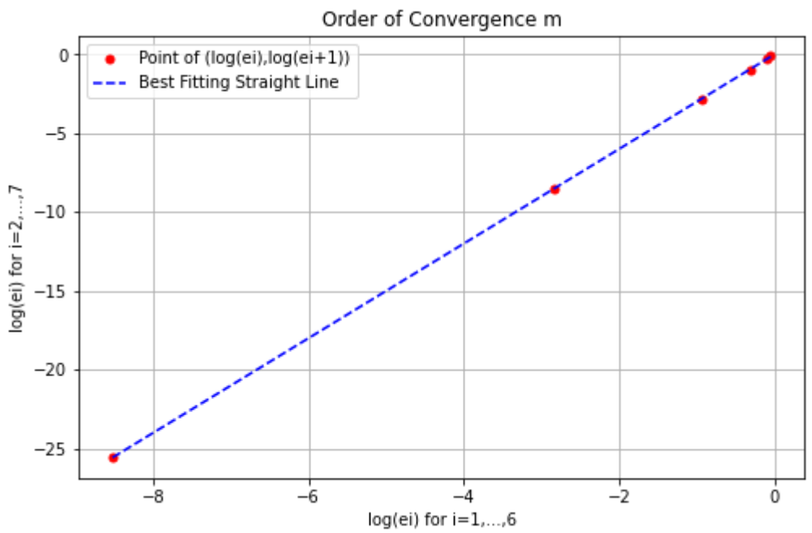
In [1]:
```
%matplotlib inline
import numpy as np
import math
import cmath
import matplotlib.pyplot as plt
```

Lab book 1

In [2]:
```
# a) See hand-written part above
```

In [3]:
```
#b) Using np.polyfit to do the fitting, estimate the order of convergence of the following sequence
#put e1 to e7 into an array
earray = np.array([9.55213728*(10**(-1)),9.06436704*(10**(-1)),7.33394145*(10**(-1)),3.89531478*(10**(-1)),5.83960912*(10**(-2)),1.97897053*(10**(-4)),7.63529928*(10**(-12))])
#set x and y to be log ei
x = np.log(earray[0:6])
y = np.log(earray[1:7])
# find the degree 1 (straight line) fitting line to x and y
slope, intercept = np.polyfit(x,y,1)
```

In [4]:
```
plt.figure(figsize=(8,5))
plt.clf()
plt.plot(x, y, 'r.', markersize=10, label='Point of (log(ei),log(ei+1))')
plt.plot(x,x*slope+intercept,'b--', label='Best Fitting Straight Line')
plt.xlabel('log(ei) for i=1,...,6')#display label of x-axis
plt.ylabel('log(ei) for i=2,...,7')#display label of y-axis
plt.title('Order of Convergence m')#display title
plt.legend(loc='best') # display legend
plt.grid()
plt.show()
print('slope of the degree 1 polynomial fitting is',slope)
```




```
slope of the degree 1 polynomial fitting is 3.0020757470700277
```

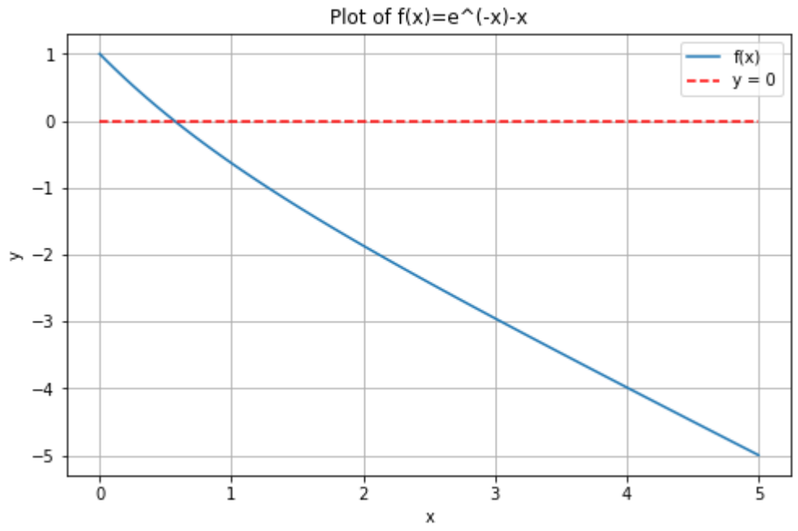
The order of convergence is the slope of the straight line, which is approximately order of 3.

Lab Book 2

Make a plot of the function f(x) = e**(-x)-x, and use this to decide on a sensible choice of starting points. Run Newton's method and the secant method for 10 iterations to find a root of f(x). Estimate the order of convergence (e.g. using f(xn) -> 0) and comment all the results you have obtained.

In [5]:
```
#plot of f(x)
x_plot = np.linspace(0,5,100)
y = np.exp(-x_plot)-x_plot

plt.figure(figsize=(8,5))
plt.clf()
plt.plot(x_plot,y,markersize=10, label='f(x)')
plt.plot([0,5],[0,0],'r--',markersize=10,label='y = 0')#the x-axis
plt.xlabel('x')#display label of x-axis
plt.ylabel('y')#display label of y-axis
plt.title('Plot of f(x)=e^(-x)-x')#display title
plt.legend(loc='best')
plt.grid()
plt.show()
```



A proper starting point would be x0 = 1.0 as the root is between 0 and 1 as shown in the plot of f(x) above.

In [6]:
```
#Newton's method
def newton(f, df, x0, niters):
    """
    Newton's method for 1D rootfinding.
    - The function f(x) is the one we want the root of
    - The function df(x) is the derivative f'(x)
    - x0 is the starting point
    - niters is the number of iterations to run
    """
    x = x0 # initial guess
    print("{0:^3}{1:^25}{2:^25}".format("k", "xk", "f(xk)"))
    for i in range(niters):
        print("{0:^3}{1:^25.15e}{2:^25.15e}".format(i, x, f(x)))
        x = x - f(x)/df(x)
    return x
```

In [7]:
```
def f(x):
    f = np.exp(-x)-x
    return f
def df(x):
    df = -np.exp(-x)-1
    return df
newton(f,df,1.0,10)
```

```
k            xk                     f(xk)
0    1.000000000000000e+00   -6.321205588285577e-01
1    5.378828427399902e-01    4.610048629168972e-02
2    5.669869914054133e-01    2.449498638371628e-04
3    5.671432859891230e-01    6.927808993140161e-09
4    5.671432904097838e-01    0.000000000000000e+00
5    5.671432904097838e-01    0.000000000000000e+00
6    5.671432904097838e-01    0.000000000000000e+00
7    5.671432904097838e-01    0.000000000000000e+00
8    5.671432904097838e-01    0.000000000000000e+00
9    5.671432904097838e-01    0.000000000000000e+00
```

Out[7]:
```
0.5671432904097838
```

As shown above, the root of f(x) using Newtons method is: 0.5671432904097838 . The order of convergence of Newton's method is about 2 as the number of accurate decimal places of f(xk) approximately doubled for each iteration. Python stores numbers to around 16 digits after the decimal point. The accurate result to the 16th decimal place is being found at around 5th iternation (k=4).

In [8]:
```
#Secant Method
def secant(f, x1, x2, niters):
    """
    Secant method for 1D rootfinding.
    - The function f(x) is the one we want the root of
    - x1 and x2 are the two starting points
    - niters is the number of iterations to run
    """
    f1 = f(x1); f2 = f(x2)
    print("{0:^3}{1:^25}{2:^25}".format("k", "xk", "f(xk)"))
    for i in range(niters):
        print("{0:^3}{1:^25.15e}{2:^25.15e}".format(i, x2, f2))
        if f1 == f2:
            print('Secant method error: division by zero')
            return x2
        x3 = x2 - f2*(x2 - x1)/(f2-f1)
        # Update x1 and x2 (don't need to modify this)
        x1 = x2; f1 = f2
        x2 = x3; f2 = f(x3)
    return x2
```

In [9]:
```
def f(x):
    f = np.exp(-x)-x
    return f
#the starting points are set to be at x1=0.0 and x2=0.1 for approaching the root between 0.0 and 1.0
secant(f,0.0,0.1,10)
```

```
k            xk                     f(xk)
0    1.000000000000000e-01    8.048374180359595e-01
1    5.123933030278593e-01    8.666682631537515e-02
2    5.621597779840559e-01    7.816932388359010e-03
3    5.670934709868598e-01    7.807487819688763e-05
4    5.671432454503060e-01    7.045794458981902e-08
5    5.671432904093786e-01    6.351585923880521e-13
6    5.671432904097840e-01   -1.110223024625157e-16
7    5.671432904097838e-01    0.000000000000000e+00
```

```
 8    5.671432904097838e-01    0.000000000000000e+00
Secant method error: division by zero
0.5671432904097838
```

Out[9]:

As shown above, the root of f(x) using Secant method is: 0.5671432904097838. The order of convergence of Secant method is appearently less than that of Newton's method but greater than linear convergence. The order of convergence of Secant method is around 1.62. The accurate result to the 16th decimal place is being found at around 8th iternation (k=7).The iteration stops at k=8, the ninth iteration, because of the lines:

```
"if f1 == f2:
    print('Secant method error: division by zero')
    return x2"
```

which stops the iteration as the xk after k=8 will be the same.

Lab Book 3

Run Newton's method to find a root of f(x) = x^4 starting from x = 1. Estimate the order of convergence and comment on the results.

In [10]:

```
def f(x):
    f = x**4
    return f
def df(x):
    df = 4*x**3
    return df
newton(f,df,1.0,20)
```

```
k             xk                      f(xk)
 0    1.000000000000000e+00    1.000000000000000e+00
 1    7.500000000000000e-01    3.164062500000000e-01
 2    5.625000000000000e-01    1.001129150390625e-01
 3    4.218750000000000e-01    3.167635202407837e-02
 4    3.164062500000000e-01    1.002259575761855e-02
 5    2.373046875000000e-01    3.171211938933993e-03
 6    1.779785156250000e-01    1.003391277553334e-03
 7    1.334838867187500e-01    3.174792714133595e-04
 8    1.001129150390625e-01    1.004524257206333e-04
 9    7.508468627929688e-02    3.178377532566913e-05
10    5.631351470947266e-02    1.005658516163750e-05
11    4.223513603210449e-02    3.181966398799364e-06
12    3.167635202407837e-02    1.006794055870111e-06
13    2.375726401805878e-02    3.185559317401524e-07
14    1.781794801354408e-02    1.007930877771576e-07
15    1.336346101015806e-02    3.189156292949127e-08
16    1.002259575761855e-02    1.009068983315935e-08
17    7.516946818213910e-03    3.192757330023075e-09
18    5.637710113660432e-03    1.010208373952613e-09
19    4.228282585245324e-03    3.196362433209441e-10
0.0031712119389339932
```

Out[10]:

The root of f(x) = x^4 should be x = 0. However, the Newton's method converge slowly to find the root. The order of convergence is clearly less than linear convergence. It took around 2 iterations to get one more digit of accuracy of f(xk). This could be explained by the almost flat f(x) around 0, the slope of the function f'(x) does not vary much for each iteration.

Lab Book 4

Brent's method does not require the derivative f'(x). For f(x) = e**(-x)-x, how does it compare to the secant method? Hint: you may want to look at soln.iterations and soln.function_calls. Check the documentation to see what these are.

In [11]:
```
import scipy.optimize as optimize

def f(x):
    f = np.exp(-x)-x
    return f
```

In [12]:
```
soln = optimize.root_scalar(f, bracket=(-1, 1), method='brentq')

print("Root is x =", soln.root,',which takes',soln.iterations, 'number of iterations','and the number of function calls is',soln.function_calls)
```

```
Root is x = 0.567143290409784 ,which takes 7 number of iterations and the number of function calls is 8
```

In [13]:
```
secant(f,0.0,0.1,10)
```

```
k            xk                     f(xk)
0    1.000000000000000e-01    8.048374180359595e-01
1    5.123933030278593e-01    8.666682631537515e-02
2    5.621597779840559e-01    7.816932388359010e-03
3    5.670934709868598e-01    7.807487819688763e-05
4    5.671432454503060e-01    7.045794458981902e-08
5    5.671432904093786e-01    6.351585923880521e-13
6    5.671432904097840e-01   -1.110223024625157e-16
7    5.671432904097838e-01    0.000000000000000e+00
8    5.671432904097838e-01    0.000000000000000e+00
Secant method error: division by zero
```

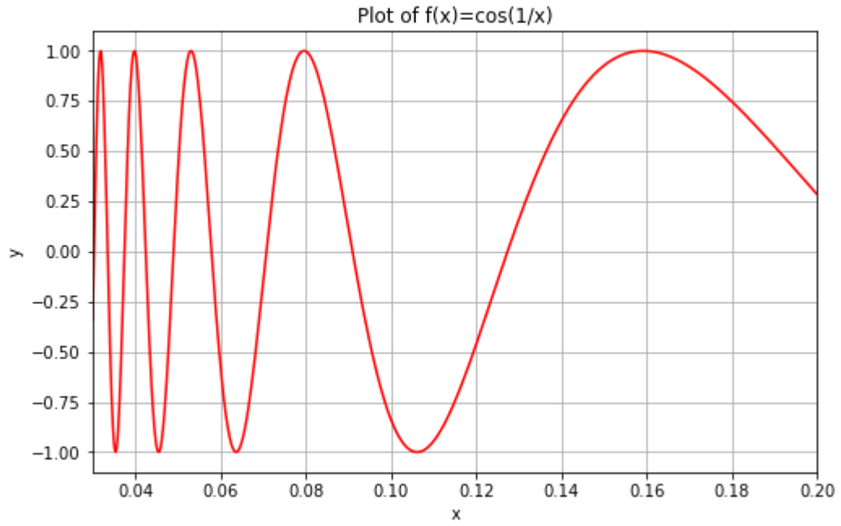
Out[13]:
```
0.5671432904097838
```

Brent's method is the combined use of Secant method and the Bisection method. Since Secant method may not work if the starting point is far from the root, Brent's method ensure the fast and robust of the root finding. Compare with the Secant method, Brent's method has a very similar order of convergence, at around 1 to 1.62 (according to the documentation),which is the order of convergence of Bisection method to that of the Secant method . It takes 7 iterations to find the root for the Brent's method (faster than the Secant method), and the Secant method takes 8 iterations to do the same job.

Lab Book 5

Try running Newton's method for this f(x) with starting points x0 = 0.05; 0.06; ... ; 0.2. What do you observe?

In [14]:
```
#Plot of f(x)=cos(1/x)
x = np.linspace(0.03,0.2,1000)
y = np.cos(1/x)
plt.figure(figsize=(8,5))
plt.clf()
plt.plot(x, y, 'r', markersize=10)
plt.xlabel('x')#display label of x-axis
plt.ylabel('y')#display label of y-axis
plt.title('Plot of f(x)=cos(1/x)')#display title
plt.xlim(0.03,0.2)
plt.grid()
plt.show()
```



In [15]:

```
def f(x):
    f = np.cos(1/x)
    return f
def df(x):
    df = np.sin(1/x)*(x**(-2))
    return df
#run Newton's method for f(x) with different starting points from 0.05 to 0.20 each for 10 iterations
for i in np.linspace(0.05,0.20,16):
    newton(f,df,i,10)
```

```
k            xk                  f(xk)
0    5.000000000000000e-02    4.080820618133920e-01
1    4.888251222762771e-02   -3.685312839673568e-02
2    4.897063263882934e-02   -4.965611485208414e-05
3    4.897075172029371e-02   -1.207079802873458e-10
4    4.897075172058318e-02    2.572377258846030e-15
5    4.897075172058318e-02   -9.803364199544708e-16
6    4.897075172058318e-02   -9.803364199544708e-16
7    4.897075172058318e-02   -9.803364199544708e-16
8    4.897075172058318e-02   -9.803364199544708e-16
9    4.897075172058318e-02   -9.803364199544708e-16
k            xk                  f(xk)
0    6.000000000000000e-02   -5.745816685191216e-01
1    5.747266057930106e-02    1.205239853450969e-01
2    5.787368748402621e-02    2.499771943416221e-04
3    5.787452474859368e-02    3.611188896040978e-09
4    5.787452476068921e-02    1.102801099869206e-15
5    5.787452476068922e-02   -2.449912578931295e-15
6    5.787452476068921e-02    1.102801099869206e-15
7    5.787452476068922e-02   -2.449912578931295e-15
8    5.787452476068921e-02    1.102801099869206e-15
9    5.787452476068922e-02   -2.449912578931295e-15
k            xk                  f(xk)
0    7.000000000000001e-02   -1.480016316209667e-01
1    7.073328356121376e-02   -4.490389314010012e-04
2    7.073553019185586e-02   -1.423174234418545e-08
3    7.073553026306459e-02   -1.225265779783942e-15
4    7.073553026306459e-02   -1.225265779783942e-15
5    7.073553026306459e-02   -1.225265779783942e-15
6    7.073553026306459e-02   -1.225265779783942e-15
7    7.073553026306459e-02   -1.225265779783942e-15
```

8	7.073553026306459e-02	-1.225265779783942e-15
9	7.073553026306459e-02	-1.225265779783942e-15

k	xk	f(xk)
0	8.000000000000002e-02	9.977982791785805e-01
1	1.762865846995726e-01	8.193019049584476e-01
2	2.206937773295049e-01	-1.802332858539768e-01
3	2.117692318526102e-01	9.732164014727695e-03
4	2.122057031707007e-01	1.971108006535980e-05
5	2.122065907854812e-01	8.244408993322455e-11
6	2.122065907891938e-01	7.044813998280222e-16
7	2.122065907891938e-01	-1.836970198721030e-16
8	2.122065907891938e-01	-1.836970198721030e-16
9	2.122065907891938e-01	-1.836970198721030e-16

k	xk	f(xk)
0	9.000000000000001e-02	1.152799495457487e-01
1	9.094003476317759e-02	6.827800664496918e-04
2	9.094568141704078e-02	4.228649565388450e-08
3	9.094568176679733e-02	-4.286263797015736e-16
4	9.094568176679733e-02	-4.286263797015736e-16
5	9.094568176679733e-02	-4.286263797015736e-16
6	9.094568176679733e-02	-4.286263797015736e-16
7	9.094568176679733e-02	-4.286263797015736e-16
8	9.094568176679733e-02	-4.286263797015736e-16
9	9.094568176679733e-02	-4.286263797015736e-16

k	xk	f(xk)
0	1.000000000000000e-01	-8.390715290764524e-01
1	8.457648954643079e-02	7.366088757789270e-01
2	9.236733666015086e-02	-1.684296694806499e-01
3	9.090951412981259e-02	4.374489014866165e-03
4	9.094566761412363e-02	1.711096076843139e-06
5	9.094568176679513e-02	2.660248995303360e-13
6	9.094568176679733e-02	-4.286263797015736e-16
7	9.094568176679733e-02	-4.286263797015736e-16
8	9.094568176679733e-02	-4.286263797015736e-16
9	9.094568176679733e-02	-4.286263797015736e-16

k	xk	f(xk)
0	1.100000000000000e-01	-9.447815861050266e-01
1	1.448850548210956e-01	8.145531709443110e-01
2	1.154088162860026e-01	-7.248851499960248e-01
3	1.294243637965088e-01	1.271162852806211e-01
4	1.272776669774745e-01	-2.856279959490525e-03
5	1.273239377719383e-01	-1.030237430170799e-06
6	1.273239544735141e-01	-1.346969580946322e-13
7	1.273239544735163e-01	3.061616997868383e-16
8	1.273239544735163e-01	3.061616997868383e-16
9	1.273239544735163e-01	3.061616997868383e-16

k	xk	f(xk)
0	1.200000000000000e-01	-4.612040391631877e-01
1	1.274849343677160e-01	9.917347589756636e-03
2	1.273237456565879e-01	-1.288089909246382e-05
3	1.273239544731738e-01	-2.112590572928739e-11
4	1.273239544735163e-01	3.061616997868383e-16
5	1.273239544735163e-01	3.061616997868383e-16
6	1.273239544735163e-01	3.061616997868383e-16
7	1.273239544735163e-01	3.061616997868383e-16
8	1.273239544735163e-01	3.061616997868383e-16
9	1.273239544735163e-01	3.061616997868383e-16

k	xk	f(xk)
0	1.300000000000000e-01	1.609705435155296e-01
1	1.272436528732162e-01	-4.956512150711360e-03
2	1.273239044855045e-01	-3.083513090773408e-06
3	1.273239544734966e-01	-1.210281024351484e-12
4	1.273239544735163e-01	3.061616997868383e-16
5	1.273239544735163e-01	3.061616997868383e-16

6	1.273239544735163e-01	3.061616997868383e-16
7	1.273239544735163e-01	3.061616997868383e-16
8	1.273239544735163e-01	3.061616997868383e-16
9	1.273239544735163e-01	3.061616997868383e-16

k	xk	f(xk)
0	1.400000000000000e-01	6.526861299196702e-01
1	1.231148781031472e-01	-2.652985102973746e-01
2	1.272855298906745e-01	-2.370934743732119e-03
3	1.273239429494940e-01	-7.108596943574465e-07
4	1.273239544735152e-01	-6.453086293832230e-14
5	1.273239544735163e-01	3.061616997868383e-16
6	1.273239544735163e-01	3.061616997868383e-16
7	1.273239544735163e-01	3.061616997868383e-16
8	1.273239544735163e-01	3.061616997868383e-16
9	1.273239544735163e-01	3.061616997868383e-16

k	xk	f(xk)
0	1.500000000000000e-01	9.273677030509756e-01
1	9.423171270400196e-02	-3.741088334939611e-01
2	9.064965848265231e-02	3.589916030087359e-02
3	9.094484509962890e-02	1.011562115345326e-04
4	9.094568175910316e-02	9.302474501012617e-10
5	9.094568176679733e-02	-4.286263797015736e-16
6	9.094568176679733e-02	-4.286263797015736e-16
7	9.094568176679733e-02	-4.286263797015736e-16
8	9.094568176679733e-02	-4.286263797015736e-16
9	9.094568176679733e-02	-4.286263797015736e-16

k	xk	f(xk)
0	1.600000000000000e-01	9.994494182244994e-01
1	9.311425334553461e-01	4.766560450054381e-01
2	4.610274824622496e-01	-5.632152540166064e-01
3	6.058996357053638e-01	-7.955786517090414e-02
4	6.351993439086883e-01	-3.512596854610388e-03
5	6.366166089311246e-01	-7.805505380346584e-06
6	6.366197723518620e-01	-3.878602422214973e-11
7	6.366197723675814e-01	6.123233995736766e-17
8	6.366197723675814e-01	6.123233995736766e-17
9	6.366197723675814e-01	6.123233995736766e-17

k	xk	f(xk)
0	1.700000000000000e-01	9.207365363804035e-01
1	2.381966117596714e-01	-4.918179531934897e-01
2	2.061480928796590e-01	1.380503695470443e-01
3	2.120715401989906e-01	3.000921091370821e-03
4	2.122065052466696e-01	1.899610229820227e-06
5	2.122065907891593e-01	7.654261007616358e-13
6	2.122065907891938e-01	7.044813998280222e-16
7	2.122065907891938e-01	-1.836970198721030e-16
8	2.122065907891938e-01	-1.836970198721030e-16
9	2.122065907891938e-01	-1.836970198721030e-16

k	xk	f(xk)
0	1.800000000000000e-01	7.467529543114472e-01
1	2.163775982685309e-01	-9.071359267696941e-02
2	2.121128711573614e-01	2.082113352106412e-03
3	2.122065495339123e-01	9.161401209150377e-07
4	2.122065907891857e-01	1.783401653398531e-13
5	2.122065907891938e-01	7.044813998280222e-16
6	2.122065907891938e-01	-1.836970198721030e-16
7	2.122065907891938e-01	-1.836970198721030e-16
8	2.122065907891938e-01	-1.836970198721030e-16
9	2.122065907891938e-01	-1.836970198721030e-16

k	xk	f(xk)
0	1.900000000000000e-01	5.233425926926495e-01
1	2.121713081331028e-01	7.836383920515344e-04
2	2.122065849301228e-01	1.301101065049216e-07
3	2.122065907891936e-01	4.257195078628523e-15

```
4   2.122065907891938e-01   -1.836970198721030e-16
5   2.122065907891938e-01   -1.836970198721030e-16
6   2.122065907891938e-01   -1.836970198721030e-16
7   2.122065907891938e-01   -1.836970198721030e-16
8   2.122065907891938e-01   -1.836970198721030e-16
9   2.122065907891938e-01   -1.836970198721030e-16
k            xk                      f(xk)
0   2.000000000000000e-01    2.836621854632262e-01
1   2.118325166213098e-01    8.321492231970928e-03
2   2.122059399973655e-01    1.445192457814206e-05
3   2.122065907871980e-01    4.431991944601637e-11
4   2.122065907891938e-01   -1.836970198721030e-16
5   2.122065907891938e-01   -1.836970198721030e-16
6   2.122065907891938e-01   -1.836970198721030e-16
7   2.122065907891938e-01   -1.836970198721030e-16
8   2.122065907891938e-01   -1.836970198721030e-16
9   2.122065907891938e-01   -1.836970198721030e-16
```

From above, the roots found by using different starting points are different. Since f(x) has multiple roots in [0.05,0.20], the root found from each starting point is the closest root to the starting point. Some starting points converge to the same root after 10 iterations, but some converge to different roots.

Lab Book 6

Implement the multidimensional version of Newton's method from lectures and use it to solve the above system of equations starting from x0 = (0.1,-1). Do you observe a quadratic local convergence rate? Hint: to measure the error, I suggest using sqrt(f1(xn)2 + f2(xn)2), where fi are the two functions you wish to make zero.

In [16]:
```
#Use scipy.optimize to fine the root first
def func(x):
    f = [x[0]+x[1]-x[0]*x[1]+2,
         x[0]*np.exp(-x[1])-1]
    return f

soln1 = optimize.root(func,[0.1,-1],method = 'broyden1')
soln2 = optimize.root(func,[0.1,-1],method = 'broyden2')
```

In [17]:
```
print(soln1)
print(soln2)
```

```
    fun: array([1.90596161e-08, 2.70308389e-08])
message: 'A solution was found at the specified tolerance.'
    nit: 13
 status: 1
success: True
      x: array([5.38656078, 1.68390708])
    fun: array([-3.22088949e-07, -4.98903796e-07])
message: 'A solution was found at the specified tolerance.'
    nit: 20
 status: 1
success: True
      x: array([ 0.09777303, -2.32510601])
```

There exist two roots for the nonlinear system of equations.

In [18]:
```
#Newton's method - Multidimensional
def newton_multi(func,J,x0,niters):
    """
    - The function func(x) is the one we want the root of
    - The function J(x) is the Jacobian matrix of func(x)
    - x0 is the starting point
```

```
    - niters is the number of iterations to run
    """
    x = np.array(x0) # initial guess
    print("{0:^3}{1:^10}{2:^15}".format("k","xk","error"))
    for i in range(niters):
        x = x - np.linalg.inv(J(x))@func(x)
        er = error(x)
        print(i,x,error(x))
        #print(error(x))
    return x
```

In [19]:

```
def func(x):
    f = np.zeros([2,1])
    f[0,0] = x[0]+x[1]-x[0]*x[1]+2.0
    f[1,0] = x[0]*np.exp(-x[1])-1.0
    return f

def J(x):
    J = np.zeros([2,2])
    J[0,0] = 1.0 - x[1]
    J[0,1] = 1.0 - x[0]
    J[1,0] = np.exp(-x[1])
    J[1,1] = -x[0]*np.exp(-x[1])
    return J

def error(x):
    error = 0
    error = ((func(x)[0])**2+(func(x)[1])**2)**(1/2)
    return error

x0 = [[0.1],
      [-1.0]] #initial values of x
newton_multi(func,J,x0,10) #run the multidimensional newton's method
```

Out[19]:

```
 k     xk          error
0 [[ 0.21008318]
 [-2.57796262]] [1.77537228]
1 [[ 0.11766933]
 [-2.37927567]] [0.27110758]
2 [[ 0.09871225]
 [-2.32748129]] [0.01204664]
3 [[ 0.09777511]
 [-2.32511085]] [2.56846498e-05]
4 [[ 0.09777309]
 [-2.32510588]] [1.15219463e-10]
5 [[ 0.09777309]
 [-2.32510588]] [0.]
6 [[ 0.09777309]
 [-2.32510588]] [0.]
7 [[ 0.09777309]
 [-2.32510588]] [0.]
8 [[ 0.09777309]
 [-2.32510588]] [0.]
9 [[ 0.09777309]
 [-2.32510588]] [0.]
array([[ 0.09777309],
       [-2.32510588]])
```

The solution found by the multidimensional Newton's method is at (0.09777309,-2.32510588), which is the same as the solution 2 found by the scipy.optimize at the beginning. The multidimensional Newton's method shows a quadratic convergence rate shown in the change of error. for each of the iteration, the number of accurate digits doubled as shown above. at the fifth iteration, the accuracy of the solution is more than 16 digits after the decimal point, which the error shows 0.

Lab Book 7

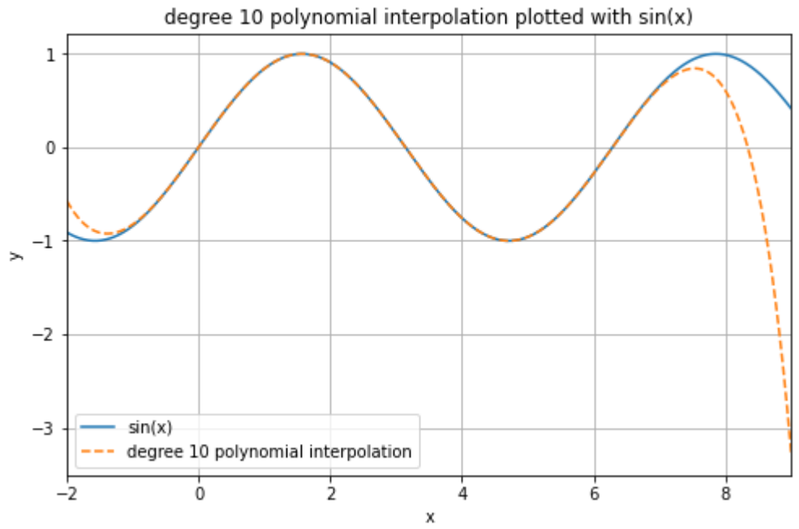
(a) Implement a function polyfit which takes in a vector of x values and y-values and returns the corresponding interpolating polynomial as a NumPy polynomial object (i.e. you should have return np.poly1d(a) as the last line of the function). You should do this by solving the corresponding Vandermonde system. (b) Use your function to find the degree-10 polynomial interpolating sin(x) at x = 0; 0.6; 1.2;...; 6. (c) Make a plot of your interpolant and sin(x) over x in [-2,9] and comment on the quality of the fit.

