# Quasi-Newton Optimization For Unconstrained Problems

Lachlan Moore
MANE 6710

2020
December

**Executive Summary**

This independent study aimed at understanding and implementing a quasi-Newton algorithm for unconstrained non-linear optimization. The BFGS algorithm was the focus due to it's popularity and robustness. Additionally, a line search satisfying the strong wolfe conditions was implemented. The line search and BFGS algorithm were verified for a series of 1, 2, and 3 dimensional problems. Minimums were achieved to a gradient tolerance of 1e-6. The initial goal was to apply these algorithms to an orbital optimization problem, but this was scaled back due to time constraints. This study was successful and can be a foundation for future work on unconstrained optimization.

## 1 Introduction

One of the basic forms of numerical optimization is the unconstrained non-linear problem. It asks the task of minimizing a given function with no bounds. It is a nice problem mathematically, though in practicality, there are few truly unconstrained engineering problems. Most problems are constrained by manufacturing, cost, time, size, etc. Nevertheless, the unconstrained non-linear problem is a useful place to begin the study of optimization.
A number of algorithms have been developed over the years for minimizing unconstrained problems. Famously there is Newton's methods, which requires a function be twice differentiable. Newton's method can converge in less iterations than other commonly used optimization algorithms, such as gradient descent. Additionally, there are quasi-Newton methods, which only require the gradient of the function. These quasi-Newton methods seek to create a good enough approximation of the objective function to produce superlinear convergence. Though convergence occurs in more iterations, in many cases quasi-Newton methods can be more efficient because of the lack of second derivative calculation.
There have emerged many examples of quasi-Newton algorithms including BFGS, SR1, DFP, and Broyden. These methods differ in the way that they approximate the Hessian of the objective function. Section 2 will discuss further the details of quasi-Newton algorithms, and the chosen algorithm for further study.

## 2 Quasi-Newton Optimization

Quasi-Newton methods are a family of solutions for the minimization of nonlinear local optimization problems of varying degrees, both constrained and unconstrained. They have become the most popular method for unconstrained optimization for their efficiency in solving high degree, complex problems.
The main idea behind quasi-Newton methods is they iteratively build an approximation of the Hessian of the objective function in order to compute the search direction. The iterative

process is outlined in the following steps. The step size computation can be done in a number of ways to satisfy different sets of conditions and will be discussed further in section 2.2.

1. Compute search direction
2. Determine step size from line search
3. Update initial variable guess with step
4. Update the Hessian approximation

These methods differ from Newton's Methods because they do not require a second derivative evaluation. This can lead to increased efficiency in computation. There are some downfalls when compared to a full Newton's method such as a less efficient search path and more convergence steps. These are normally overshadowed by the benefits from faster computation and the lack of a need for the second derivative.

## 2.1 The BFGS Method

One of the most widely used and popular quasi-Newton algorithms was developed by Broyden, Fletcher, Goldfarb, and Shanno, and is fittingly called the BFGS method.
The BFGS algorithm is very robust, and has superlinear convergence. As commented on in [2], the convergence rate is slower than Newton's method, though significantly faster than steepest descent. The convergence is sufficient for practical use.
This algorithm functions in the same way as discussed in section 2, and differs in its method of Hessian approximation. The BFGS method uses equation 2.1. A problem that occurs with this algorithm is how to choose a suitable initial guess for the Hessian. Unfortunately there isn't one correct answer, so the identity matrix is sufficient.

