# ME596 Homework 4

Erin Schmidt

**Problem Statement**

Maximum in-plane stress of a plane with a through-hole is given by

$$\sigma = \frac{Kp}{(D-d)t},$$

where $t$ is the thickness of the plate, $p$ is the pressure applied, and $K$ is the stress concentration factor. $K$ is given by

$$K = 1.11 + 1.11\left(\frac{d}{D}\right)^{-0.18}.$$

We must find the hole size that minimizes the $\sigma$. We shall make the notational simplification $\frac{d}{D} = x$. We can note from the problem formulation that we cannot write $\sigma$ strictly in terms of $x$; we must assume a value for $D$. We shall thence assume $D$, as well as $p$, and $t$ are all equal to 1. Given our assumptions we can write the stress equation as

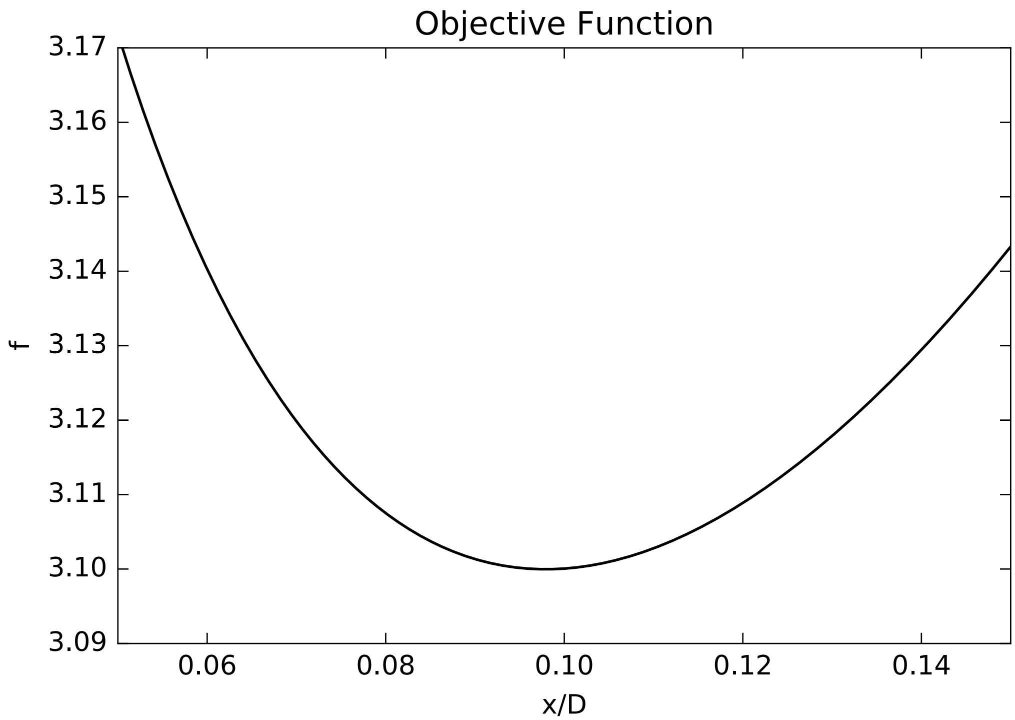
$$\sigma(1-x) = K.$$

This can be further simplified in terms of an explicit objective funtion as

$$\sigma = \frac{1.11 + 1.11x^{-0.18}}{1-x}.$$

The objective function is plotted below.

```
In [14]: import numpy as np
         import matplotlib
         import matplotlib.pyplot as plt
         from __future__ import division
         %config InlineBackend.figure_formats=['svg']
         %matplotlib inline
         plt.rc('pdf',fonttype=3)          # for proper subsetting of fonts
         plt.rc('axes',linewidth=0.5)      # thin axes; the default for lines is 1pt
         al = np.linspace( 0.05, 0.15, 500)
         plt.plot(al, (1.11 + 1.11*al**(-0.18))/(1 - al), 'k')
         plt.axis([0.05, 0.15, 3.09,3.17])
         plt.title("Objective Function")
         plt.ylabel("f")
         plt.xlabel("x/D")
         plt.show()
```




We can see (at least qualitatively), from the plot of the objective function that on the interval $0.05 < \alpha < 0.15$ the optimum value lies somewhere between 0.09 and 0.10, and the function evaluated in that range has an average value of about 3.10.

We shall proceed to find the minimum of the objective function by using both an equal interval search algorithm and a polynomial approximation.

### Equal Interval Search

In [18]:
```
#Equal Interval Search
#Erin Schmidt

#Adapted, with significant modification, from Arora et al.'s APOLLO
#implementation found in "Introduction to Optimum Design" 1st Ed. (1989).

import numpy as np

def func(al, count): #the objective function
    count = count + 1
    f = (1.11 + 1.11*al**(-0.18))/(1 - al)
    return f, count

def mini(au, al, count): #evaluates f at the minimum (or optimum) stationary point
    alpha = (au + al)*0.5
    (f, count) = func(alpha, count)
    return f, alpha, count

def equal(delta, epsilon, count, al):
    (f, count) = func(al, count)
```

```python
    fl = f #function value at lower bound
    #delta = 0.01 #step-size
    #au = 0.15 #alpha upper bound

    while True:
        aa = delta
        (f, count) = func(aa, count)
        fa = f
        if fa > fl:
            delta = delta * 0.1
        else:
            break

    while True:
        au = aa + delta
        (f, count) = func(au, count)
        fu = f
        if fa > fu:
            al = aa
            aa = au
            fl = fa
            fa = fu
        else:
            break

    while True:
        if (au - al) > epsilon: #compares interval size to convergence criteria
            delta = delta * 0.1
            aa = al #intermediate alpha
            fa = fl #intermediate alpha function value
            while True:
                au = aa + delta
                (f, count) = func(au, count)
                fu = f
                if fa > fu:
                    al = aa
                    aa = au
                    fl = fa
                    fa = fu
                    continue
                else:
                    break
            continue
        else:
            (f, alpha, count) = mini(au, al, count)
            return f, alpha, count


#run the program
delta = 0.01
epsilon = 1E-3
count = 0
al = 0.01 # alpha lower bound
```

```
(f, alpha, count) = equal(delta, epsilon, count, al)
print('The minimum is at {:.4f}'.format(alpha))
print('The function value at the minimum = {:.4f}'.format(f))
print('Total number of function calls = {}'.format(count))
```

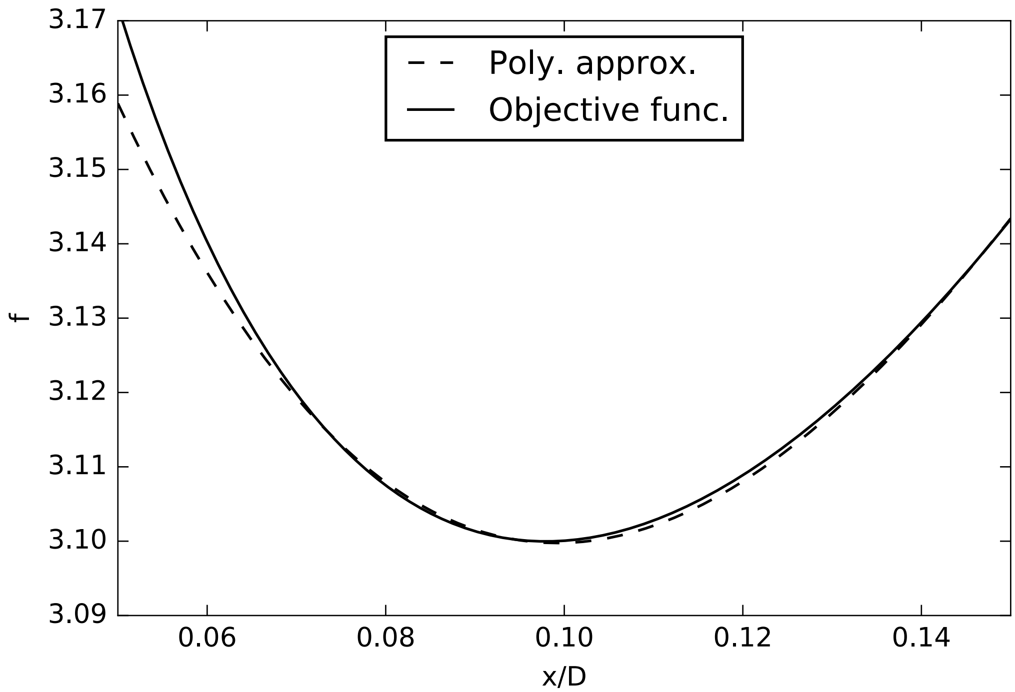
```
The minimum is at 0.0979
The function value at the minimum = 3.1000
Total number of function calls = 32
```

### Polynomial Approximation

In [22]:
```
# Polynomial approximation (4-point cubic)
# -Erin Schmidt

import numpy as np
from math import sqrt

# make an array with random values between 0.05 and 0.15 with 4 entries
x = (0.05 + np.random.sample(4)*0.15)

# make an array of function values at the 4 points of x
def f(x): # the objective function
    return (1.11 + 1.11*x**(-0.18))/(1 - x)

f_array = []
i = 0
while i <= len(x) - 1:
    f_array.append(f(x[i]))
    i += 1

# use the equations from Vanderplaats 1984 to solve coefficients
q1 = x[2]**3 * (x[1] - x[0]) - x[1]**3 * (x[2] - x[0]) + x[0]**3 * (x[2] - x[1])
q2 = x[3]**3 * (x[1] - x[0]) - x[1]**3 * (x[3] - x[0]) + x[0]**3 * (x[3] - x[1])
q3 = (x[2] - x[1]) * (x[1] - x[0]) * (x[2] - x[0])
q4 = (x[3] - x[1]) * (x[1] - x[0]) * (x[3] - x[0])
q5 = f_array[2] * (x[1] - x[0]) - f_array[1] * (x[2] - x[0]) + f_array[0] * (x[2] - x[1])
q6 = f_array[3] * (x[1] - x[0]) - f_array[1] * (x[3] - x[0]) + f_array[0] * (x[3] - x[1])

a3 = (q3*q6 - q4*q5)/(q2*q3 - q1*q4)
a2 = (q5 - a3*q1)/q3
a1 = (f_array[1] - f_array[0])/(x[1] - x[0]) - \
a3*(x[1]**3 - x[0]**3)/(x[1] - x[0]) - a2*(x[0] + x[1])
a0 = f_array[0] - a1*x[0] - a2*x[0]**2 - a3*x[0]**3
a = [a1, 2*a2, 3*a3] #coefficients of f'

# find the zeros of the f' polynomial (using the quadratic formula)
b = a2**2 - 3*a1*a3
X1 = (-a2 + sqrt(b))/(3*a3)
X2 = (-a2 - sqrt(b))/(3*a3)

print('roots = ', X1, X2)

# plot the results
plt.rc('pdf',fonttype=3)          # for proper subsetting of fonts
plt.rc('axes',linewidth=0.5)      # thin axes; the default for lines is 1pt
```

```python
x = np.linspace( 0.05, 0.15, 500)
plt.plot(x, a0 +a1*x + a2*x**2 + a3*x**3, 'k--', label='Poly. approx.')
plt.plot(x, (1.11 + 1.11*x**(-0.18))/(1 - x), 'k', label='Objective func.')
plt.axis([0.05, 0.15, 3.09,3.17])
legend = plt.legend(loc='upper center', shadow=False, fontsize='large')
plt.ylabel("f")
plt.xlabel("x/D")
plt.show()

poly_root = [0.097620387704, 0.0985634827486, 0.0969340736066, \
             0.098775097463, 0.102426814371, 0.101638472077, \
             0.0991941169039, 0.095873175811]
print('polynomial root std. deviation = ', np.std(poly_root))
```

```
roots =  0.0992038240925 0.283337512729
```




```
polynomial root std. deviation =  0.00208605896509
```

**Discussion**

Both the equal interval search and the polynomial approximation seem to yield results that agree to 3 decimal places. Both also return values squarely reside within our expected bounds (being between 0.09 and 0.10), which we determined by a qualitative examination of the plot of the original objective function.

The roots of our polynomial are real and unique. On the bounds $0.05 < x < 0.15$ the minimum of the function by the polynomial approximation is the first root, at $x = 0.99$. Though this agrees well with the result obtained via the equal interval search method for this problem, the polynomial approximation method appears to be sensitive to the initial guess points of $x$, even using the relatively accurate 4-point cubic approximation. The standard deviation of ten sets of randomly sampled $x$ points on our bounds is 0.0021.

The equal interval search might also suffer from choice of an initial point x, on the objective function, especially if the objective function is multi-modal. Practically speaking the hole size $d$ must be on the bounds $0 < d < 1$ to make physical sense. If $d \geq D$ or $d \leq 0$ the search function will return divide by zero errors. However, within the bounds of 0.05 and 0.15 the equal interval search function returns an average value of 0.0979.