In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import copy
plt.rc('font',family='serif')
```
executed in 301ms, finished 02:18:25 2021-12-05

In [2]:
```
d=1000 # d: dimension
n=2000 # n: number of points
A = np.random.normal(size=(n,d)) / np.sqrt(n) # matrix containing the data points
y = np.random.normal(size=n)
lambd= 1
```
executed in 78ms, finished 02:18:25 2021-12-05

We consider the Ridge cost function:

$$f(x) = \frac{1}{2}\|Ax - y\|^2 + \frac{\lambda}{2}\|x\|^2, \tag{1}$$

where $\lambda > 0$ is some regularization parameter that we take equal to 1. The matrix A and the vector y are defined in the cell above.

(a) Show that f is can be written in the format the function f of Problem 12.2, for some $M \in \mathbb{R}^{d\times d}$, $b \in \mathbb{R}^d$ and $c \in \mathbb{R}$. Compute numerically the values of L and μ. Plot the eigenvalues of $H_f(x)$ using an histogram.

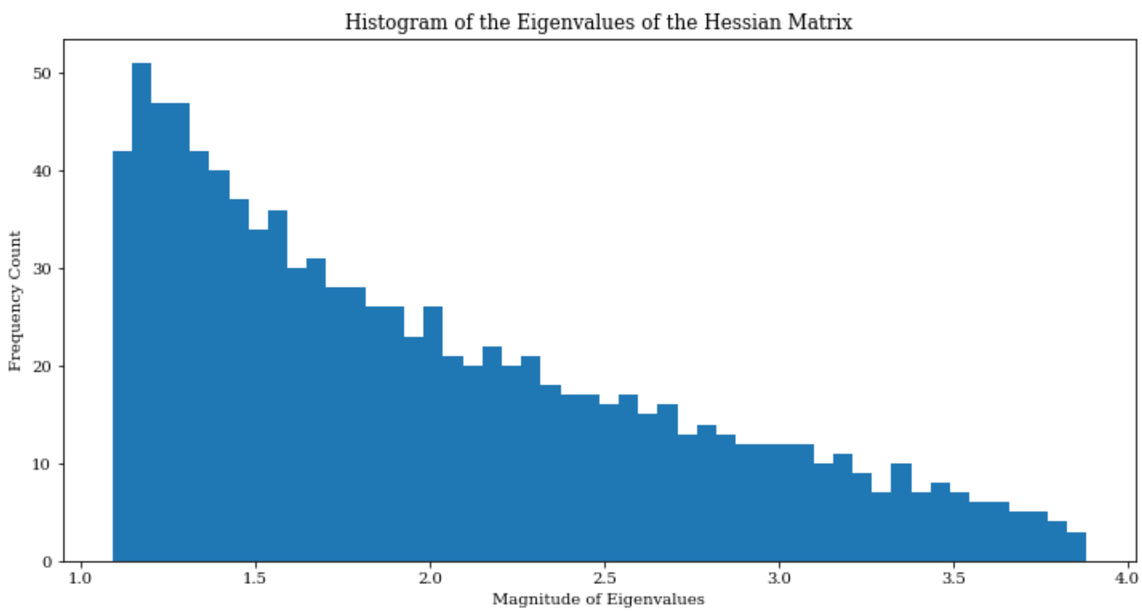
We can manipulate the expression by expanding the terms with the squared norms, and then perform algebra to achieve the desired structure of $f(x) = x^T Mx - \langle x, b\rangle + c$

$$\begin{aligned}
f(x) &= \frac{1}{2}\|Ax - y\|^2 + \frac{\lambda}{2}\|x\|^2 \\
f(x) &= \frac{1}{2}(Ax - y)^T(Ax - y) + \frac{\lambda}{2}x^T x \\
f(x) &= \frac{1}{2}(x^T A^T Ax - 2x^T A^T y + y^T y) + \frac{\lambda}{2}x^T x \\
f(x) &= \frac{1}{2}x^T(A^T A + \lambda Id)x - x^T A^T y + \frac{y^T y}{2}
\end{aligned} \tag{2}$$

Now we have our ridge regression cost function in similar structure to that of a quadratic convex function, $f(x) = x^T Mx - \langle x, b\rangle + c$, where in our case, $M = A^T A + \lambda Id$, $b = A^T y$ and, $c = \frac{y^T y}{2}$

In [3]:
```
#Create the identity matrix and the hessian
I = np.identity(n=d)
hessian_matrix = A.T@A + lambd*I

#Get eigenvalues / vectors using numpy
eigen_values, eigen_vectors = np.linalg.eigh(hessian_matrix)

#Save min and max eigenvals for later
L = np.max(eigen_values)
mu = np.min(eigen_values)

#Plot the eigenvalues using a histogram
plt.figure(figsize=(12,6))
plt.hist(eigen_values, bins=50)[2]
plt.title('Histogram of the Eigenvalues of the Hessian Matrix')
plt.xlabel('Magnitude of Eigenvalues')
plt.ylabel('Frequency Count')
```
executed in 461ms, finished 02:18:25 2021-12-05

Out[3]: `Text(0, 0.5, 'Frequency Count')`



(b) Implement gradient descent with constant step-size $\beta = 1/L$ (as in Problem 12.2), with random initial position x_0. Plot the log-error $\log(\|x_t - x_*\|)$ as a function of t.

In [4]:
```
#Create a function to return log errors
def log_error_func(xt,xopt):
    difference = xt - xopt
    return np.log(np.linalg.norm(difference))
```
executed in 13ms, finished 02:18:25 2021-12-05

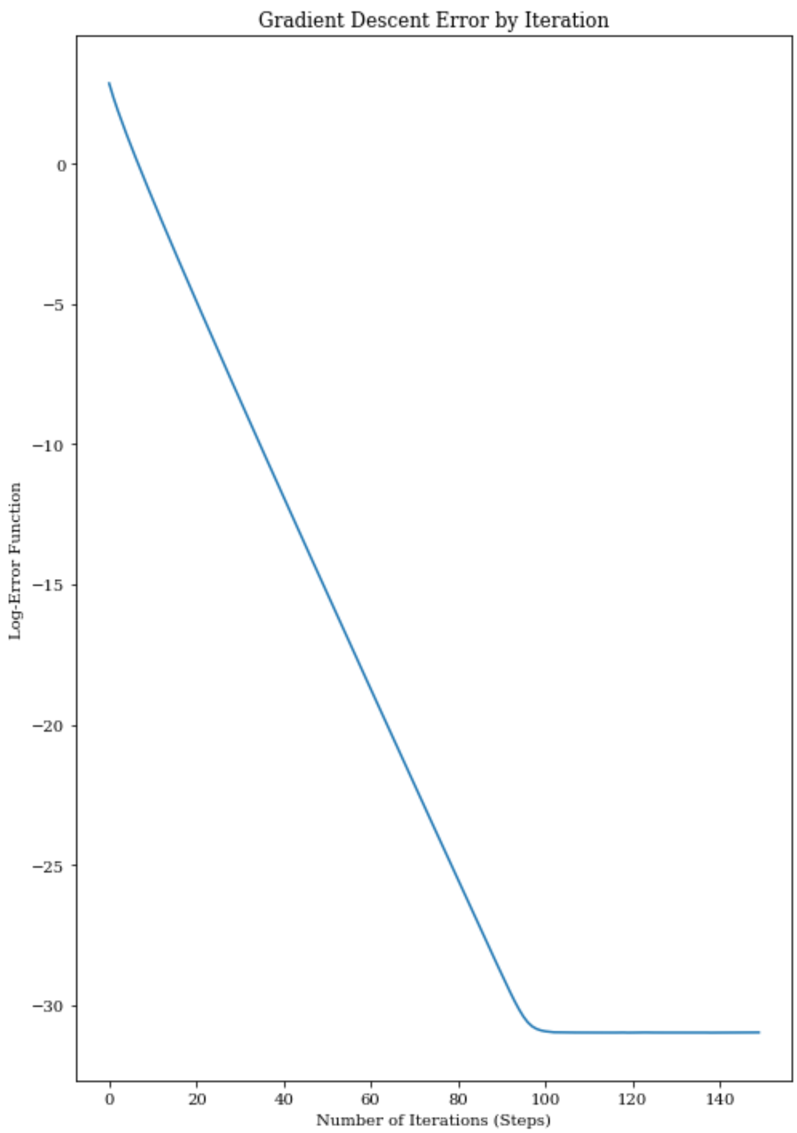
In [5]:
```
#Initialize a vairable to hold random position, create step size
random_position = np.random.normal(size=d)
step_size = 1/L
current_position = random_position

#Calculate optimal x* from part a
optimal_x = np.linalg.inv(hessian_matrix)@A.T@y
```
executed in 143ms, finished 02:18:25 2021-12-05

In [6]:
```
#Initialize variables for graph
iterations = []
log_error_arr = []

#Begin gradient descent
for i in range(150):
    current_position = current_position - ((1/L)*(hessian_matrix@current_position - A.T@y))
```

```
    iterations.append(i)
    log_error_arr.append(log_error_func(current_position, optimal_x))

plt.figure(figsize=(8,12))
plt.plot(iterations,log_error_arr)
plt.xlabel('Number of Iterations (Steps)')
plt.ylabel('Log-Error Function')
plt.title('Gradient Descent Error by Iteration')
```

executed in 316ms, finished 02:18:26 2021-12-05

Out[6]: `Text(0.5, 1.0, 'Gradient Descent Error by Iteration')`



(c) Implement gradient descent with momentum, with the same parameters as in Problem 12.4. Plot the log-error $\log(\|x_t - x_*\|)$ as a function of t, on the same plot than the log-error of gradient descent without momentum. On the same plot, plot also the lines of equation

$$y = \log(1 - \mu/L) \times t \quad \text{and} \quad y = \log\left(\frac{\sqrt{L} - \sqrt{\mu}}{\sqrt{L} + \sqrt{\mu}}\right) \times t. \tag{3}$$

In [7]:
```
#Define gradient_descent_with_momentum:
def gradient_with_momentum(beta, gamma, current, old):

    #Calcualte new position
    new = current - beta*(hessian_matrix@current - A.T@y) + gamma*(current - old)

    #Return the new position, and the t-1 position
    return new, current
```
executed in 13ms, finished 02:18:26 2021-12-05

In [8]:
```
#Define the bounds of each gradient descent approach
def momentum_bound_gd(arr):
    messy_part = np.log(((L**0.5)-(mu**0.5))/((mu**0.5)+(L**0.5)))
    return messy_part*np.array(arr)

def standard_gd_bound(arr):
    messy_part = np.log(1-(mu/L))
    return messy_part * np.array(arr)
```
executed in 13ms, finished 02:18:26 2021-12-05

In [9]:
```
#Initialize random position, current position, and t-1 position variables
starting_position = np.random.normal(size=d)
current_position = old_position = starting_position

#Get the optimal solution for x
optimal_x = np.linalg.inv(hessian_matrix)@A.T@y

#Calculate beta and gamma parameters from 12.4
beta = 4/(((L**0.5)+(mu**0.5))**2)
gamma = (((L**0.5)-(mu**0.5))/((L**0.5)+(mu**0.5)))**2
```
executed in 160ms, finished 02:18:26 2021-12-05

In [10]:
```
#Initialize variables to track iteration
iterations2 = []
log_error_arr2 = []
```
executed in 14ms, finished 02:18:26 2021-12-05

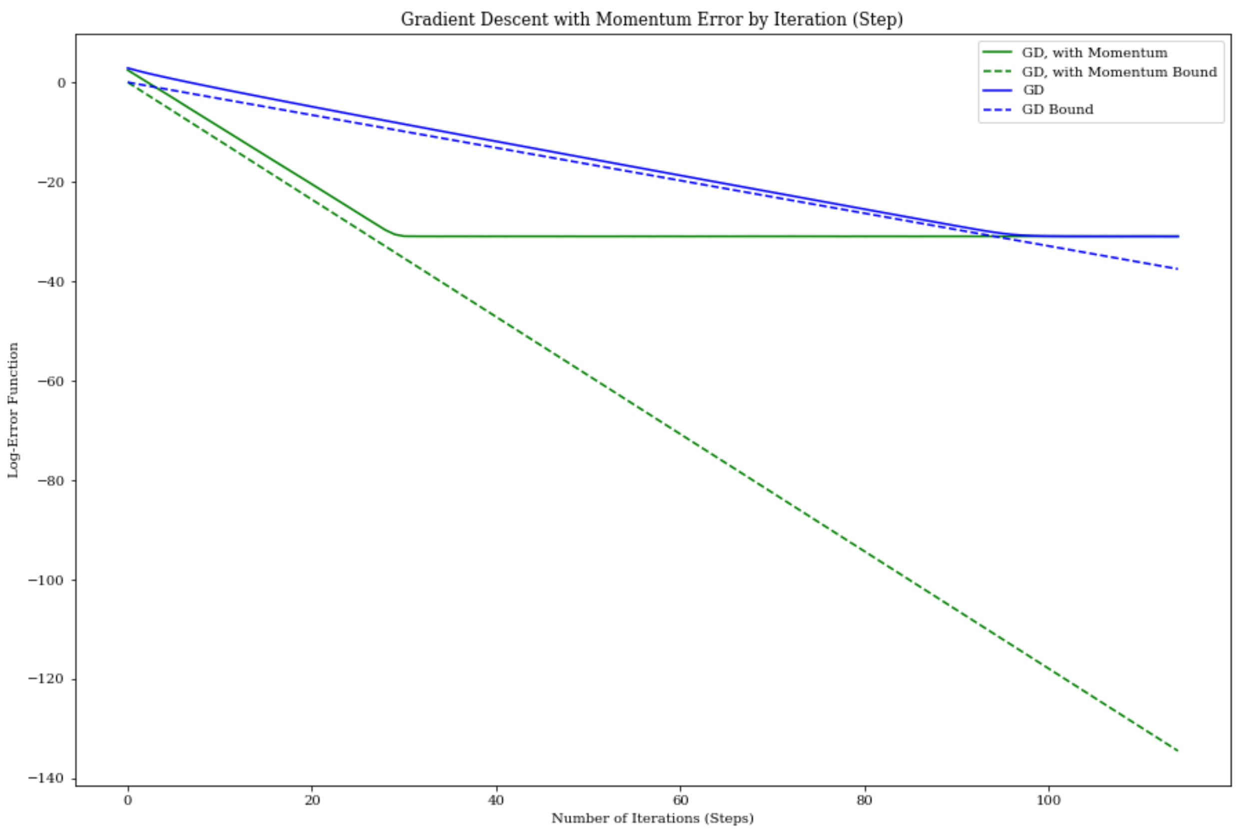
In [11]:
```
#Begin Iterating
for i in range(150):
    current_position, old_position = gradient_with_momentum(beta, gamma, current_position, ol
    iterations2.append(i)
    log_error_arr2.append(log_error_func(current_position, optimal_x))
```
executed in 190ms, finished 02:18:26 2021-12-05

In [12]:

```
#Plot Graphs
plt.figure(figsize=(15,10))
plt.plot(iterations2[:115],log_error_arr2[:115], c='g', label='GD, with Momentum')
plt.plot(iterations2[:115], momentum_bound_gd(iterations2[:115]),'g--',label='GD, with Moment
plt.plot(iterations[:115],log_error_arr[:115], c='b',label='GD')
plt.plot(iterations[:115], standard_gd_bound(iterations[:115]),'b--',label='GD Bound')

plt.xlabel('Number of Iterations (Steps)')
plt.title('Gradient Descent with Momentum Error by Iteration (Step)')
plt.ylabel('Log-Error Function')
plt.legend()
```

executed in 189ms, finished 02:18:26 2021-12-05

Out[12]: <matplotlib.legend.Legend at 0x21203a2d3c8>



Observation: Gradient Descent with Momentum approaches optimal solution nearly 3 times faster than normal gradient descent, which requires nearly 100 iterations to achieve optimal solution