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.1}$$

The following algorithm is taken from Nocedal and Wright, *Numerical Optimization* and for further background on the full derivation, the reader can see chapter 6.1. [2]

**Algorithm 1:** The BFGS Method [2]

**Result:** Objective Function Local Minimum
Given starting point $x_0$, convergence tolerance $\epsilon > 0$, inverse Hessian approximation
k $\leftarrow 0$
**while** $||\nabla f_k|| > \epsilon$ **do**
  Compute search direction

$$p_k = -H_k \nabla f_k \tag{2.2}$$

  Set $x_{k+1} = \alpha_k p_k$ where $\alpha_k$ is computed from a line search procedure satisfying the Strong Wolfe conditions
  Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$
  Compute $H_{k+1}$ from equation (2.1)
  $k \leftarrow k + 1$
**end**

## 2.2 A Line Search Satisfying the Strong Wolfe Conditions

The Wolfe Conditions constrain the line search and are made up of two existing conditions, the Armijo (sufficient decrease) and the curvature condition. These are useful for stepping towards a minimizer but in order to satisfy the conditions, a point doesn't need to be necessarily close to the minimum. To speed up the search, the curvature conditions can be

modified, creating the Strong Wolfe Conditions. This ensures that the new point is sufficiently close to a minimizer or stationary point. The final set of conditions used is (2.3) and (2.5). A visual representation of these conditions are presented in figure 1.

**Armijo Condition**

$$\phi(0) \leq \phi(0) + c_1\phi'(0)\alpha \tag{2.3}$$

**Curvature Condition**

$$\phi'(0) \geq c_2\phi'(0) \tag{2.4}$$

**Strong Wolfe Curvature Condition**

$$|\phi'(0)| \leq c_2|\phi'(0)| \tag{2.5}$$

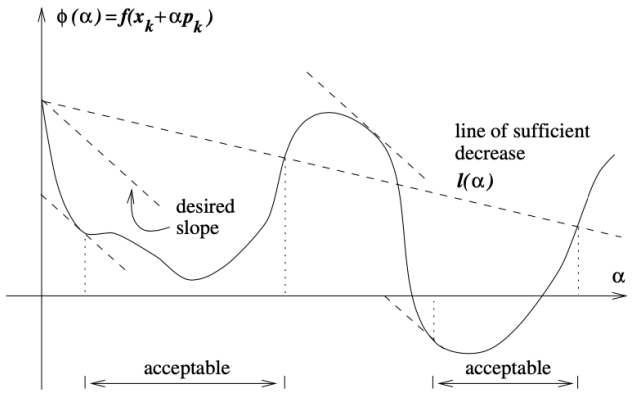



Figure 1: The Wolfe Conditions [2]

Algorithms 2 and 3 implement these conditions in an iterative search until a sufficient length is found. When being used within the BFGS algorithm, the initial guess for the line search will always be 1. For more information on the implementation and the derivations, the reader can see [2].

**Algorithm 2:** A Strong Wolfe Line Search [2]

**Result:** Step Length $\alpha$ Satisfying the Strong Wolfe Conditions
Set $\alpha_0 \leftarrow 0$, choose $\alpha_{\max} > 0$ and $\alpha_1 \in (0, \alpha_{\max})$
$i \leftarrow 1$
**repeat**
  Evaluate $\phi(\alpha_i)$
  **if** $\phi(\alpha_i) > \phi(0) + c_i\alpha_i\phi'(0)$ *or* $[\phi(\alpha_i \geq \phi(\alpha_{i-1}$ *and* $i > 1]$ **then**
    $\alpha_* \leftarrow$ **zoom**$(\alpha i - 1, \alpha_i)$ and **stop**
  **end**
  Evaluate $\phi'(\alpha_i)$
  **if** $|\phi(\alpha_i)| \leq -c_2\phi'(0)$ **then**
    **set** $\alpha_* \leftarrow \alpha_i$ and **stop**
  **end**
  **if** $\phi(\alpha_i) \geq 0$ **then**
    **set** $\alpha_* \leftarrow$ **zoom**$(\alpha i, \alpha_{i-1})$ and **stop**
  **end**
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$
  $i \leftarrow i + 1$

The *zoom* function seeks to find a step length that satisfies the sufficient decrease condition, if the condition fails within algorithm 2.