In [20]:
```
#a)
def polyfit(x,y):
    A = np.zeros([len(x),len(y)])
    for i in range(len(x)):
        for j in range(len(x)):
            A[i,j] = (x[i])**j #Vandermonde matrix
    b = y
    a = np.linalg.solve(A,b)
    #reverse the order of a
    a_reverse = list(reversed(a))
    return np.poly1d(a_reverse)
```

In [21]:
```
#b)
x = np.linspace(0.0,6.0,11)
y = np.sin(x)
polyfit(x,y)
```

Out[21]:
```
poly1d([-3.34568332e-08, -1.27253778e-06,  5.13729692e-05, -5.18769514e-04,
        1.20748275e-03,  5.43100710e-03,  4.44155295e-03, -1.70805212e-01,
        2.10279698e-03,  9.99564646e-01,  0.00000000e+00])
```

In [22]:
```
x_plot = np.linspace(-2,9,1000)
plt.figure(figsize=([8,5]))
plt.clf()
plt.plot(x_plot,np.sin(x_plot),label='sin(x)')
plt.plot(x_plot,polyfit(x,y)(x_plot),'--',label='degree 10 polynomial interpolation')
plt.xlim(-2,9)
plt.title('degree 10 polynomial interpolation plotted with sin(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid()
plt.show()
```



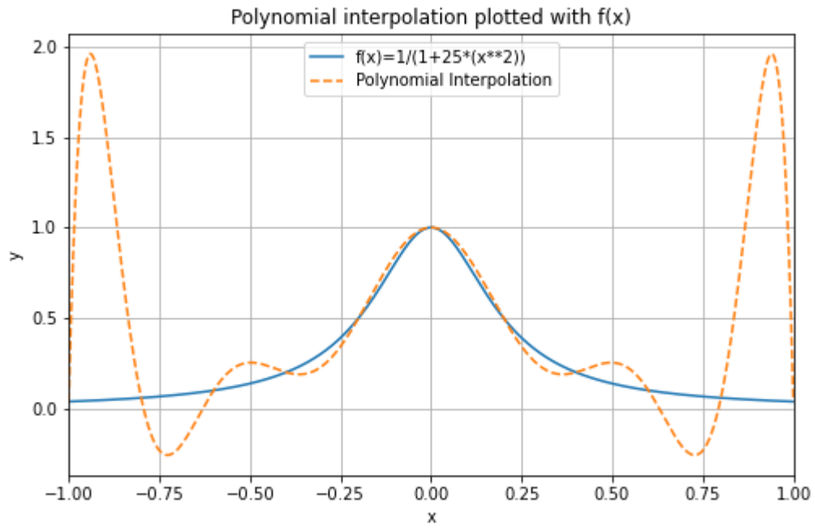
The interpolation is done for x in [0,6]. As shown above, the interpolation is relatively good for x in [0,6], the interpolation and the original function sin(x) are almost the same. However, as shown in the plot, there exist an increase in gap between the interpolation and sin(x) going away from the range of interpolation. Specifically, there exist a huge difference between the interpolation and sin(x) at x = 9. The interpolation is no longer fitting the function outside of [0,6].

Lab Book 8

Using 11 equally spaced nodes in [-1,1], find the polynomial interpolant for Runge's function f(x) = 1/(1+25*x^2) Make a graph of the true f(x) and the interpolant p(x) and comment on what you see.

In [23]:
```
import scipy.interpolate as interpolate
x = np.linspace(-1.0,1.0,11)
fx = 1/(1+25*(x**2))
p = interpolate.BarycentricInterpolator(x,fx)
```

In [24]:
```
x_plot = np.linspace(-1,1,1000)
plt.figure(figsize=([8,5]))
plt.clf()
plt.plot(x_plot,1/(1+25*(x_plot**2)),label='f(x)=1/(1+25*(x**2))')
plt.plot(x_plot,p(x_plot),'--',label='Polynomial Interpolation')
plt.xlim(-1,1)
plt.title('Polynomial interpolation plotted with f(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='upper center')
plt.grid()
plt.show()
```



The polynomial interpolation is relatively accurate close to 0, but becomes very apart from the true function near +/- 1. The interpolation is not very suitable for this f(x) as shown in the plot above.

Lab Book 9

Repeat the above experiment using Chebyshev points (i.e. interpolate Runge's function at n + 1 Chebyshev points for n = 10). Plot the true f(x) and your new interpolant p(x) and compare to your previous result.

