I attempted to number all of the relevant tasks, but a few sections may border on the edges of the tasks. Additionally, the source code for the python parts of my project are attached. For symbolic operations, a significant part was performed using GNU Maxima.

Part I.

(Task 1) The complete code for my modified solve_1D.py is attached at the end.

(Task 2) Given the function: \( f(x) = e^{k(x-c)} - \cos(k(x-c)) - k(x-c) \), I found the first derivative to be:

\[
 f'(x) = k * e^{k(x-c)} + k * \sin(k(x-c)) - k
\]

and the second derivative:

\[
 f''(x) = k^2 * e^{k(x-c)} + k^2 * \cos(k(x-c))
\]

Using the original Newton-Raphson solver as well as my modified Newton-Raphson solver (included in the attached source code), I gained the following results:

Calculating with classic Newton solver.

\[
\begin{array}{ccc}
 i & x & d \\
 2 & 1.24444 & -0.18121 \\
 3 & 1.33059 & -0.08615 \\
 4 & 1.37270 & -0.04211 \\
 5 & 1.39353 & -0.02083 \\
 6 & 1.40389 & -0.01036 \\
 7 & 1.40906 & -0.00517 \\
 8 & 1.41164 & -0.00258 \\
 9 & 1.41292 & -0.00129 \\
 10 & 1.41357 & -0.00064 \\
 11 & 1.41389 & -0.00032 \\
 12 & 1.41405 & -0.00016 \\
 13 & 1.41413 & -0.00008 \\
 14 & 1.41417 & -0.00004 \\
 15 & 1.41419 & -0.00002 \\
 16 & 1.41420 & -0.00001 \\
 17 & 1.41421 & -0.00001 \\
\end{array}
\]

Calculating with modified Newton-Raphson solver.

\[
\begin{array}{ccc}
 i & x & d \\
 2 & 1.40216 & -0.33893 \\
 3 & 1.41420 & -0.01204 \\
 4 & 1.41421 & -0.00001 \\
 5 & 1.41421 & -0.00000 \\
\end{array}
\]

(Task 4) This demonstrates that while it takes 17 iterations of the classic Newton-Raphson method, only 5 iterations of the modified method are required to arrive at an answer within \( \pm 10^{-5} \). It also clearly confirms that there is a root of \( f(x) \) at \( x = c = \sqrt{2} \). Because the modified solver uses the 2\textsuperscript{nd} derivative as well as the square of the 1\textsuperscript{st} derivative in the denominator of each step, it converges with the solution quadratically. This gives the significant performance improvement seen in the 71\% reduction in steps needed to find a root.
Part II

The graph of \( f(x) \) with \( k = 10^{-8} \) showed a large number of sharp swings back and forth and points at the end of these swings. Many of these “spikes” cross the x-axis, indicating that there is a large number of zeros in the interval \([-1,4]\). As a result, the conventional methods will not be able to find a zero.

(Task 7) Given \( f(x) \) and its first and second derivatives found in Part I, I found the 3rd derivative of \( f(x) \): \( f'''(x) = k^3 e^{k(x-c)} - k^3 \sin(k(x-c)) \). This gives the Taylor approximation \( p_3(x) \):

\[
p_3(x) = e^{k(x_0-c)} - \cos(k(x_0-c)) - k(x_0-c) + \left( k e^{k(x_0-c)} + k \sin(k(x_0-c)) - k \right)(x-x_0)
+ \frac{k^2 e^{k(x_0-c)} + k^2 \cos(k(x_0-c))}{2}(x-x_0)^2 + \frac{k^3 e^{k(x_0-c)} - k^3 \sin(k(x_0-c))}{6}(x-x_0)^3
\]

(Task 8) This gives the first derivative, \( p_3'(x) \):

\[
p_3'(x) = \left( k^3 e^{k(x_0-c)} - k^3 \sin(k(x_0-c)) \right) \frac{1}{2}(x-x_0)^2 + \left( k^2 \cos(k(x_0-c)) + k^2 e^{k(x_0-c)} \right)(x-x_0)
+ k \sin(k(x_0-c)) + k e^{k(x_0-c)} - k
\]

The second derivative, \( p_3''(x) \) is:

\[
p_3''(x) = \left( k^3 e^{k(x_0-c)} - k^3 \sin(k(x_0-c)) \right)(x-x_0) + k^2 \cos(k(x_0-c)) + k^2 e^{k(x_0-c)}
\]

The Taylor series was calculated by hand, and the derivatives determined using GNU Maxima.
(Task 10) Based on the graph, while it appears that \( p_3(x) \) closely approximates \( f(x) \), it does not appear to cross the x-axis and thus will be difficult to use to (approximately) solve \( f(x) = 0 \). Very likely, \( p_3(x) \) stays above the axis because of the way the computer calculates the function. Very small values in each term from the evaluation of \( f(x) \) in calculating the coefficients for the Taylor approximation keep the overall value slightly positive, due to the bias in rounding errors.

(Task 11) Attempting to apply the Newton-Raphson method with iterations=100 and xtol=1e-5 to \( p_3(x) \) results in the algorithm iterating through all 100 steps without finding a solution that approximates \( f(x) \). The last few lines of output, displayed below, show X is wildly fluctuating, indicating that it is not able to approach a zero. This also indicates that even with a much larger number of iterations, the approximation would still fail to reach a zero. This is evident from the graph, as shown where \( p_3(x) \) has a minimum that does not cross nor reach the x-axis.

\[
\text{i = 93, x = 1.65222, d = 0.44474} \\
\text{i = 94, x = 1.23669, d = 0.41552} \\
\text{i = 95, x = 1.72300, d = -0.48631} \\
\text{i = 96, x = 1.34006, d = 0.38294} \\
\text{i = 97, x = 2.32885, d = -0.98879} \\
\text{i = 98, x = 1.79437, d = 0.53448} \\
\text{i = 99, x = 1.41865, d = 0.37572} \\
\text{i = 100, x = -14.47664, d = 15.89529} \\
\text{i = 101, x = -6.52677, d = -7.94987} \\
\text{Reached maximum number of iterations}
\]

**Part III**

(Task 12)
(Task 13) Using the min() and max() functions in Python, I determined that p() and q() maintained an almost constant $5.3075 \times 10^{-17}$ spacing, which is approximately 2 orders of magnitude smaller than the graph. This is consistent with a single order of magnitude offset between the two as a result of the different values for $x_0$.

(Task 14) Using a Newton-Raphson solver to find the location where $\frac{dp}{dx} = 0$ to find a local extreme (local minimum) for $p(x)$ and then applying this value to $f(x)$ to check for a zero resulted in a value of $x = 1.41421$, giving $f(x) = f(1.41421) = 0$, which confirms the presence of the zero at $\sqrt{2}$. The output is given below:

Looking for 0 based on dp/dx.
Found dp/dx=0 at 1.41421.
f(1.41421)=0.


## Appendix I – proj1.py (Main Source File)

```python
# Math/CSc 4610 Project 1

from solve_1D import *
import math
from matplotlib.pylab import axhline

# Setup f(x), df(x), and ddf(x)
# f(x)=e^{k(x-c)}-\cos(k(x-c))-k(x-c)
# k=1, c=sqrt(2) (Part I)
# derivative and 2nd derivative found
# manually and confirmed with maxima
fx_k=1
fx_c=sqrt(2)

def f(x):
    global fx_k
    global fx_c
    xminc=x-fx_c #x-c
    kxc=fx_k*xminc #k(x-c)
    r=pow(math.e,kxc)-\cos(kxc)-kxc
    #print "f(x)=%.5f" % r
    return r

def df(x):
    global fx_k
    global fx_c
    xminc=x-fx_c #x-c
    kxc=fx_k*xminc #k(x-c)
    r=fx_k*\pow(math.e,kxc)+fx_k*\sin(kxc)-fx_k
    #print "df(x)=%.5f" % r
    return r

def ddf(x):
    global fx_k
    global fx_c
    xminc=x-fx_c #x-c
    kxc=fx_k*xminc #k(x-c)
    r=pow(fx_k,2)*\pow(math.e,kxc)+pow(fx_k,2)*\cos(kxc)
    #print "ddf(x)=%.5f" % r
    return r

x0=-1 # Left Endpoint
print "Part I."
newton_raphson(x0,f,df,1e-5,100,False)

print "Calculating with modified Newton-Raphson solver."
newton_raphson_mod(x0,f,df,ddf,1e-5,100,False)

# Create graph of f(x) over 100 points in [-1,4]
x_var = linspace(-1,4,100)
y_var_1 = f(x_var)
#y_var_2 = df(x_var)
```

---

Note: The code snippet above is a simplified representation of the content in the image. It includes the main parts of the code related to setting up the function, calculating its derivatives, and performing Newton's method. The graphical part is also included for creating a graph of the function over a specified interval.

---

The code snippet demonstrates the setup of a mathematical function and its derivatives, followed by the implementation of Newton's method for finding roots of the function. It also includes a section for creating a graph of the function over a given interval.
# y_var_3 = ddf(x_var)
plot(x_var, y_var_1, 'k-') # f(x), black
#plot(x_var, y_var_2, 'r-') # first derivative, red
#plot(x_var, y_var_3, 'b-') # second derivative, blue
axhline(y=0) # add a horizontal axis

# Part II
print "\nPart II."
fx_k = pow(10, -8)
fx_c = sqrt(2) # explicit
p_x0 = 1.3 # per part 2, item 8

# Draw the plot
x_var = linspace(-1, 4, 100)
vals_f = f(x_var)
figure()
plot(x_var, vals_f, 'k-')
axhline(y=0) # X-Axis

def p(x):
    # 3rd-order taylor polynomial of f(x)
    global fx_k
    global fx_c
    global p_x0 # Taylor poly x0 point
    kx0c = fx_k*(p_x0-fx_c) # k(x0-c), used a lot
    # first term
    t1 = pow(math.e, kx0c) - cos(kx0c) - kx0c
    # second term
    t2 = fx_k* pow(math.e, kx0c) + fx_k* sin(kx0c) - fx_k
    t2 *= (x-p_x0)
    # third term
    t3 = pow(fx_k, 2)*pow(math.e, kx0c) + pow(fx_k, 2)*cos(kx0c)
    t3 /= 2
    t3 *= pow(x-p_x0, 2)
    # fourth term
    t4 = pow(fx_k, 3)*pow(math.e, kx0c) - pow(fx_k, 3)*sin(kx0c)
    t4 /= 6
    t4 *= pow(x-p_x0, 3)
    # results
    #print "t1=%g t2=%g t3=%g t4=%g" % (t1,t2,t3,t4)
    return t1+t2+t3+t4

def dp(x):
    # p'(x)
    global fx_k
    global fx_c
    global p_x0 # Taylor poly x0 point
    kx0c = fx_k*(p_x0-fx_c) # k(x0-c), used a lot
    # first term
    t1 = pow(fx_k, 3)*pow(math.e, kx0c) - pow(fx_k, 3)*sin(kx0c)
    t1 /= 2
    t1 *= pow(x-p_x0, 2)
    # second term
    t2 = pow(fx_k, 2)*cos(kx0c) + pow(fx_k, 2)*pow(math.e, kx0c)
    t2 *= x-p_x0
    # third term
    t3 = fx_k*sin(kx0c) + fx_k*pow(math.e, kx0c) - fx_k
def ddp(x):
    # p''(x)
    global fx_k
    global fx_c
    global p_x0  # Taylor poly x0 point
    kx0c = fx_k*(p_x0-fx_c)  # k(x0-c), used a lot
    # first term
    t1=pow(fx_k,3)*pow(math.e,kx0c)+pow(fx_k,3)*sin(kx0c)
    t1*=x-p_x0
    # second term
    t2=pow(fx_k,2)*cos(kx0c)+pow(fx_k,2)*pow(math.e,kx0c)
    # results
    return t1+t2

# Plot p, in red
vals_p=p(x_var)
plot(x_var,vals_p,'r-')

# Attempt Newton's method on p(x)
print "Applying Newton's Method to Taylor's Approximation"
newton_raphson(x0,p,dp,1e-5,100,False)

# Part III
print "\nPart III"

# Because p is versatile enough to use the global p_x0 to
# calculate the value of the polynomial, we can just alias
# q to p and change p_x0 for task 12
p_x0 = 0
q=p
dq=dp
ddq=ddp

# Graph q over same x_var.  This will now have p_x0=0
# This is plotted in green.
vals_q=q(x_var)
plot(x_var,vals_q,'g-')

# Compare p (vals_p) and q (vals_q).
diffs=vals_p-vals_q
mindiff=min(diffs)
maxdiff=max(diffs)
print "Difference between p and q over the "
print "interval [-1,4] is between %g and %g." % (mindiff, maxdiff)

# Look for dp=0 via Newton-Raphson method
print "\nLooking for 0 based on dp/dx."
point=newton_raphson(fx_c,dp,ddp,1e-5,100)
print "Found dp/dx=0 at %g." % point
print "f(%g)=%g." % (point,f(point))

# Show all graphs
show()
Appendix II – solve_1D.py (Solver Functions)

# Solver library
# Original version by Dr. Rob Clewley, GSU Math Department

from __future__ import division
from numpy import *
from matplotlib.pylab import figure, plot, show, hold

def bisection(xlo, xhi, f, xtol, max_iter=100, quiet=True):
    flo = f(xlo)
    fhi = f(xhi)
    # safety check on input arguments
    assert xlo < xhi
    assert sign(flo) != sign(fhi)
    assert flo != fhi
    assert xtol > 0
    assert max_iter > 1
    i = 1
    while i <= max_iter:
        d = (xhi - xlo)/2.
        p = xlo + d
        if d < xtol:
            break
        fp = f(p)
        if fp == 0:
            break
        i += 1
        if fp*flo > 0:
            # have opposite signs
            xlo = p
            flo = fp
        else:
            # have same signs
            xhi = p

    if i >= max_iter:
        print "Reached maximum number of iterations"
    return p

def fixed_point(x0, f, xtol, max_iter=100, quiet=True):
    assert xtol > 0
    assert max_iter > 1
    # initialize x with x0
    x = x0
    x_new = f(x)
    abs_error = abs(x_new - x0)
    i = 1
    while abs_error > xtol and i <= max_iter:
        x_new = f(x)
        abs_error = abs(x_new - x)
        i += 1
        x = x_new
    if not quiet and i >= max_iter:
        print "Reached maximum number of iterations"
print "Reached maximum number of iterations"
return x_new

def newton_raphson(x0, f, df, xtol, max_iter=100, quiet=True):
    assert xtol > 0
    assert max_iter > 1
    i = 1
    f_x0 = f(x0)
    df_x0 = df(x0)
    d = f_x0/df_x0
    x_new = x0 - d
    if abs(d)<xtol:
        return x0
    while abs(d) > xtol and i <= max_iter:
        d = f(x_new)/df(x_new)
        x_new = x_new - d
        i += 1
        if not quiet:
            print "i = %d, x = %.5f, d = %.5f" % (i, x_new, d)
        if not quiet and i >= max_iter:
            print "Reached maximum number of iterations"
    # Added 10/21/08 -- DWT
    if not quiet and i == 1:
        print "Failed to meet initial condition abs(d) > xtol"
    return x_new

# Modified Newton-Raphson method -- DWT 10/20/08
# f, df, ddf = function, 1st, and 2nd derivatives
def newton_raphson_mod(x0, f, df, ddf, xtol, max_iter=100, quiet=True):
    assert xtol > 0
    assert max_iter > 1
    i = 1
    f_x0 = f(x0)
    df_x0 = df(x0)
    d = f_x0/df_x0
    x_new = x0 - d
    while abs(d) > xtol and i <= max_iter:
        d = f(x_new)*df(x_new)/(pow(df(x_new),2) - f(x_new)*ddf(x_new))
        x_new = x_new - d
        i += 1
        if not quiet:
            print "i = %d, x = %.5f, d = %.5f" % (i, x_new, d)
        if not quiet and i == max_iter:
            print "Failed to meet initial condition abs(d) > xtol"
    return x_new

def secant(x0, x1, f, xtol, max_iter=100, quiet=True):
    assert xtol > 0
    assert max_iter > 1
    assert x0 != x1
    i = 2
f0 = f(x0)
f1 = f(x1)
not_done = True
while not_done and i <= max_iter:
    x = x1 - f1*(x1-x0)/(f1-f0)
    fx = f(x)
    x0 = x1
    f0 = f1
    d = abs(x-x1)
    x1 = x
    f1 = fx
    i += 1
    if not quiet:
        print "i = %d, x = %5.5f, d = %5.5f" % (i, x, d)
    not_done = d > xtol
if not quiet and i >= max_iter:
    print "Reached maximum number of iterations"
return x