**Algorithm 3:** Zoom [2]

Interpolate using (quadratic, cubic, or bisection) to find a trial step length $\alpha_j$ between $\alpha_{lo}$ and $\alpha_{hi}$.
Evaluate $\phi(\alpha_j)$
**if** $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ *or* $\phi(\alpha_j) > \phi(\alpha_{lo})$ **then**
  $\alpha_{hi} \leftarrow \alpha_j$
**else**
  Evaluate $\phi'(\alpha_j)$
  **if** $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$ **then**
    Set $\alpha_* \leftarrow \alpha_j$ and **stop**
  **end**
  **if** $\phi'(\alpha_j)(\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
    $\alpha_{hi} - \alpha_{lo}$
  **end**
  $\alpha_{hi} \leftarrow \alpha_j$
**end**

# 3 Results and Discussion

## 3.1 Line Search Verification

The following section will discuss the verification of the line search using a set of one dimensional test problems. The line search was tested independently from the BFGS algorithm due to the complexity of the algorithm. One dimensional problems were the easiest way to observe the expected behavior and verify the algorithm. The following equation set was used.

$$y(x) = x^2 \tag{3.1}$$

$$y(x) = \sin(x) + \sin(10/3 * x) \tag{3.2}$$

Figure 2 shows the search for a step size on a one dimensional parabola, equation 3.1. This is a useful case due to the symmetry of the function. The search would be expected to perform exactly the same from an initial guess and the negative guess of the same magnitude. The figure displays an initial position of 0.25 and -0.25 accordingly. As expected, the behavior is symmetrical about the parabola, converging to the same final step size.

Table 1: Line Search Behavior, 1D Parabola

| | $x_0$ | $x_{\text{final}}$ | $\alpha_0$ | $\alpha_{\text{final}}$ |
|---|---|---|---|---|
| LHS | -0.25 | 0 | 0 | 0.25 |
| RHS | 0.25 | 0 | 0 | 0.25 |

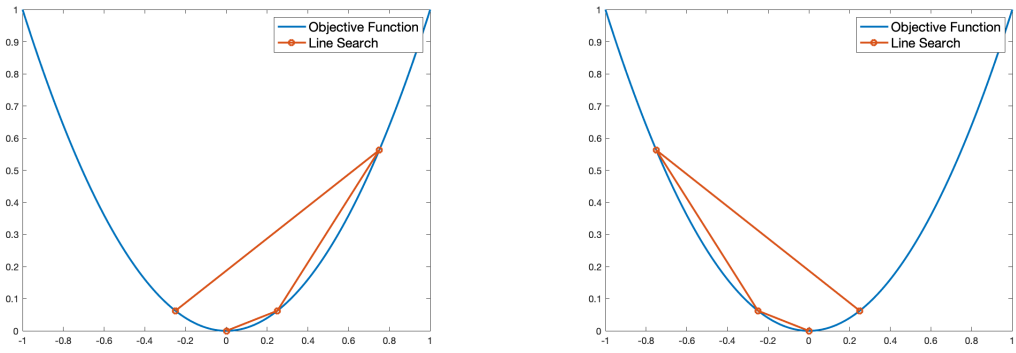
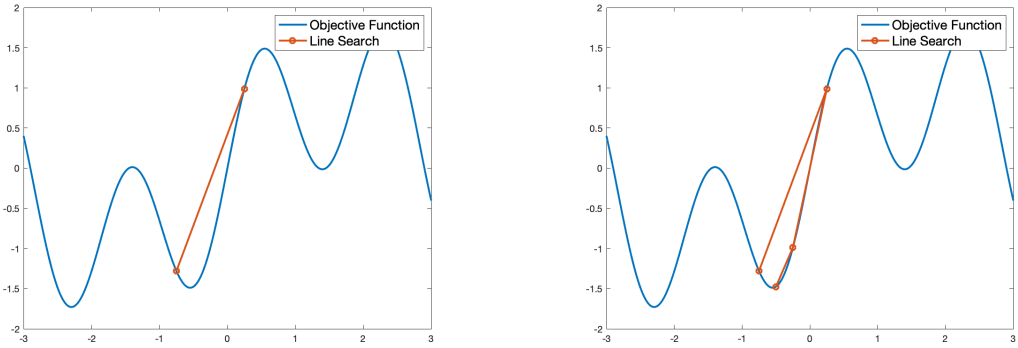



Figure 2: Line Search Behavior, 1D Parabola

Investigating a function that may not be symmetrical should show convergence to different step lengths and positions based on the initial position. Figure 3 shows the search for a step length on a non-symmetric trigonometric function, equation 3.2, with two iterations exhibited. The first valid step is then returned into the search as the next initial position. This iterative process is how the line search is used in the minimization algorithm. The search should improve upon the initial location with each iteration. That behavior is seen here.

An interesting behavior is also seen in this example. With the initial starting point of 0.25, the initial step length of 1 satisfies the conditions and the search concludes. When this is returned back into the search, the initial step length returns the search back to the beginning of iteration 1, though this time, the position doesn't satisfy the conditions and the search proceeds, moving down closer to the local minimum.




(a) Iteration 1

(b) Iteration 2

Figure 3: Iterative Line Search

### 3.2 Multi-Dimensional Quasi-Newton Verification

The following section will cover a set of multi-dimensional test problems to verify the BFGS quasi-Newton algorithm in conjunction with the line search discussed in section 2.2. The test set contains 2 two-dimensional problems and 1 three-dimensional. These functions increase in complexity which is useful for finding limitations of both the algorithms and computing power. Additionally these functions have known local and global minimum which help for verifying the accuracy for the optimization.

**Three Hump Camel Function**

$$f(x) = 2x_1^2 - 1.05x_1^4 + x_1^6/6 + x_1x_2 + x_2^2 \tag{3.3}$$

Global Minimum: $f(x^*) = 0, x^* = [0, 0]$

**Easom Function**

$$f(x) = -\cos(x_1)\cos(x_2)\exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2) \tag{3.4}$$

Global Minimum: $f(x^*) = -1, x^* = [\pi, \pi]$

**Hartmann 3-Dimensional Function**

$$f(x) = -\sum_{i=1}^{4} \alpha_i \exp(-\sum_{j=1}^{3} A_{ij}(x_j - P_{ij})^2) \tag{3.5}$$

$$\alpha = (1.0, 1.2, 3.0, 3.2)^T$$

$$A = \begin{pmatrix} 3.0 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3.0 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix}$$

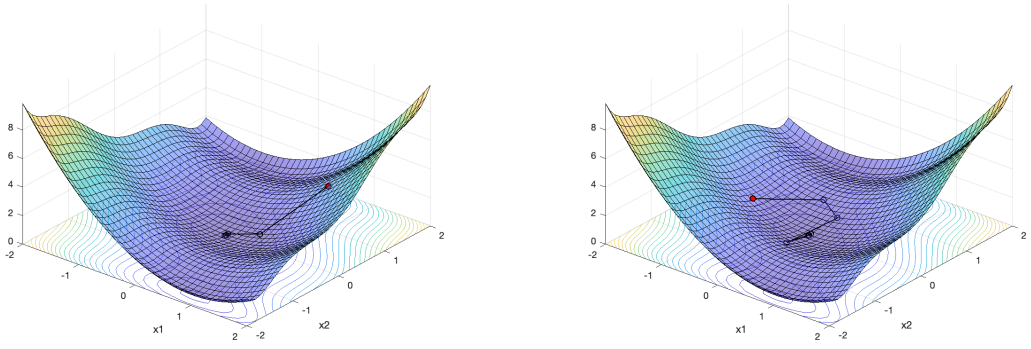
$$P = 10^{-4} \begin{pmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{pmatrix}$$

Global Minimum: $f(x^*) = -3.86278, x^* = [0.114614, 0.555649, 0.852547]$

This function set was obtained from Simon Fraser University, Library of Simulation Experiments [1].

### 3.2.1 Three Hump Camel Function: 2D

The first test function is the Three Hump Camel, equation 3.3. There are a number of local minima and it is usually evaluated in the domain [-5,5]. In order to better show the function behavior near the minimums, here it is evaluated on [-2,2]. Two initial guesses were run to verify convergence to the same point via different routes.




(a) Initial x=(1,1)

(b) Initial x=(-1,0)

Figure 4: Three Hump Camel Global Minimum Convergence

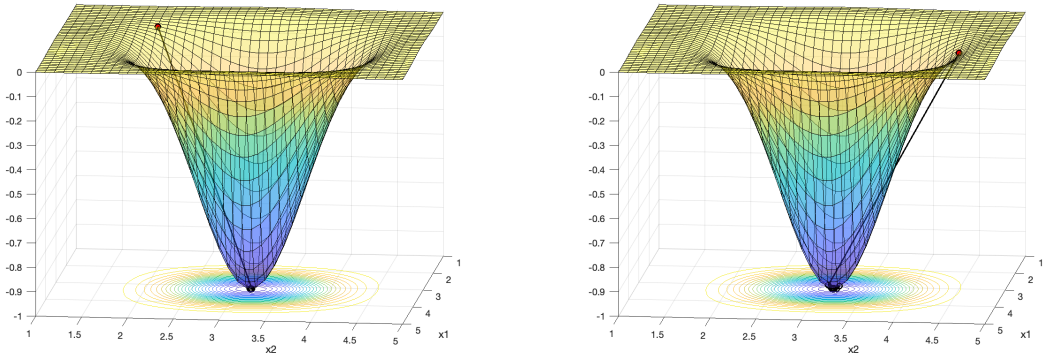
Table 2 shows the convergence behavior and results from the two displayed test cases, from different starting positions.

Table 2: Three Hump Camel Convergence

| | Iterations | Final Gradient | f(x) | Local Minimum |
|---|---|---|---|---|
| (1,1) | 14 | 5.936538e-08 | 7.610e-14 | (1.008e-08, 1.039e-08) |
| (-1,0) | 11 | 7.501148e-07 | 4.159e-16 | (8.849e-08, 2.055e-07 ) |

### 3.2.2 Easom Function: 2D

The second test case is the Easom 2D function. This is a steep drop to a minimum within a global flat plane. This function is usually evaluated on the square [-100, 100], but is zoomed in for better visualization. Again, the starting location was tested from multiple locations, each time converging to the same minimum in approximately the same number of iterations. The convergence history is shown in figure 5 and table 3.




(a) Initial x=(2,2)

(b) Initial x=(3,4.5)

Figure 5: Easom Function Global Minimum Convergence

Table 3: Easom Function Convergence

| | Iterations | Final Gradient | f(x) | Local Minimum |
|---|---|---|---|---|
| (2,2) | 10 | 3.404e-07 | -1 | (3.141593, 3.141593) |
| (3,4.5) | 12 | 5.436e-07 | -1 | (3.141593, 3.141593) |

### 3.2.3 Hartmann 3-Dimensional Function

The final test case, of a higher dimensional function, was the hartmann 3D function. This is usually evaluated on the hypercube $x_i \in (0, 1)$ for i = 1,2,3. This demonstrates the algorithm works for higher order functions, not capable of being visualized.

Table 4: Hartmann 3-Dimensional Convergence

| | Iterations | Final Gradient | f(x) | Local Minimum |
|---|---|---|---|---|
| (1,1,1) | 17 | 5.273662e-07 | -3.863 | (0.11458, 0.55565, 0.85254) |
| (0.5,1,0.75) | 23 | 5.669580e-07 | -3.863 | (0.11458, 0.55565, 0.85254) |

# 4 Discussion of Results and Conclusion

The goal of this independent study was researching and implementing a working version of the BFGS quasi-Newton minimization algorithm, and an associated strong wolfe line search. This is one of the most popular algorithms for use in unconstrained optimization. The implementation was successful and was capable of obtaining a local minimum for multi-dimensional functions with a tolerance of 1e-6. For practical purposes this is a small enough tolerance to ensure a minimum was truly reached.
Future work would aim to compare this algorithm to other existing quasi-Newton algorithms, observing the accuracy and cost. An orbital optimization problem would also be revisited using the presented algorithms. Additional comparison could be made to other optimization methods such as Newton's method or steepest descent, verifying existing research.

## References


[1] Derek Bingham. *Virtual Library of Simulation Experiments:* 2013. URL: https://www.sfu.ca/~ssurjano/index.html.

[2] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2nd. New York, NY, USA: Springer, 2006.

# A MATLAB Code

All code can additionally be found at: lachlanmoore/QuasiNewtonOptimization.

Listing 1: The BFGS Algorithm

```
function bfgs(xk, n, fun_input)
% bfgs(xk, n, fun_input)
%
% A BFGS quasi-Newton optimization algorithm for unconstrained minimization
%
% -----------------
% Inputs:
% xk: Initial guess
% n: Number of design variables
% fun_input: Objective function
% ------------------
%
% Taken from "Numerical Optimization, Nocedal and Wright, 2006"
% Algorithm 6.1
%
% Lachlan Moore
% 2020 December

close all

H  = eye(n);

func     = @(x) fun_input(x);

[func_eval, func_deriv] = func(xk);

i = 1;
iter = 300;
eps = 1e-6;


while norm(func_deriv) > eps

    pk     = -H * func_deriv; % search direction
    pk     = pk/norm(pk);

    alpha = 1; %compute from search direction, will be line search
    alpha = line_search(xk, pk, alpha);
    xk_new = xk + alpha * pk;

    [func_eval_new, func_deriv_new] = func(xk_new);

    sk     = xk_new - xk;
    yk     = func_deriv_new - func_deriv;

    rk     = 1/(yk'*sk);
    H      = (eye(n) - rk*(sk*yk'))*H*(eye(n) - rk*(yk*sk')) + rk*(sk*sk');

    xk     = xk_new;
    func_eval = func_eval_new;
    func_deriv = func_deriv_new;


    i      = i + 1;
    if i >= iter
        break
    end

end

fprintf('Final Evaluation %d\n', func_eval)
fprintf('Iterations: %d\n', i-1)
fprintf('Gradient %d\n',norm(func_deriv))
fprintf('Local Minimum at \n')
fprintf('%c \n', xk)
disp(H)

end
```

Listing 2: A Strong Wolfe Line Search

```
function [alphastar] = line_search(x, dir, alpha1)
% [alphastar] = line_search(x, dir, alpha1)
%
% A line search algorithm that satisfies the strong wolfe conditions
%
% -----------------
% Inputs:
% x: Initial starting location
% dir: Search direction
% alpha1: Initial step length
%
% Outputs:
% alphastar: Step length meeting the strong wolfe conditions
% -----------------
%
% Taken from "Numerical Optimization, Nocedal and Wright, 2006"
% Algorithms 3.5, 3.6
%
% Lachlan Moore
% 2020 December


    a0 = 0;
    a1 = alpha1;
    amax = 10*a1;

    c1 = 1e-4;
    c2 = .9;
    i = 1;

    [fun0, grad0]   = obj(x);
    [fold, ~] = obj(x+a0*dir);

    d = 1;
    while 1

        if d >= 1000
               error('Runtime too long')
        end

        [fun1, grad1]  = obj(x+a1*dir);

        if (fun1 > (fun0 + c1*a1*grad0'*dir)) || ((fun1 > fold) && (i > 1)) %Violates sufficient decrease
           alphastar = zoom(a0, a1);
           break
        end

        if abs(grad1'*dir) <= -c2*grad0'*dir %Sufficient Decrease
            alphastar = a1;
            break
        end
        if grad1'*dir >= 0 %Curvature condition
            alphastar = zoom(a1, a0);
            break
        end

        a0 = a1;
        a1 = (a0 + amax )/2;
        i = i+1;
        fold = fun1;
        d = d+1;

    end


    function [alphastar] = zoom(alphalo, alphahi)
        % [alphastar] = zoom(alphalo, alphahi)
        %
        % Inputs:
        % alphalo, alphahi: Bounds of search location
        %
        % Output:
        % alphastar: Step length meeting stong wolfe conditions

        k = 1;
        while 1
            if k >= 5000
                error('Runtime too long')
            end

            alphaj = (alphalo+alphahi)/2;
            [funj, gradj] = obj(x+alphaj*dir);
            [funlo, ~] = obj(x+alphalo*dir);

            if (funj > fun0 + c1*alphaj*grad0'*dir) || (funj >= funlo)
               alphahi = alphaj;
            else
                if abs(gradj'*dir) <= -c2 * grad0'*dir
                    alphastar = alphaj;
                    return
                end

                if gradj'*dir*(alphahi-alphalo) >= 0
                    alphahi = alphalo;
                end
                alphalo = alphaj;
            end
```

```
            k = k+1;
        end
    end

end
```

Listing 3: Objective Function and Gradient Evaluation

```
function [f, g] = obj(x)
% [f, g] = obj(x)
%
% Objective function and gradient evaluation
%
% -----------------
% Inputs:
% x: Evaluation location
%
% Outputs:
% f: function eval
% g: function gradient, complex step approximation
% ------------------
%
% Lachlan Moore
% 2020 December




h = 1e-60; % complex step
f = sub(x);
g = zeros(length(x), 1);

for i = 1:length(x)

   xc = x;
   xc(i) = complex(xc(i), h); % complex step
   g(i) = imag(sub(xc)/h);  % complex step

end


    function [val] = sub(xc)
     % [val] = sub(xc)
     % Objective Function Definition


%% Line Search Test Cases
%       val = xc(1)^2;                                  % 1D Test
%       val = xc(1)*sin(xc(1)) + xc(1)*cos(2*xc(1)); % 1D Test
%       val = sin(xc(1)) + sin(10/3*xc(1));         % 1D Test

%% Final Project Test Cases
     val = -cos(xc(1))*cos(xc(2))*exp(-(xc(1)-pi)^2-(xc(2)-pi)^2); %Easom Function 2D
%       val = hartmann3(xc);                                    %Hartman 3D
%       val = camel3hump(xc);                                   %3 Hump Camel Function 2D

    end
end
```

Listing 4: 3 Hump Camel Function

```
function [y] = camel3hump(x)
% [y] = camel3(x)
%
% 3 Hump Camel Function
%
% Lachlan Moore
% 2020 December

x1 = x(1);
x2 = x(2);

y = 2*x1^2 + -1.05*x1^4 + x1^6 / 6 + x1*x2 + x2^2;

end
```

Listing 5: Hartmann 3D Function

```
function [y] = hartmann3(x)
% [y] = hartmann3(x)
%
% 3 Hump Camel Function
%
% Lachlan Moore
% 2020 December

alpha = [1.0; 1.2; 3.0; 3.2];

A = [3, 10, 30;
    0.1, 10, 35;
    3, 10, 30;
    0.1, 10, 35];

P = 1e-4 * [3689, 1170, 2673;
             4699, 4387, 7470;
             1091, 8732, 5547;
             381, 5743, 8828];

y = 0;
for i = 1:4
   si = 0;
   for j = 1:3
       si = si + A(i,j)*(x(j) - P(i,j))^2;
   end
   si = -1 * si;
   y = y + alpha(i) * exp(si);
end
y = -1 * y;

end
```