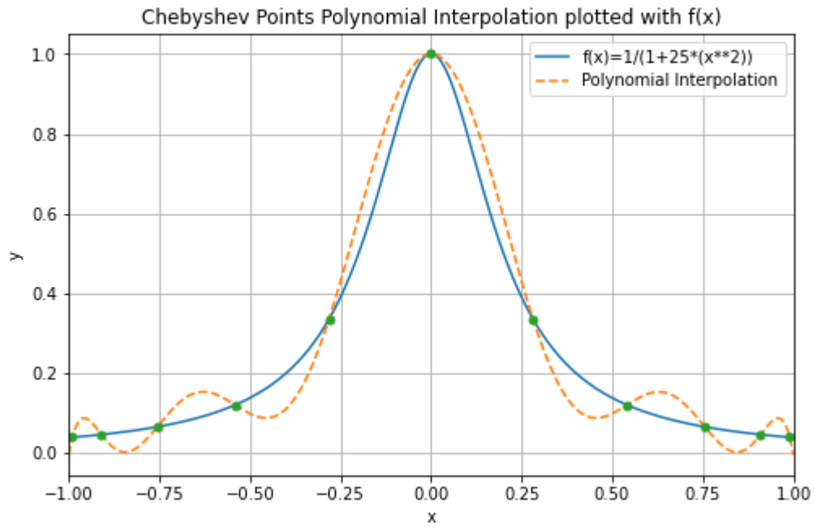
In [25]:
```
def chebyshev_point(n):
    x = np.zeros(n+1)
    for i in range(n+1):
        x[i] = np.cos((2*i+1)*np.pi/(2*n+2))
    return x
```

In [26]:
```
# Chebyshev points for n = 10
print(chebyshev_point(10))
```

```
[ 9.89821442e-01  9.09631995e-01  7.55749574e-01  5.40640817e-01
  2.81732557e-01  2.83276945e-16 -2.81732557e-01 -5.40640817e-01
 -7.55749574e-01 -9.09631995e-01 -9.89821442e-01]
```

In [27]:
```
x = chebyshev_point(10)
fx = 1/(1+25*(x**2))
p = interpolate.BarycentricInterpolator(x,fx)

x_plot = np.linspace(-1,1,1000)
plt.figure(figsize=([8,5]))
plt.clf()
plt.plot(x_plot,1/(1+25*(x_plot**2)),label='f(x)=1/(1+25*(x**2))')
plt.plot(x_plot,p(x_plot),'--',label='Polynomial Interpolation')
plt.plot(x,fx,'.',markersize=10)
plt.xlim(-1,1)
plt.title('Chebyshev Points Polynomial Interpolation plotted with f(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid()
plt.show()
```



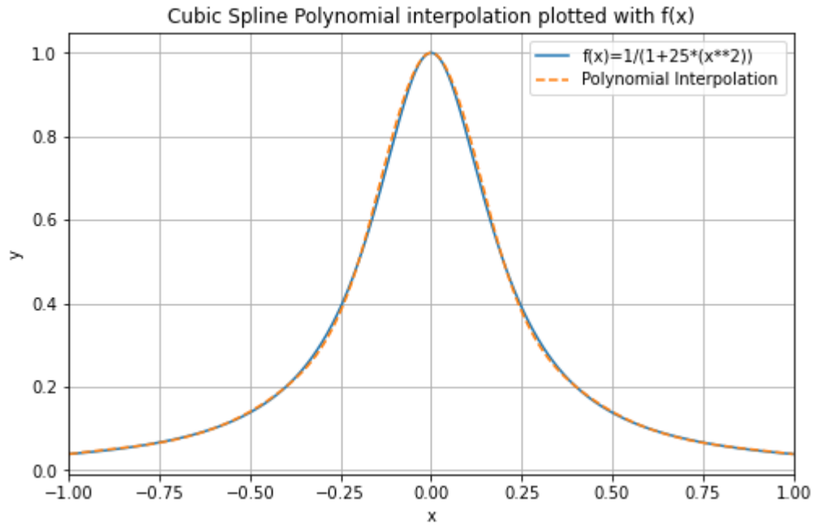
Compare this chebyshev point polynomial interpolation with the equally spaced polynomial interpolation, the chebyshev point interpolation is more precise than the equally spaced interpolation. The polynomial is having some oscillation around f(x), but is generally following the f(x). The error is reduced at the end points a lot to a resonable value. Overall, the cubic splien interpolation is much more accurate than the equally spaced polynomial interpolation.

Lab Book 10

Repeat the above experiment using 11 equally spaced points in [-1, 1] and a cubic spline with “not-a-knot” extra conditions. Plot f(x) and your new interpolant and compare to your result for polynomial interpolation with equally spaced points.

In [28]:
```
x = np.linspace(-1.0,1.0,11)
fx = 1/(1+25*(x**2))
sp = interpolate.CubicSpline(x,fx,bc_type='not-a-knot')
```

In [29]:
```
x_plot = np.linspace(-1,1,1000)
plt.figure(figsize=([8,5]))
plt.clf()
plt.plot(x_plot,1/(1+25*(x_plot**2)),label='f(x)=1/(1+25*(x**2))')
plt.plot(x_plot,sp(x_plot),'--',label='Polynomial Interpolation')
plt.xlim(-1,1)
plt.title('Cubic Spline Polynomial interpolation plotted with f(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid()
plt.show()
```



Compare this cubic spline interpolation with the equally spaced polynomial interpolation, the cubic spline interpolation is much more precise than the equally spaced interpolation. it is clear to see from the above plot that the polynomial and the f(x) are almost the same. The fitting is quite accurate at x = -1, 0, 1, with very small divergence from f(x) at at around x = -0.25 and +0.25. Overall, the cubic splien interpolation is much more accurate than the equally spaced polynomial interpolation.

In [ ]: