In [2]:
```
"""
Arjun Srivastava
arj1
AMATH 301 B
"""

import numpy as np
import scipy.linalg
import scipy.optimize
import matplotlib.pyplot as plt
import time
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

In [32]:
```
# Problem 1

# a)

# Function to generate discrete Poisson matrix of given dimension
def discrete_poisson(dim: int):
    A = np.zeros((dim, dim))
    np.fill_diagonal(A, 2), np.fill_diagonal(A[1:], -1), np.fill_diagonal(A[:,
1:], -1)
    return A

A = discrete_poisson(1000)
D = np.diag(np.diag(A))
U = np.triu(A, 1)
L = np.tril(A, -1)

# Returns minimum eigenvalue because section search methods can only find the
 minima. This method allows me to flip the function
def max_eig(omega: float):
    P = ((1/omega) * D) + L
    T = (((omega - 1)/omega) * D) + U
    M = -scipy.linalg.solve(P, T)
    w, V = np.linalg.eig(M)
    return -np.max(np.abs(w))

"""
We cannot use Newton's method for this problem because it requires the functio
n to be differentiable. The function to find the
maximum eigenvalue, however, uses matrices and is not differentiable. As such,
we must use a different method to find the maximum.
"""

# Section Search

t0 = time.time()
a = 1
b = 2
c = 0.5001
tolerance = 1e-8

for k in range(100):
    x = c * a + (1 - c) * b
    y = (1 - c) * a + c * b

    if max_eig(x) < max_eig(y):
        b = y
    else:
        a = x

    if (b - a) < tolerance:
        break
t1 = time.time()
ttime = t1 - t0

print('Section Search:', x, k + 1, ttime, sep='\n')
```

```
"""
A114:
k = 27
time = 0.6003904342651367
omega = 1.0000000074879227

A1000:
k = 27
time = 85.3413507938385
omega = 1.0000007255143664
"""

# Golden Section Search

t0 = time.time()
a = 1
b = 2
c = (-1 + np.sqrt(5)) / 2

x = c * a + (1 - c) * b
fx = max_eig(x)
y = (1 - c) * a + c * b
fy = max_eig(y)
for k in range(100):
    if fx < fy:
        b = y
        y = x
        fy = fx
        x = c * a + (1 - c) * b
        fx = max_eig(x)
    else:
        a = x
        x = y
        fx = fy
        y = (1 - c) * a + c * b
        fy = max_eig(y)
    if (b - a) < tolerance:
        break
t1 = time.time()
ttime = t1 - t0

print('\n', 'Golden Section Search:', x, k + 1, ttime, sep='\n')

"""
A114:
k = 39
time = 0.43085408210754395
omega = 1.000000002700889

A1000:
k = 39
time = 63.05891537666321
omega = 1.000000002700889
"""

"""
With a 1000x1000 sized Poisson matrix, the normal section search method takes
```

```
 85.34 seconds and the golden section search takes
63.06 seconds. While the difference of .2 seconds beetween the two methods for
the 114x114 matrix was not very significant, this
difference of ~20 seconds reinforces the idea that the golden section method i
s more efficient. The number
of steps required to find the maximum is greater for the golden section method
(38 vs. 26 for the section method), but the time
difference is much more significant.
"""
```

```
Section Search:
1.0000007255143664
26
85.3413507938385


Golden Section Search:
1.000000002700889
38
63.05891537666321
```

Out[32]:

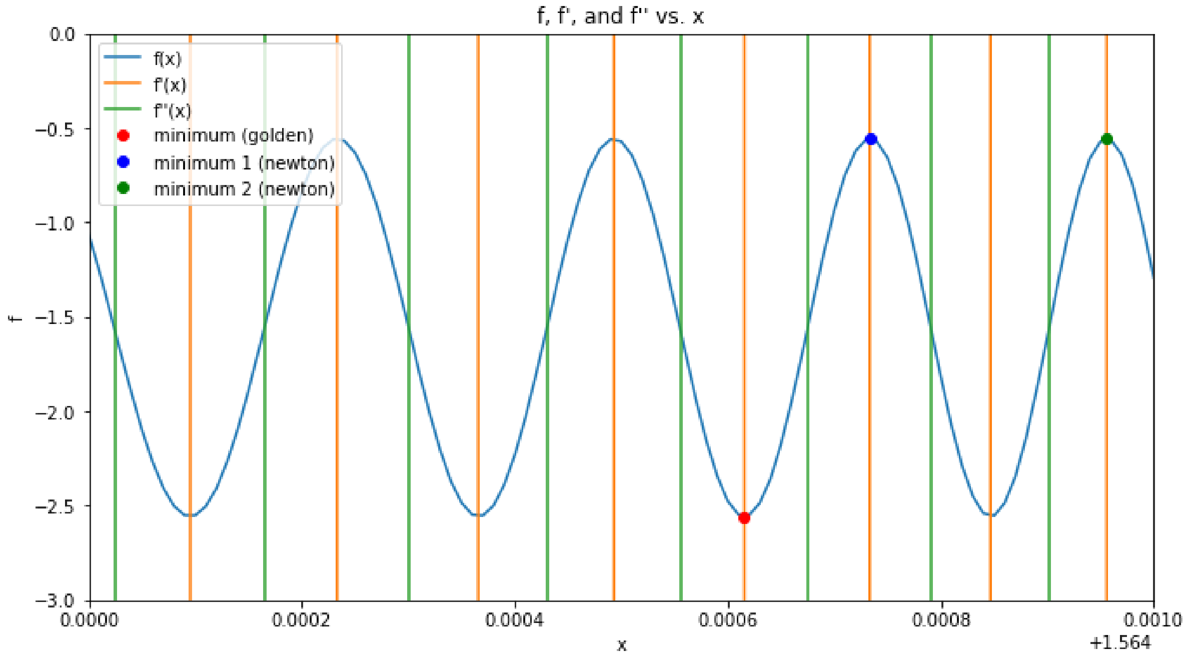
```
'\nA114:\nk = 38\ntime = 0.43085408210754395\nomega = 1.000000002700889\n\nA1
000:\nk = \ntime = \nomega = \n'
```

In [3]:
```
# b)

# Used Wolfram Alpha
f = lambda x : np.sin(np.tan(x)) - np.tan(np.sin(x))
fprime = lambda x : (1/np.cos(x)**2) * np.cos(np.tan(x)) - np.cos(x) * (1/np.c
os(np.sin(x))**2)
fdprime = lambda x : -(1/np.cos(x)**2) * ((1/np.cos(x)**2) * np.sin(np.tan(x))
- 2 * np.tan(x)*np.cos(np.tan(x))) - (1/np.cos(np.sin(x)**2) * (2 * np.cos(x)*
*2 * np.tan(np.sin(x)) - np.sin(x)))

# Golden Section Search

tolerance = 1e-16
a = 1.5646
b = 1.5647
c = (-1 + np.sqrt(5)) / 2

x = c * a + (1 - c) * b
fx = f(x)
y = (1 - c) * a + c * b
fy = f(y)
for k in range(100):
    if fx < fy:
        b = y
        y = x
        fy = fx
        x = c * a + (1 - c) * b
        fx = f(x)
    else:
        a = x
        x = y
        fx = fy
        y = (1 - c) * a + c * b
        fy = f(y)
    if (b - a) < tolerance:
        break

# x = 1.5646156310416386
# min = -2.55734229768688

# Newtons method

# Function assumes f, f', and f
def newtons_method(guesses: int, x0: float, tolerance: float):
    X = np.zeros(guesses + 1)
    X[0] = x0
    for k in range(guesses):
        X[k + 1] = X[k] - fprime(X[k]) / fdprime(X[k])
        if np.abs(fprime(X[k + 1])) < tolerance:
            break
    X = X[:(k+2)]
    return X[len(X)-1]

final_guess1 = newtons_method(100, 1.5647, tolerance)
final_guess2 = newtons_method(100, 1.5648, tolerance)
print(final_guess1, final_guess2, sep='\n')
```

```
"""
After trying numerous different guesses, I have included the most interesting
 guess results. In the graph below, it is clear that
the golden section search method found the correct result. With Newton's metho
d, however, guessing with the result of the golden
section search yielded the correct minimum, but using guesses 1.5647 and 1.564
8 yielded incorrect maxima and minima, as shown
in the graph. Due to the incredibly small window between inflections, this is
 to be expected since the Newton method does not include
bounds. If we make a guess even slighly out of the 1.5646 and 1.5647 window, t
he function converges to the wrong x value.
Even though my 1.5647 guess is at the outer bound, the function still converge
d to the next extreme, and the same thing happened
with my other guess. It seems like Newton's method is not very effective for s
olving trig functions or other functions with very
small wavelengths (like f in this problem). The only case where Newton's metho
d would work is if we made a very accurate initial
guess. With Newton's methods, we are checking whether f' is close to zero in t
he test, but this is true for both minima and maxima.
Thus, using this method is ineffective for this type of problem.
"""

# Plots

xs = np.arange(1.56, 1.57, .00001)
plt.figure(figsize=(11, 6))
plt.title("f, f', and f'' vs. x")
plt.plot(xs, f(xs), xs, fprime(xs), xs, fdprime(xs), x, f(x), 'ro', final_gues
s1, f(final_guess1), 'bo', final_guess2, f(final_guess2), 'go')
plt.ylim(-3, 0)
plt.xlim(1.564, 1.565)
plt.xlabel('x')
plt.ylabel('f')
plt.legend(("f(x)", "f'(x)", "f''(x)", "minimum (golden)", 'minimum 1 (newto
n)', 'minimum 2 (newton)'))
```

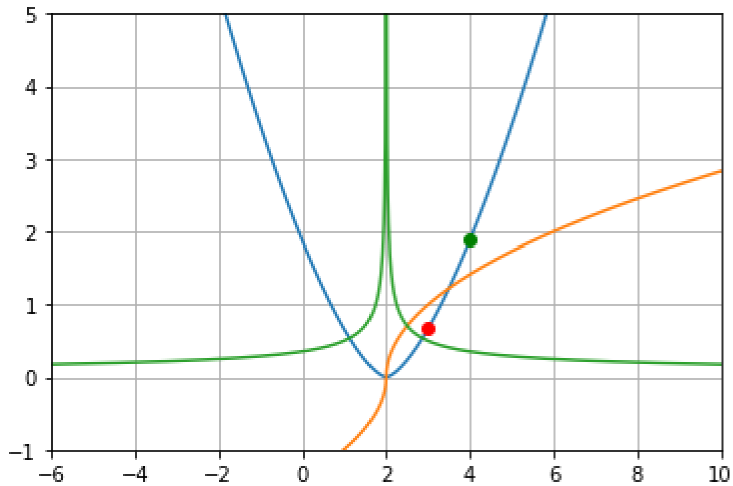
```
1.5647333556076446
1.5649558448108543
```

Out[3]: <matplotlib.legend.Legend at 0x285162802c8>



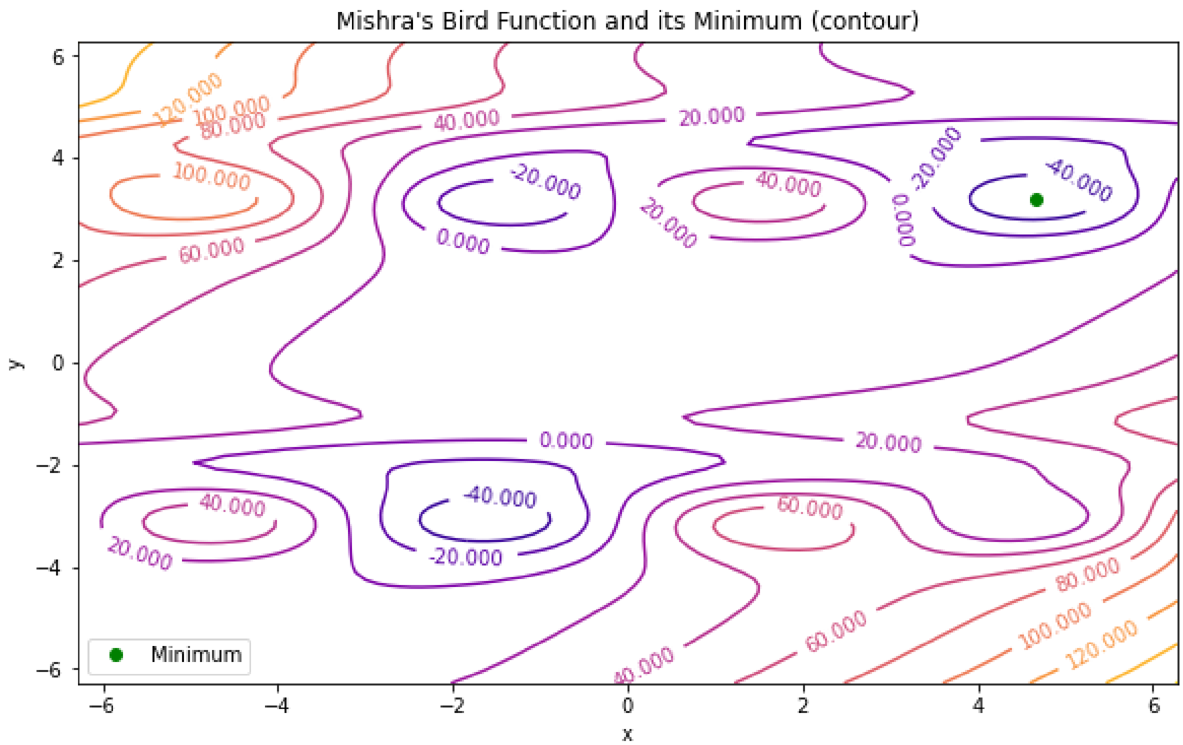
In [50]:
```
# c)

f = lambda x : (2/3)*np.abs(x-2)**(3/2)
fprime = lambda x : (x-2)/np.sqrt(np.abs(x-2))
fdprime = lambda x : (1/2)*np.abs(x-2)**(-1/2)

guess1 = newtons_method(90, 3, tolerance)
guess2 = newtons_method(90, 4, tolerance)
xs = np.linspace(-6, 10, 1000)
plt.plot(xs, f(xs), xs, fprime(xs), xs, fdprime(xs), guess1, f(guess1), 'ro',
guess2, f(guess2), 'go')
plt.xlim(-6, 10)
plt.ylim(-1, 5)
plt.grid()
# Graph shows guess at a random non-extrema point

"""
The guess never converges. No matter how many iterations the for loop complete
s, the result will never reach the true minimum.
As can be seen in the graph both guesses converged to seemingly random non-ext
rema points. They are actually not random though; the
guesses are simply the initial guesses I tried (3 and 4). Since both derivativ
es are undefined for x = 2 (the true minimum),
Python cannot actually make a guess for x = 2, as the result in Newton's equat
ion would be undefined.
"""
guess1
```

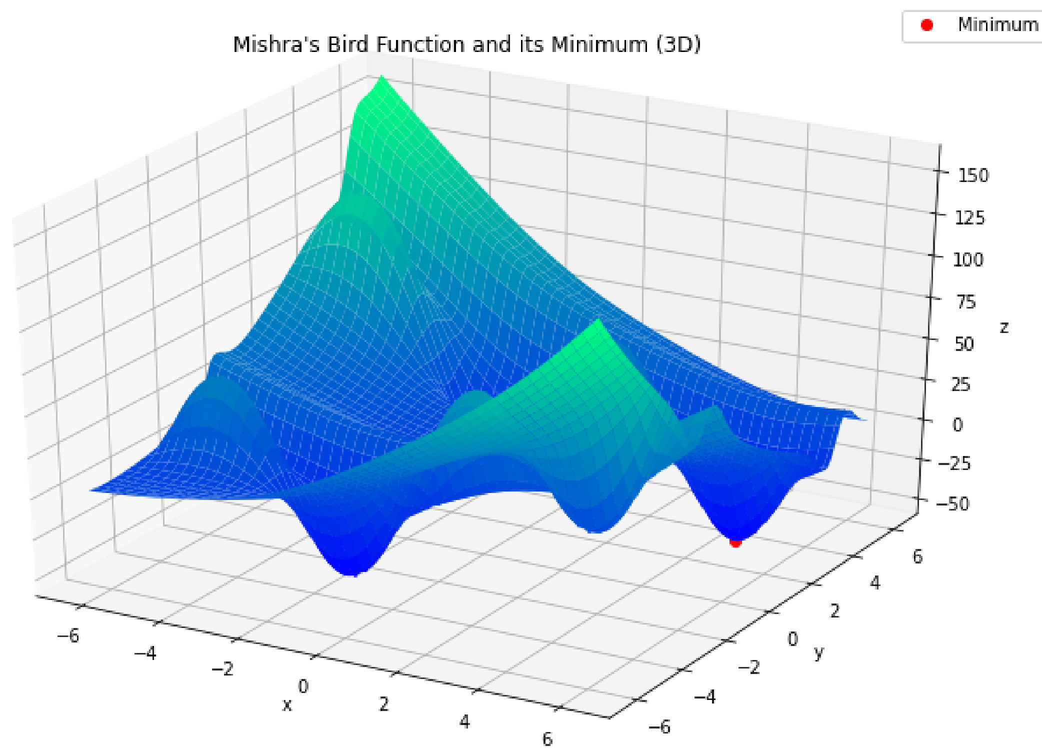
Out[50]: 3.0



In [5]:
```
# Problem 2 (some letters are missing because not every part of the problem in
volves writing new code)

# a)

f = lambda v : np.sin(v[0])*np.exp((1-np.cos(v[1]))**2) + np.cos(v[1])*np.exp
((1-np.sin(v[1]))**2) + (v[0]-v[1])**2

# b)

x = y = np.linspace(-2*np.pi, 2*np.pi, 100)
X, Y = np.meshgrid(x, y)

# c)

Z = f([X, Y])

# i)

plt.figure(figsize=(10, 6))
con = plt.contour(X, Y, Z, levels=10, cmap=cm.plasma)
plt.clabel(con)
xmin = scipy.optimize.minimize(f, [5, 3], method='Nelder-Mead')
x_, y_ = xmin.x
z_ = f(np.array([x_, y_]))
plt.plot(x_, y_, 'go', label='Minimum')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Mishra's Bird Function and its Minimum (contour)")
plt.legend(loc='lower left')

# o)

fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111, projection='3d', alpha=0.8)
ax.plot_surface(X, Y, Z, cmap=cm.winter)
ax.plot([x_], [y_], [z_], 'ro', label='Minimum')
ax.set_title("Mishra's Bird Function and its Minimum (3D)")
ax.set_ylabel('y')
ax.set_xlabel('x')
ax.set_zlabel('z')
ax.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x28516c56948>





In [ ]: