Optimization Assignment 2
Computation Effort Comparison between Zero-Order, First-Order, and Second-Order Optimization Algorithms

Kim Paolo Laberinto (7771083)

Winter 2021

Contents

1 Methodology

In this report several optimization algorithms were run on the 5D Rosenbrock function, with various initial guess vectors. The initial guess vectors used in this report can be in Table 1.

	Vector
Initial Vector 1	[0.00, 0.00, 0.00, 0.00, 0.00]
Initial Vector 2	[0.36, 1.18, -1.01, -1.73, -0.90]
Initial Vector 3	[1.07, 1.42, 0.32, 1.83, 0.61]
Initial Vector 4	[0.26, -1.20, 0.60, 0.59, -1.77]
Initial Vector 5	[-0.16, -0.81, -1.96, -1.55, 1.37]

Table 1: Table of Initial Vectors used for analysing performance

The equation for the 5D Rosenbrock function is below. For some algorithms, the gradient and hessian of the 5D Rosenbrock function were also required. Autodifferentiation techniques using ForwardDiff.jl were used to get the gradient and hessian.

$$f(x) = \sum_{i=1}^{5-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \tag{1}$$

Each of the algorithms were measured using different metrics such as:

- Number of Function Evaluations
- Number of Gradient Function Evaluations
- Number of Hessian Function Evaluations
- Number of Linear System Solves (e.g. inversions and similar)
- Run-time
- Percentage Time Spent in Garbage Collection

The remainder of this report showcases the various Loss vs Iteration plots for each of the algorithms with a short commentary. The final concluding sections of this report show the metrics measured on each algorithm.

1.1 Limitations

Note that this report is solely for educational purposes only in possible methods of analysis, and not meant to be used to rigorously compare the various algorithms.

For example, timing measurements in Table 3 also include the time for recording trial data/metrics such as loss vs iterations and saving hessian matrices for further analysis. These things may have a significant impact on memory allocation and run-time. In this report, these effects are not excluded.

Performance is also highly dependent on the implementation of the algorithm itself. The algorithm implementations written by the author used in this report are not optimized for time or memory usage.

1.2 Further Areas to Explore

There are more opportunities in analyzing these algorithms. Potential areas of analysis include:

- Optimizing for memory use and allocation to see its impact on performace
- Profiling the code to examine which lines run the longest

- Controling and comparing the hyperparameters across algorithms (gradient tolerences, max iterations, etc.)
- Increasing the number of parameters and solving much bigger optimization problems

2 Algorithm Performances - Loss vs Iterations

In this section, the Loss vs Iterations plots for each algorithm can be seen. Short commentaries can be found in the subsections below discussing the results seen in the plots.

2.1 Steepest Descent

As seen in Fig. 1, this naive technique took a significant amount of iterations to reach the stopping condition $\|\nabla f\| = 10^{-4}$. The author hypothesizes that this may be due to a "zig-zag" descent pattern in the steep valleys of the objective function. Each iteration in the gradient descent only lowers the loss by a small amount (relative to the other more efficient algorithms below).

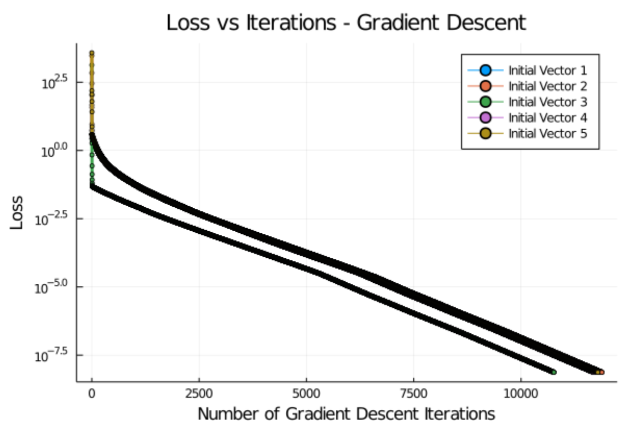

Figure 1: Gradient Descent Loss vs Iterations Plot

2.2 Powell's Conjugate Direction

Powell's Conjugate Direction on Initial Vector 2 had to abort due to facing a Case 4: Non-Unimodal error in the Swann's Bracketing Method algorithm as discussed in the notes/pseudocode. The implementation aborted, leaving the run for Initial Vector 2 with only a very short run.

The other runs with the other initial vectors stopped without any errors upon reaching the stopping condition specified in the code.

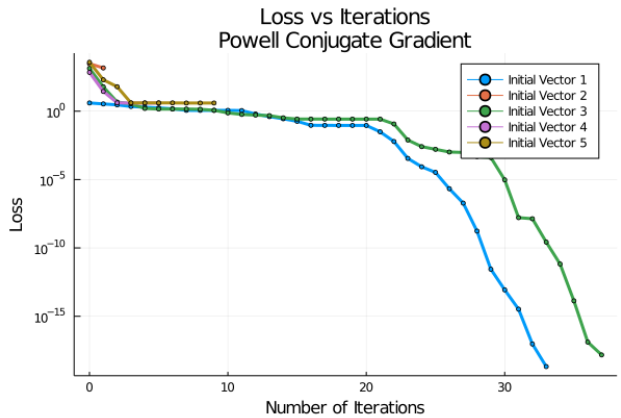

Figure 2: Powells Conjugate Directions Loss vs Iterations

2.3 Conjugate Gradient Techniques

As seen in Fig. 3, the Fletcher-Reeves Conjugate Gradient algorithm reached a loss in the order of 10^{-10} to 10^{-20} to the stopping condition $\|\nabla f\| = 10^{-4}$ in only 40 - 80 iterations. Whereas the Hestenes-Stiefel and Polak-Ribière variations of the Conjugate Gradient method took much longer to reach the same stopping condition.

2.3.1 Fletcher-Reeves

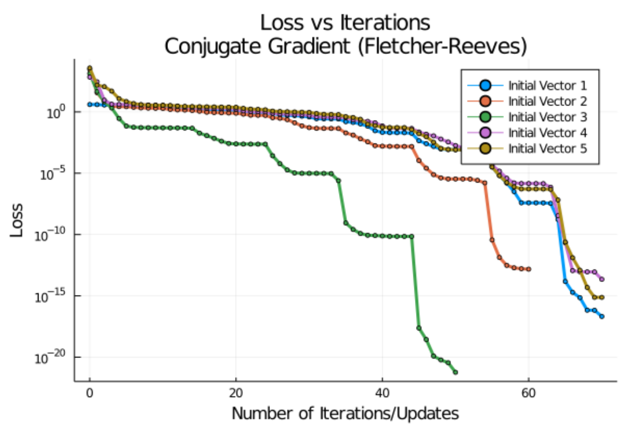

Figure 3: Fletcher-Reeves Conjugate Gradient - Loss vs Iterations

2.3.2 Hestenes-Stiefel

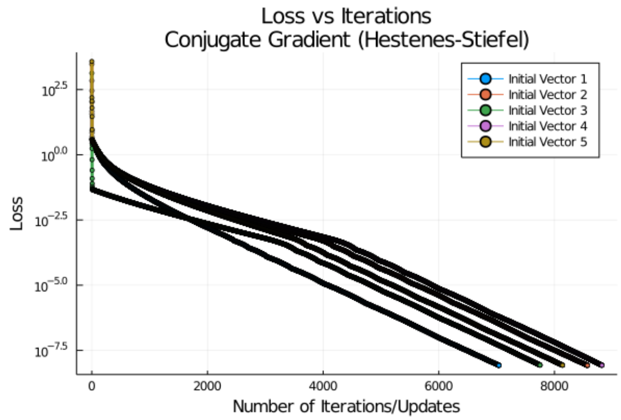

Figure 4: Hestenes-Stiefel Conjugate Gradient - Loss vs Iterations

2.3.3 Polak-Ribière

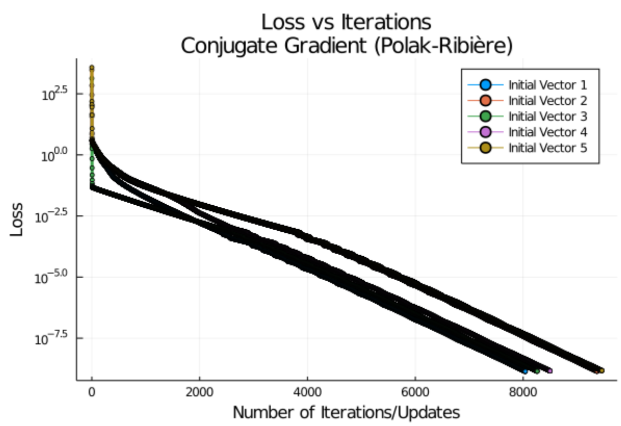

Figure 5: Polak-Ribière Conjugate Gradient - Loss vs Iterations

2.4 Hooke-Jeeves Direct Search

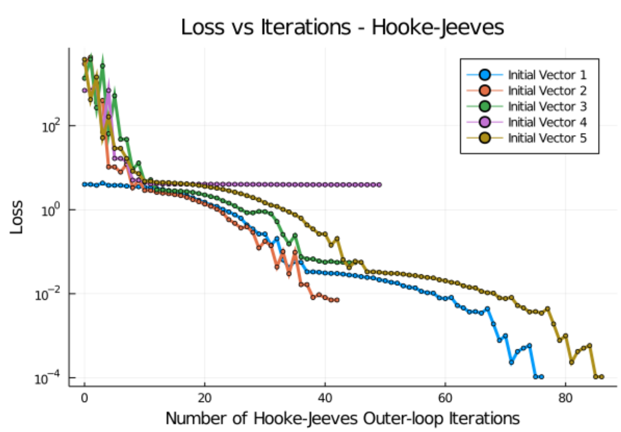

Figure 6: Hooke-Jeeves - Loss vs Iterations

2.5 Nelder-Mead Simplex Search

The stopping condition for this implementation of Nelder-Mead was to reach 500 iterations. As seen in Fig. 7, after 500 iterations, a loss of 10^1 to 10^{-3} was achieved. This is relatively inefficient algorithm in terms of amount of iterations and lowest loss achieved compared to the other algorithms in this report.

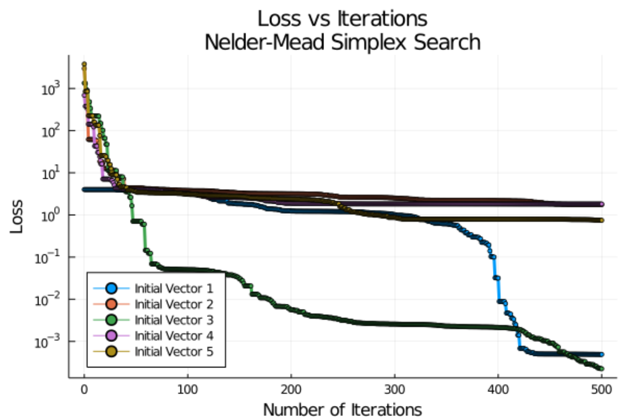

Figure 7: Nelder-Mead - Loss vs Iterations

2.6 Original Newton's Method

The update equation for Newton's Method is seen in the equation below.

$$x_{k+1} = x_k - H_k^{-1} g_k \tag{2}$$

The stopping condition for this implementation of the Original Newton's Method is reaching a $\|\nabla f\| = 10^{-3}$.

As seen in Fig. 8, all initial starting vectors reached a loss of 10^{-10} to 10^{-25} in 25 iterations or less. This shows an efficient algorithm in terms of iterations.

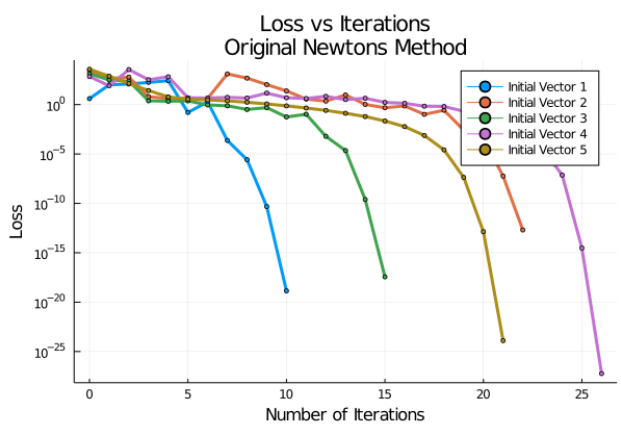

Figure 8: Original Newtons Method - Loss vs Iterations

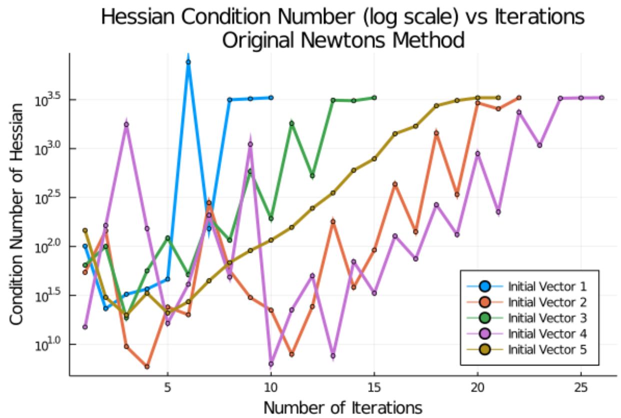

Figure 9: Original Newtons Method - Condition Number of Hessian vs Iteration

2.7 Modified Newton's Method with Levenberg-Marquardt Modification

The algorithm in this section uses the following update equation.

$$x_{k+1} = x_k - \alpha(H_k + \mu I)^{-1} g_k \tag{3}$$

Where α_k was determined from a line search. i.e.

$$\alpha_k = \text{argmin}_\alpha f(x_k + \alpha d_k) \tag{4}$$

In this section different mu values were analyzed.

- $\mu = 0.0$
- $\mu = 1.0$
- $\mu = 10.0$

2.7.1 Loss vs Iterations

As one can see in this section, $\mu = 0.0$ performed the best out of the other μ parameters. The author hypothesizes that this is due to the fact that the 5-D Rosenbrock function contains many valleys where gradient descent can zig-zag and get stuck on.

The $\mu = 10.0$ case looks similar to the Gradient Descent method above in Fig 1, with a slow descent and taking many iterations. The author hypothesizes that this is due to the common "zig-zag" effect with Gradient Descent methods.

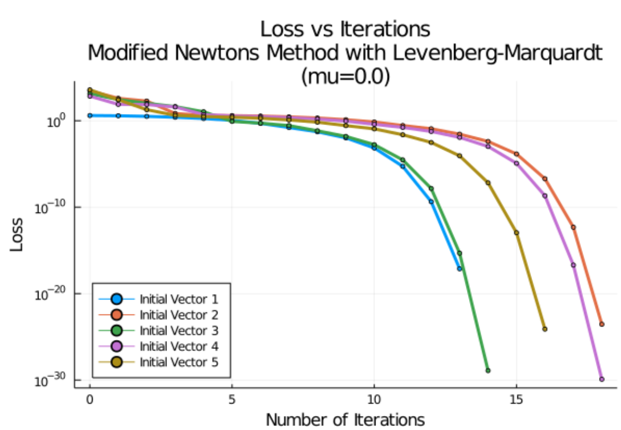

Figure 10: Modified Newton's method - Loss vs Iterations (mu = 0.0)

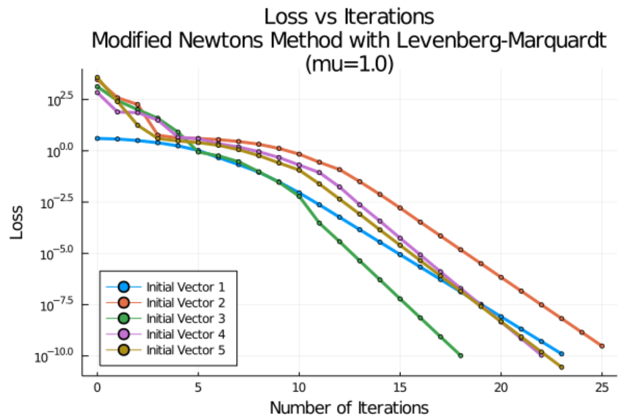

Figure 11: Modified Newton's method - Loss vs Iterations (mu = 1.0)

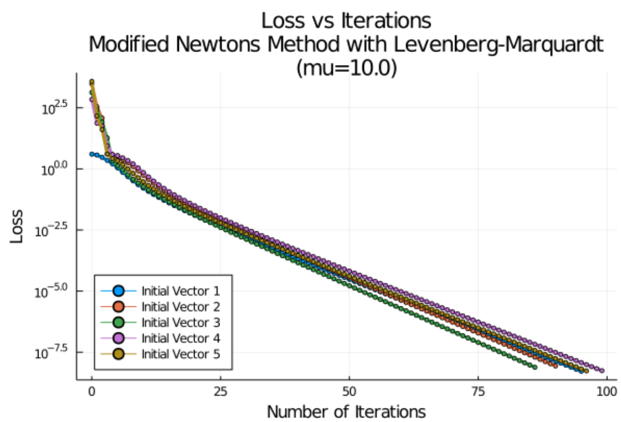

Figure 12: Modified Newton's method - Loss vs Iterations (mu = 10.0)

2.7.2 Condition Number of LM-Matrix

In this section, the LM-Matrix denotes the $(H_k + \mu I)$ i.e the sum of the Hessian Matrix with the diagonal μ-parameter matrix.

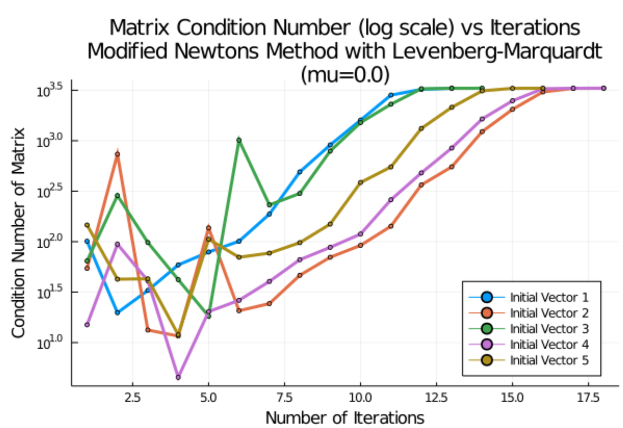

Figure 13: Condition Number of LM-Matrix vs Iterations (mu = 0.0)

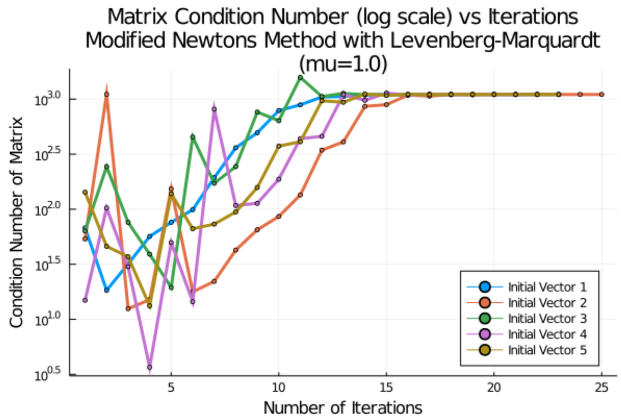

Figure 14: Condition Number of LM-Matrix vs Iterations (mu = 1.0)

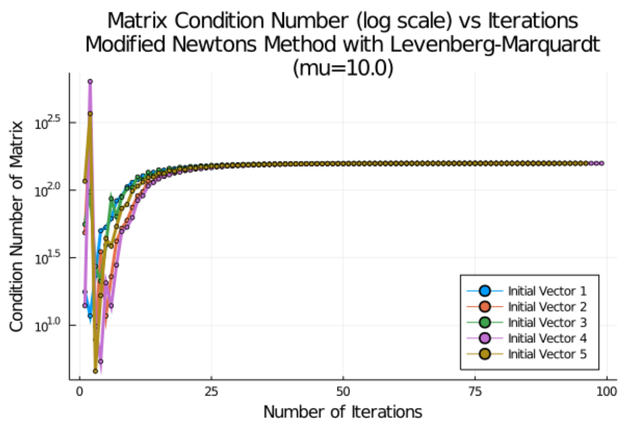

Figure 15: Condition Number of LM-Matrix vs Iterations (mu = 10.0)

3 Metric Comparisons between Algorithms

Table 2 contains a summary of all the metric performances of all the algorithms for Initial Vector 1. Table 3 contains the runtime measured for each algorithm on Initial Vector 1. Multiple samples were taken. Only one initital vector was used for this section with Metrics and Timings.

	Count f evals*	Count grad_f evals	Count hessian_f evals	Count inverse solves	Final Loss
Steepest/Gradient Descent	330625	11809	0	0	7.50E-9
Powell's Conjugate Direction	5111	0	0	0	2.09E-19
Fletcher-Reeves Conjugate Gradient	1	2047	0	0	2.08E-17
Hestenes-Stiefel Conjugate Gradient	1	30991	0	0	8.40E-9
Polak-Ribière Conjugate Gradient	1	34976	0	0	1.38E-9
Hooke-Jeeves Direct Search	1176	0	0	0	1.05E-4
Nelder-Mead Simplex Search	3501	0	0	0	4.91E-4
Original Newtons Method	1	11	10	10	1.39E-19
Modified Newton with LM (mu = 0.0)	1	97	13	13	7.44E-18
Modified Newton with LM (mu = 1.0)	1	141	23	23	1.25E-10
Modified Newton with LM (mu = 10.0)	1	462	95	95	5.28E-9

Table 2: Metric Comparison for all Algorithms for Initial Vector 1. Note that all algorithms has one additional f-evaluation performed to determine the final loss.

	Min. Time	Min. Time % GC	Median Time	Median Time % GC	Max. Time	Max. Time % GC	Final Loss
Steepest/Gradient Descent	4.110 s	11.42%	4.112 s	11.58%	4.115 s	11.73%	7.50E-9
Powell's Conjugate Direction	57.033 ms	0.00%	66.822 ms	12.67%	69.982 ms	14.78%	2.09E-19
Fletcher-Reeves Conjugate Gradient	4.059 ms	0.00%	4.111 ms	0.00%	11.548 ms	62.42%	2.08E-17
Hestenes-Stiefel Conjugate Gradient	127.564 ms	6.78%	132.510 ms	8.64%	143.174 ms	15.61%	8.40E-9
Polak-Ribiere Conjugate Gradient	145.596 ms	5.95%	151.075 ms	8.03%	161.783 ms	13.55%	1.38E-9
Hooke-Jeeves Direct Search	6.710 ms	0.00%	7.305 ms	0.00%	15.919 ms	52.91%	1.05E-4
Nelder-Mead Simplex Search	1.096 ms	0.00%	1.173 ms	0.00%	7.342 ms	81.75%	4.91E-4
Original Newtons Method	63.703 us	0.00%	67.867 us	0.00%	5.439 ms	96.89%	1.39E-19
Modified Newton with LM (mu = 0.0)	329.212 us	0.00%	341.094 us	0.00%	7.433 ms	94.53%	7.44E-18
Modified Newton with LM (mu = 1.0)	522.033 us	0.00%	539.425 us	0.00%	7.402 ms	90.66%	1.25E-10
Modified Newton with LM (mu = 10.0)	1.924 ms	0.00%	2.000 ms	0.00%	14.185 ms	81.14%	5.28E-9

Table 3: Total Run-time for each algorithm as measured by the BenchmarkTools.jl library. Multiple samples were taken. Initial Vector 1 used as input. GC stands for Garbage Collection.

There are several interesting observations to be made from Table 2 and Table 3. These are summarized below in point-form.

- The algorithms which were the fastest (i.e lowest run time) and most effective (i.e. lowest final loss) were "Original Newtons Method", "Modified Newton with LM (mu = 0.0)", and "Modified Newton with LM (mu = 1.0)".
  - These algorithms achieved run times of under 1 milisecond.
  - These algorithms achieved the lowest number of f, ∇f and H_f evaluations compared to the other algorithms.
  - These algorithms used second-order (hessian) information.
  - These algorithms also used inverse solves (i.e. solving for x in $Ax = b$)
  - Not all problems can be effectively represented with gradients and hessians. This limits the use of such effective and efficient first order and second order methods.
  - Furthermore, for some problems with very large number of parameters, inverse solving unstructured (non-sparse) large matrices can be a very difficult and take a long time.
  - Fortunately for this problem with the 5D Rosenbrock objective function, inverse-solving a 5x5 matrix problem is fast. These inverses also made significant reduction in the loss function.

- "Modified Newton with LM (mu = 10.0)" was poorly tuned with $mu = 10.0$, however was informative to how the method approaches the behavior of gradient descent as mu increases.
- "Powell's Conjugate Direction" and "Fletcher-Reeves Conjugate Gradient" methods also did well, achieving losses of under 10^{-10} similar to the second-order techniques.
  - However, Powell's Conjugate Direction and Fletcher-Reeves Conjugate Gradient was not able to achieve fast (under 1ms) run-times.
- Hestenes-Stiefel and Polak-Ribiere used a relatively large amount of ∇f function evaluations compared to Fletcher-Reeves despite all of them also being Conjugate Gradient techniques.
- "Hooke-Jeeves Direct Search" and "Nelder-Mead Simplex Search" both achieved similar final loss errors in the order of 10^{-4} with a similar amount of function evaluations. These in general performed faster, but achieved a higher loss compared to Powell's Conjugate Direction.
- Steepest/Gradient Descent took the longest run-time and took the largest amount of function evaluations.

4 Conclusion

As seen in the results from this report, second-order methods such as Modified Newton with LM and Original Newtons Method can achieve much lower loss, with faster run times and with an lower amount of overall function evaluations. However, second-order methods may not always be the best algorithm to use in all problems.

There are several problems facing being able to effectively second-order methods for other optimization problems.

- There might not be a nice representations of their gradients and hessians to be calculated by a computer.
- Inverse-solving the matrix equation for very large problems may be infeasible even if gradients and hessians are found.

For first-order methods, the Conjugate Gradient methods worked much better than Steepest-Descent. In this report, with 5D Rosenbrock and the initial starting vectors, Fletcher-Reeves update equation worked best compared to Hestenes-Stiefel and Polak-Ribiere.

For zero-order methods as analyzed in this report, Powell's Conjugate Direction algorithm achieved very low losses, however took longer to run with more function evaluations. Whereas with the implementations of Hooke-Jeeves, and Nelder-Mead had lower run-times but were not able to achieve similarly low losses compared to Powell's Conjugate Direction.

Overall, there exist many different optimization methods. Each optimization method has its pros and cons and the best algorithm to use is dependent on the situation and known information surrounding the objective function.

A Source Code

A.1 main.jl

```
include("makeplots.jl")

println("\nGradient Descent")
@time evaluateGradientDescent()

println("\nPowell Conjugate Gradient Descent")
@time evaluatePowellConjugateGradient()

println("\nConjugate Gradient (Fletcher-Reeves)")
@time evaluateConjugateGradientFletcherReeves()

println("\nConjugate Gradient (Hestenes-Stiefel)")
@time evaluateConjugateGradientHestenesStiefel()

println("\nConjugate Gradient (Polak-Ribiere)")
@time evaluateConjugateGradientPolakRibiere()

println("\nHookeJeeves")
@time evaluateHookeJeeves()

println("\nNelder Mead")
@time evaluateNelderMead()

println("\nOriginal Newtons Method")
@time evaluateOriginalNewtonsMethod()

println("\nModified Newtons Method with Levenberg Marquardt")
@time evaluateModifiedNewtonsWithLM()

println("\nEvaluating Times (BenchmarkTools)")
evaluateTimes()
```

A.2 makeplots.jl

```
include("A1Module.jl")
include("A2Module.jl")
include("objectivefunction.jl")

# Self-written modules import
using .A1Module: HookeJeeves, Q2SteepestDescent
using .A2Module
using .objectivefunctionModule: NDRosenbrock, autodiffGradientNDRosenbrock,
    autodiffHessianNDRosenbrock

# Useful external modules
using BenchmarkTools
using LinearAlgebra: cond
using Memento
using OrderedCollections
using Plots
using ValueHistories: MVHistory, History
import YAML

# Suppress Memento from inner modules
setlevel!(getlogger(A1Module), "not_set")
setlevel!(getlogger(A2Module.A1Module), "not_set")

# Define general settings (N = 5 dimensional rosenbrock)
const test_initial_point = zeros(5);

#Generated Initial Vectors 2 to 5 are generated using generate_random_inits.jl
const array_of_inits = [[ 0.00,  0.00,  0.00,  0.00,  0.00],
                        [ 0.36,  1.18, -1.01, -1.73, -0.90],
                        [ 1.07,  1.42,  0.32,  1.83,  0.61],
                        [ 0.26, -1.20,  0.60,  0.59, -1.77],
                        [-0.16, -0.81, -1.96, -1.55,  1.37]];

# Define objective functions to use
```

```
global N_f_evals = 0
global N_grad_evals = 0
global N_hessian_evals = 0

function Rosenbrock5D(x::Array{T}) where T <: Real
    global N_f_evals += 1
    return NDRosenbrock(5, x)
end

function GradRosenbrock5D(x::Array{T}) where T <: Real
    global N_grad_evals += 1
    return autodiffGradientNDRosenbrock(5, x)
end

function HessianRosenbrock5D(x::Array{T}) where T <: Real
    global N_hessian_evals += 1
    return autodiffHessianNDRosenbrock(5, x)
end

function generatePlot_LossVsIterations(array_of_histories::Array{MVHistory{History}},
    array_of_labels::Array{String},
    symbol_to_get::Symbol)

    @assert length(array_of_histories) == length(array_of_labels)

    resultant_plot = plot()

    for (label, historyofhistories) in zip(array_of_labels, array_of_histories)
        is, xs = get(historyofhistories, symbol_to_get)

        errors = []
        for (i, x) in zip(is, xs)
            error = Rosenbrock5D(x)
            push!(errors, error)
        end

        plot!(resultant_plot, is, errors, label=label,
            yscale=:log10, lw=3, shape = :circle, markersize=3)
    end

    return resultant_plot
end

function makeDataDict(initial_vector, final_vector, final_loss;
        N_f_evals = 0, N_grad_evals = 0, N_hessian_evals = 0, N_linsys_solves = 0)

    return OrderedDict(
        "initial_vector" => initial_vector,
        "final_vector" => final_vector,
        "final_loss" => final_loss,
        "N_f_evals" => N_f_evals,
        "N_grad_evals" => N_grad_evals,
        "N_hessian_evals" => N_hessian_evals,
        "N_linsys_solves" => N_linsys_solves
    )

end

function onerunGradientDescent(x_0::Array{Float64})
    tol = 1e-4;
    global N_f_evals = 0;
    global N_grad_evals = 0;
    best_result, history = A1Module.Q2SteepestDescent(Rosenbrock5D,
        GradRosenbrock5D, x_0, tol;
        linesearch_method="SwannsBracketingMethod")
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss;
        N_f_evals = N_f_evals,
        N_grad_evals = N_grad_evals)

    return data_dict, best_result, history
end

function onerunPowellConjugateGradient(x_0::Array{T}) where T <: Real
    tol = 1e-6
    linesearch_tol = 1e-3
    max_iter = 10000;
    global N_f_evals = 0;
    best_result, history = powellsConjugateGradientMethod(Rosenbrock5D, x_0,
        tol; max_iter = max_iter, linesearch_tol=linesearch_tol)
```

```
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss;
        N_f_evals = N_f_evals)

    return data_dict, best_result, history
end

function onerunConjugateGradient(x_0::Array{T}, method::String) where T <: Real
    tol_for_linesearch = 1e-3;
    g_tol = 1e-4
    k_max = 10000;
    n_resetsearchdir = 10;

    global N_f_evals = 0;
    global N_grad_evals = 0;
    best_result, history = conjugateGradient(Rosenbrock5D, GradRosenbrock5D, x_0,
        g_tol, k_max, n_resetsearchdir;
        method=method, tol_for_linesearch=tol_for_linesearch)
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss;
        N_f_evals = N_f_evals,
        N_grad_evals = N_grad_evals)

    return data_dict, best_result, history
end

function onerunHookeJeeves(x_0::Array{Float64})
    initial_delta = 1.;
    final_delta = 1.e-3;
    orthogonal_directions = [[1., 0., 0., 0., 0.],
                             [0., 1., 0., 0., 0.],
                             [0., 0., 1., 0., 0.],
                             [0., 0., 0., 1., 0.],
                             [0., 0., 0., 0., 1.]];
    global N_f_evals = 0;
    best_result, history = HookeJeeves(Rosenbrock5D, x_0, initial_delta, final_delta,
        orthogonal_directions)
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss; N_f_evals = N_f_evals)

    return data_dict, best_result, history
end

function onerunNelderMead(x_0::Array{T}) where T <: AbstractFloat
    initial_sidelength = 1.0;
    max_iter = 500;
    stuck_max = 10;
    stuck_coef = 0.5;
    global N_f_evals = 0;
    best_result, history = nelderMeadSimplexSearch(Rosenbrock5D, x_0, initial_sidelength;
        max_iter = max_iter, stuck_max = stuck_max, stuck_coef = stuck_coef)
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss; N_f_evals = N_f_evals)
    return data_dict, best_result, history
end

function onerunOriginalNewtonsMethod(x_0::Array{T}) where T <: Real
    g_tol = 1e-3
    max_iter = 1000
    global N_f_evals = 0
    global N_grad_evals = 0
    global N_hessian_evals = 0
    best_result, history, num_linsys_solves = originalNewtonsMethod(GradRosenbrock5D,
        HessianRosenbrock5D, x_0; g_tol = g_tol, max_iter = max_iter)
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss;
        N_f_evals = N_f_evals,
        N_grad_evals = N_grad_evals,
        N_hessian_evals = N_hessian_evals,
        N_linsys_solves = num_linsys_solves)

    return data_dict, best_result, history
end

function onerunModifiedNewtonsMethodWithLM(x_0::Array{T}, mu_param::T) where T <: Real
    linesearch_tol = 1e-3;
```

```
    g_tol = 1e-3;
    max_iter = 1000;

    global N_f_evals = 0
    global N_grad_evals = 0
    global N_hessian_evals = 0
    best_result, history, num_linsys_solves = modifiedNewtonsWithLMMethod(GradRosenbrock5D,
        HessianRosenbrock5D, x_0, linesearch_tol = linesearch_tol,
        mu_param = mu_param, g_tol = g_tol, max_iter = max_iter)
    final_loss = Rosenbrock5D(best_result)

    data_dict = makeDataDict(x_0, best_result, final_loss;
        N_f_evals = N_f_evals,
        N_grad_evals = N_grad_evals,
        N_hessian_evals = N_hessian_evals,
        N_linsys_solves = num_linsys_solves)

    return data_dict, best_result, history
end

function evaluateGradientDescent()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunGradientDescent(x_0)
        # @show typeof(history)
        # @show typeof(array_of_histories)
        # @show data_dict
        # @show best_result

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end
    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/GradientDescent_TrialOutputs.yml", all_trial_dicts)


    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :Nd_point)
    xlabel!(plot_losses, "Number of Gradient Descent Iterations")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations - Gradient Descent")
    savefig(plot_losses, "assets/GradientDescentLossPlot.png")
end

function evaluatePowellConjugateGradient()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunPowellConjugateGradient(x_0)

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end

    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/PowellConjugateGradient_TrialOutputs.yml", all_trial_dicts)

    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
    xlabel!(plot_losses, "Number of Iterations")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nPowell Conjugate Gradient")
    savefig(plot_losses, "assets/PowellConjugateGradient_LossPlot.png")
end

function evaluateConjugateGradientFletcherReeves()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunConjugateGradient(x_0, "FletcherReeves")
        # @show typeof(history)
        # @show typeof(array_of_histories)
        # @show data_dict
        # @show best_result

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
```

```
        array_of_histories[i] = history
    end

    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/ConjugateGradientFletcherReeves_TrialOutputs.yml", all_trial_dicts)

    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
    xlabel!(plot_losses, "Number of Iterations/Updates")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nConjugate Gradient (Fletcher-Reeves)")
    savefig(plot_losses, "assets/ConjugateGradientFletcherReeves_LossPlot.png")
end

function evaluateConjugateGradientHestenesStiefel()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunConjugateGradient(x_0, "HestenesStiefel")
        # @show typeof(history)
        # @show typeof(array_of_histories)
        # @show data_dict
        # @show best_result

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end

    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/ConjugateGradientHestenesStiefel_TrialOutputs.yml", all_trial_dicts)

    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
    xlabel!(plot_losses, "Number of Iterations/Updates")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nConjugate Gradient (Hestenes-Stiefel)")
    savefig(plot_losses, "assets/ConjugateGradientHestenesStiefel_LossPlot.png")
end

function evaluateConjugateGradientPolakRibiere()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunConjugateGradient(x_0, "PolakRibiere")
        # @show typeof(history)
        # @show typeof(array_of_histories)
        # @show data_dict
        # @show best_result

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end

    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/ConjugateGradientPolakRibiere_TrialOutputs.yml", all_trial_dicts)

    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
    xlabel!(plot_losses, "Number of Iterations/Updates")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nConjugate Gradient (Polak-Ribière)")
    savefig(plot_losses, "assets/ConjugateGradientPolakRibiere_LossPlot.png")
end

function evaluateHookeJeeves()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunHookeJeeves(x_0)
        # @show typeof(history)
        # @show typeof(array_of_histories)
        # @show data_dict
        # @show best_result

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end
```

```
    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/HookeJeeves_TrialOutputs.yml", all_trial_dicts)


    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_1)
    xlabel!(plot_losses, "Number of Hooke-Jeeves Outer-loop Iterations")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations - Hooke-Jeeves")
    savefig(plot_losses, "assets/HookeJeevesLossPlot.png")
end

function evaluateNelderMead()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunNelderMead(x_0)

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end
    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/NelderMead_TrialOutputs.yml", all_trial_dicts)


    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_best)
    plot!(plot_losses, legend=:bottomleft)
    xlabel!(plot_losses, "Number of Iterations")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nNelder-Mead Simplex Search")
    savefig(plot_losses, "assets/NelderMead_LossPlot.png")
end

function evaluateOriginalNewtonsMethod()
    array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
    array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
    array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));

    for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
        data_dict, best_result, history = onerunOriginalNewtonsMethod(x_0)

        array_of_trials_dicts[i] = OrderedDict(label => data_dict)
        array_of_histories[i] = history
    end

    all_trial_dicts = merge(array_of_trials_dicts...)
    YAML.write_file("assets/OriginalNewtonsMethod_TrialOutputs.yml", all_trial_dicts)

    plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
    xlabel!(plot_losses, "Number of Iterations")
    ylabel!(plot_losses, "Loss")
    title!(plot_losses, "Loss vs Iterations\nOriginal Newtons Method")
    savefig(plot_losses, "assets/OriginalNewtonsMethod_LossPlot.png")

    begin
        plot_condnum_matrices = plot()
        for (label, historyofhistories) in zip(array_of_labels, array_of_histories)
            is, matrices = get(historyofhistories, :hessian_current)
            condnums = []
            for (i, M) in zip(is, matrices)
                condnum = cond(M)
                push!(condnums, condnum)
            end
            plot!(plot_condnum_matrices, is, condnums, label=label,
                yscale=:log10, lw=3, shape = :circle, markersize=3, legend=:bottomright)
        end
        xlabel!(plot_condnum_matrices, "Number of Iterations")
        ylabel!(plot_condnum_matrices, "Condition Number of Hessian")
        title!(plot_condnum_matrices, "Hessian Condition Number (log scale) vs Iterations\nOriginal Newtons Method")
        savefig(plot_condnum_matrices, "assets/OriginalNewtonsMethod_ConditionNumberHessianPlot.png")
    end
end

function evaluateModifiedNewtonsWithLM()
    array_mu_params = [0.0, 1.0, 10.0]
    for (mu_index, mu_param) in enumerate(array_mu_params)
        array_of_labels = ["Initial Vector $i" for i in 1:length(array_of_inits)];
        array_of_trials_dicts = Array{OrderedDict}(undef, length(array_of_inits));
        array_of_histories = Array{MVHistory{History}}(undef, length(array_of_inits));
```

```
        for (i, (label, x_0)) in enumerate(zip(array_of_labels, array_of_inits))
            data_dict, best_result, history = onerunModifiedNewtonsMethodWithLM(x_0, mu_param)

            merge!(data_dict, OrderedDict("mu_param"=>mu_param))
            array_of_trials_dicts[i] = OrderedDict(label => data_dict)
            array_of_histories[i] = history
        end

        all_trial_dicts = merge(array_of_trials_dicts...)
        YAML.write_file("assets/ModifiedNewtons/ModifiedNewtonsWithLM_TrialOutputs_$mu_index.yml", all_trial_dicts)

        plot_losses = generatePlot_LossVsIterations(array_of_histories, array_of_labels, :x_current)
        plot!(legend=:bottomleft)
        xlabel!(plot_losses, "Number of Iterations")
        ylabel!(plot_losses, "Loss")
        title!(plot_losses, "Loss vs Iterations\nModified Newtons Method with Levenberg-Marquardt\n(mu=$mu_param)")
        savefig(plot_losses, "assets/ModifiedNewtons/ModifiedNewtonsWithLM_LossPlot_$mu_index.png")

        begin
            plot_condnum_matrices = plot()
            for (label, historyofhistories) in zip(array_of_labels, array_of_histories)
                is, matrices = get(historyofhistories, :LM_matrix)
                condnums = []
                for (i, M) in zip(is, matrices)
                    condnum = cond(M)
                    push!(condnums, condnum)
                end
                plot!(plot_condnum_matrices, is, condnums, label=label,
                    yscale=:log10, lw=3, shape = :circle, markersize=3, legend=:bottomright)
            end
            xlabel!(plot_condnum_matrices, "Number of Iterations")
            ylabel!(plot_condnum_matrices, "Condition Number of Matrix")
            title!(plot_condnum_matrices, "Matrix Condition Number (log scale) vs Iterations\nModified Newtons Method with Levenberg-M
            savefig(plot_condnum_matrices, "assets/ModifiedNewtons/ModifiedNewtonsWithLM_ConditionNumberHessianPlot_$mu_index.png")
        end

        begin
            begin
                plot_condnum_matrices_with_mu = plot()
                for (label, historyofhistories) in zip(array_of_labels, array_of_histories)
                    is, matrices = get(historyofhistories, :LM_matrix)
                    condnums_with_diag = []
                    for (i, M) in zip(is, matrices)
                        condnum = cond(M)
                        push!(condnums_with_diag, condnum)
                    end
                    plot!(plot_condnum_matrices_with_mu, is, condnums_with_diag, label=label,
                        yscale=:log10, lw=3, shape = :circle, markersize=3, legend=:topright)
                end
                xlabel!(plot_condnum_matrices_with_mu, "Number of Iterations")
                ylabel!(plot_condnum_matrices_with_mu, "Condition Number\nLM Matrix")
                title!(plot_condnum_matrices_with_mu, "LM Matrix Condition Number vs Iterations")
            end

            begin
                plot_condnum_matrices_hessian = plot()
                for (label, historyofhistories) in zip(array_of_labels, array_of_histories)
                    is, matrices = get(historyofhistories, :hessian_current)
                    condnums_hessian = []
                    for (i, M) in zip(is, matrices)
                        condnum = cond(M)
                        push!(condnums_hessian, condnum)
                    end
                    plot!(plot_condnum_matrices_hessian, is, condnums_hessian, label=label,
                        yscale=:log10, lw=3, shape = :circle, markersize=3, legend=:bottomright)
                end
                xlabel!(plot_condnum_matrices_hessian, "Number of Iterations")
                ylabel!(plot_condnum_matrices_hessian, "Condition Number\nPure Hessian Matrix")
                title!(plot_condnum_matrices_hessian, "Hessian Condition Number (no mu diagonal) vs Iterations")
            end

            layout_combined = @layout [a; b]
            plot_combined = plot(plot_condnum_matrices_hessian, plot_condnum_matrices_with_mu, layout = layout_combined)
            savefig(plot_combined, "assets/ModifiedNewtons/ModifiedNewtonsWithLM_MatrixCompare_$mu_index.png")
        end

    end
end

function evaluateTimes()
```

```
    setlevel!(getlogger(A2Module), "not_set")

    x_0 = array_of_inits[1]
    println("\n\nSteepest Descent")
    b = @benchmark onerunGradientDescent($x_0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nPowell Conjugate Direction")
    b = @benchmark onerunPowellConjugateGradient($x_0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nCG- Fletcher Reeves")
    b = @benchmark onerunConjugateGradient($x_0, "FletcherReeves")
    show(stdout, MIME("text/plain"), b)

    println("\n\nCG- HestenesStiefel")
    b = @benchmark onerunConjugateGradient($x_0, "HestenesStiefel")
    show(stdout, MIME("text/plain"), b)

    println("\n\nCG- PolakRibiere")
    b = @benchmark onerunConjugateGradient($x_0, "PolakRibiere")
    show(stdout, MIME("text/plain"), b)

    println("\n\nHooke Jeeves")
    b = @benchmark onerunHookeJeeves($x_0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nNelder Mead")
    b = @benchmark onerunNelderMead($x_0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nOriginal Newtons Method")
    b = @benchmark onerunOriginalNewtonsMethod($x_0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nModified Newton (LM) mu = 0.0")
    b = @benchmark onerunModifiedNewtonsMethodWithLM($x_0, 0.0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nModified Newton (LM) mu = 1.0")
    b = @benchmark onerunModifiedNewtonsMethodWithLM($x_0, 1.0)
    show(stdout, MIME("text/plain"), b)

    println("\n\nModified Newton (LM) mu = 10.0")
    b = @benchmark onerunModifiedNewtonsMethodWithLM($x_0, 10.0)
    show(stdout, MIME("text/plain"), b)
end
```

A.3 objectivefunction.jl

```
module objectivefunctionModule

using ForwardDiff

export NDRosenbrock

export analyticGradientNDRosenbrock
export analyticHessianNDRosenbrock

export autodiffGradientNDRosenbrock
export autodiffHessianNDRosenbrock

function NDRosenbrock(N::Integer, x::Array{T}) where T <: Real
    @assert length(x) == N
    @assert N >= 2
    result = 0
    for i in 1:(N-1)
        result += 100*(x[i+1] - x[i]^2)^2 + (1 - x[i])^2
    end
    return result
end

typeATerm(x_i) = 2*(1-x_i)*(-1)
typeBTerm(x_i, x_next) = 100*2*(x_next - x_i^2)*(-2*x_i)
typeCTerm(x_prev, x_i) = 100*2*(x_i - x_prev^2)
```

```julia
function analyticGradientNDRosenbrock(N::Integer, x::Array{T}) where T <: Real
    gradresult = zeros(eltype(x), N)

    for i in 1:N
        if i == 1 #First
            gradresult[i] =
                typeATerm(x[i]) + typeBTerm(x[i], x[i+1])
        elseif i == N #Last
            gradresult[i] =
                typeCTerm(x[i-1], x[i])
        else #Intermediary terms
            gradresult[i] =
                typeATerm(x[i]) + typeBTerm(x[i], x[i+1]) + typeCTerm(x[i-1], x[i])
        end
    end

    return gradresult
end

function analyticHessianNDRosenbrock(N::Integer, x::Array{T}) where T <: Real
    error("Currently undefined. Please use autodiffHessianNDRosenbrock.")
end

function autodiffGradientNDRosenbrock(N::Integer, x::Array{T}) where T <: Real
    return ForwardDiff.gradient(in -> NDRosenbrock(N, in), x)
end

function autodiffHessianNDRosenbrock(N::Integer, x::Array{T}) where T <: Real
    return ForwardDiff.hessian(in -> NDRosenbrock(N, in), x)
end

end
```

A.4 generate_random_inits.jl

```julia
using Random

const MIN_MAX_VAL = 2;
function generate_random_inits()
    rng = Random.MersenneTwister(1234);

    random_array = (rand(rng, 4, 5) .* 2 .- 1) * MIN_MAX_VAL;
    round.(random_array; digits = 2)
end

generate_random_inits()
```

A.5 A1Module.jl

```julia
module A1Module

using LinearAlgebra #External module for taking norm
using Memento #Invenia Module for logging
using Printf #External module for formatting strings
using Plots
using ValueHistories #External Package for keeping track of values

export SwannsBracketingMethod
export PowellsBracketingMethod
export GoldenSectionSearch
export Q1LineSearch
export Q2SteepestDescent
export HookeJeeves

# Set up Memento Logger
const LOGGER = getlogger(@__MODULE__)
function __init__()
    Memento.register(LOGGER)
end
```

```
function SwannsBracketingMethod(f, x_0, initial_step_length; magnification = 2)
    info(LOGGER, "Inside Swann's Bracketing Method")

    #Define test points
    x_l = x_0 - abs(initial_step_length) #lower
    x_u = x_0 + abs(initial_step_length) #upper
    x_m = x_0 #middle


    #Do function evaluations
    debug(LOGGER, "Performing initial function evaliations")
    f_l = f(x_l) #lower
    f_u = f(x_u) #upper
    f_m = f(x_0) #middle

    history = History(Tuple{Float64, Float64, Float64})
    push!(history, 0, convert(Tuple{Float64, Float64, Float64}, (x_l, x_u, x_m)))

    # 4 Cases.
    # 1) Keep moving right
    # 2) Keep moving left
    # 3) Initial interval is bracket
    # 4) Error (non-unimodal)

    debug(LOGGER, @sprintf "x_l x_m x_u = [%3.2f, %3.2f %3.2f]" x_l x_m x_u)
    if f_l >= f_m >= f_u
        debug(LOGGER, "Case 1: Keep Moving Right")
        i = 1
        while f_u < f_m
            debug(LOGGER, @sprintf "x_l to x_m = [%3.2f, %3.2f]" x_l x_m)

            (x_l, x_m, f_l, f_m) = (x_m, x_u, f_m, f_u)
            x_u = x_u + (magnification^i) * abs(initial_step_length)
            f_u = f(x_u)

            push!(history, i, convert(Tuple{Float64, Float64, Float64}, (x_l, x_u, x_m)))
            i += 1
        end
        # f_u is now higher than f_m
        # This now forms a valid bracketing interval

        info(LOGGER, @sprintf "Outputting x_l x_u = [%3.2f %3.2f]" x_l x_u)
        return (x_l, x_u), history

    elseif f_l <= f_m <= f_u
        debug(LOGGER, "Case 2: Keep Moving Left")
        i = 1
        while f_l < f_m
            debug(LOGGER, @sprintf "x_u to x_m = [%3.2f, %3.2f]" x_l x_m)

            (x_u, x_m, f_u, f_m) = (x_m, x_l, f_m, f_l)
            x_l = x_l - (magnification^i) * abs(initial_step_length)
            f_l = f(x_l)

            push!(history, i, convert(Tuple{Float64, Float64, Float64}, (x_l, x_u, x_m)))
            i += 1
        end
        # f_l is now higher than f_m
        # This forms a valid bracketing interval
        info(LOGGER, @sprintf "Outputting x_l x_u = [%3.2f %3.2f]" x_l x_u)
        return (x_l, x_u), history

    elseif f_l >= f_m <= f_u
        debug(LOGGER, "Case 3: Initial interval is bracket")
        info(LOGGER, @sprintf "Outputting x_l x_u = [%3.2f %3.2f]" x_l x_u)
        return (x_l, x_u), history

    else # 4) Error  f_l <= f_m >= f_u
        error(LOGGER, "Case 4: Error (non-unimodal)")
    end

end

function PowellsBracketingMethod(f, a_1, delta, delta_max)
    #TODO: Gets an error when the function is perfectly quadratic e.g x -> x^2
    info(LOGGER, "Inside Powell's Bracketing Method")

    c_1 = a_1 + delta
    F_a = f(a_1)
    F_c = f(c_1)
```

```
    if F_a > F_c
        b_1 = a_1 + 2*delta
        F_b = f(b_1)
        forward = true
    else
        (b_1, c_1, a_1) = (c_1, a_1, a_1 - delta)
        (F_b, F_c, F_a) = (F_c, F_a, f(a_1))
        forward = false
    end

    debug(LOGGER, @sprintf "Direction is forward: %s" forward)
    debug(LOGGER, @sprintf " a_1, b_1, c_1 = %s" (a_1, b_1, c_1))
    debug(LOGGER, @sprintf " F_a, F_b, F_c = %s" (F_a, F_b, F_c))

    a_current = a_1
    b_current = b_1
    c_current = c_1

    a_next = a_1
    b_next = b_1
    c_next = c_1

    history = History(Tuple{Float64, Float64, Float64})
    push!(history, 0, convert(Tuple{Float64, Float64, Float64}, (a_current, b_current, c_current)))

    N_iterations = 0
    while !(F_c < F_a && F_c < F_b)
        debug(LOGGER, @sprintf "New loop iteration...")
        debug(LOGGER, @sprintf "F_a F_b F_c = %5.3f %5.3f %5.3f" F_a F_b F_c)
        a_current = a_next
        b_current = b_next
        c_current = c_next
        N_iterations += 1

        debug(LOGGER, @sprintf "a_current, b_current, c_current = [%5.3f, %5.3f, %5.3f]" a_current b_current c_current)


        p_numerator = (c_current - b_current)*F_a + (a_current - c_current)*F_b + (b_current-a_current)*F_c
        p_denominator = ((b_current - c_current)*(c_current - a_current)*(a_current - b_current))
        p = p_numerator / p_denominator
        debug(LOGGER, @sprintf "p = %5.3f" p)

        # Cases Summarized
        # 1) Moving Forward
        # 1.1) Quadratic has maximum (p <= 0)
        # 1.2) Quadratic has no maximum
        # 1.2.1) Quadratic minimum is too far away
        # 1.2.2) Quadratic minimum is within reach
        # 2) Moving Backward
        # 2.1) Quadratic has maximum (p <= 0)
        # 2.2) Quadratic has no maximum
        # 2.2.1) Quadratic minimum is too far away
        # 2.2.2) Quadratic minimum is within reach

        if forward #Moving Forward
            debug(LOGGER, @sprintf "Moving Forward...")
            if p <= 0 #Quadratic has a maximum, move as far as possible
                debug(LOGGER, @sprintf "Quadratic has a maximum. Move as far forward as possible.")
                a_next = a_current
                b_next = b_current + delta_max
                c_next = c_current

                F_b = f(b_next)
            else
                #p > 0 concave up
                x_star_num = (b_current^2 - c_current^2)*F_a + (c_current^2 - a_current^2)*F_b + (a_current^2 - b_current^2)*F_c
                x_star_denom = (b_current - c_current)*F_a + (c_current - a_current)*F_b + (a_current - b_current)*F_c
                x_star = (1/2) * x_star_num / x_star_denom

                debug(LOGGER, @sprintf "p > 0, with x_star = %5.3f" x_star)

                if (x_star - b_current) > delta_max #Quadratic minimum is too far
                    debug(LOGGER, @sprintf "Quadratic minimum is too far. Moving as far forward as possible." )
                    b_next = b_current + delta_max
                else
                    debug(LOGGER, @sprintf "Quadratic minimum is reachable. (forward)" )
                    b_next = x_star
                end

                a_next = c_current
```

```
                c_next = b_current

                (F_a, F_c, F_b) = (F_c, F_b, f(b_next))
            end
        else #Moving Backward
            debug(LOGGER, @sprintf "Moving Backward...")
            if p <= 0 #Quadratic has a maximum, move as far as possible
                debug(LOGGER, @sprintf "Quadratic has a maximum. Move as far back as possible.")
                a_next = b_current - delta_max
                b_next = b_current
                c_next = c_current

                F_a = f(a_next)
            else
                #p > 0 concave up
                x_star_num = (b_current^2 - c_current^2)*F_a + (c_current^2 - a_current^2)*F_b + (a_current^2 - b_current^2)*F_c
                x_star_denom = (b_current - c_current)*F_a + (c_current - a_current)*F_b + (a_current - b_current)*F_c
                x_star = (1/2) * x_star_num / x_star_denom

                debug(LOGGER, @sprintf "p > 0, with x_star = %5.3f" x_star)

                if (a_current - x_star) > delta_max #Quadratic minimum is too far
                    debug(LOGGER, @sprintf "Quadratic minimum is too far. Moving as far back as possible." )
                    a_next = a_current - delta_max
                else
                    debug(LOGGER, @sprintf "Quadratic minimum is reachable. (backwards)" )
                    a_next = x_star
                end

                b_next = c_current
                c_next = a_current
                (F_b, F_c, F_a) = (F_c, F_a, f(a_next))
            end
        end

        push!(history, N_iterations, convert(Tuple{Float64, Float64, Float64}, (a_next, b_next, c_next)))

        debug(LOGGER, @sprintf "a_next b_next c_next = %5.3f %5.3f %5.3f" a_next b_next c_next)
        debug(LOGGER, @sprintf "End of loop iteration...")
    end

    a_current = a_next
    b_current = b_next
    c_current = c_next

    debug(LOGGER, @sprintf "Exiting with a_current, b_current c_current = %5.3f %5.3f %5.3f" a_current b_current c_current)
    debug(LOGGER, @sprintf "Exiting with F_a F_b F_c = %5.3f %5.3f %5.3f" F_a F_b F_c)

    return (a_current, b_current), history
end

function GoldenSectionSearch(f, a, b, tolerance)
    info(LOGGER, "Inside Golden Section Search")

    TAU = 0.618_033_988_7

    F_a = f(a)
    F_b = f(b)

    c = a + (1 - TAU)*(b - a)
    F_c = f(c)

    d = b - (1 - TAU)*(b - a)
    F_d = f(d)

    a_current = a
    b_current = b
    c_current = c
    d_current = d

    a_next = a
    b_next = b
    c_next = c
    d_next = d

    history = History(Tuple{Float64, Float64, Float64, Float64})
    push!(history, 0, convert(Tuple{Float64, Float64, Float64, Float64}, (a_current, b_current, c_current, d_current)))

    N_iterations = 0
    interval_size = abs(b - a)
    while !(interval_size < tolerance)
```

```
        a_current = a_next
        b_current = b_next
        c_current = c_next
        d_current = d_next
        N_iterations += 1

        debug(LOGGER, @sprintf "Start of loop iteration... (a_current, b_current, c_current, d_current) = (%5.3f %5.3f %5.3f %5.3f)"
        debug(LOGGER, @sprintf "Start of loop iteration... (F_a, F_b, F_c, F_d) = (%5.3f %5.3f %5.3f %5.3f)" F_a F_b F_c F_d)

        if F_c < F_d
            debug(LOGGER, "F_c < F_d case")
            a_next = a_current
            b_next = d_current
            c_next = a_next + (1 - TAU)*(b_next - a_next)
            d_next = c_current

            (F_a, F_b) = (F_a, F_d)
            (F_c, F_d) = (f(c_next), F_c)

        else
            debug(LOGGER, "F_c > F_d case")
            a_next = c_current
            b_next = b_current
            c_next = d_current
            d_next = b_next - (1 - TAU)*(b_next - a_next)

            (F_a, F_b) = (F_c, F_b)
            (F_c, F_d) = (F_d, f(d_next))

        end

        push!(history, N_iterations, convert(Tuple{Float64, Float64, Float64, Float64}, (a_next, b_next, c_next, d_next)))
        interval_size = abs(b_next - a_next)
        debug(LOGGER, @sprintf "End of loop iteration... interval_size = %5.3f" interval_size)
    end

    a_current = a_next
    b_current = b_next
    c_current = c_next
    d_current = d_next

    info(LOGGER, @sprintf "Exiting Golden Section Search with a_current, b_current = %5.3f %5.3f" a_current b_current)
    info(LOGGER, @sprintf "Exiting Golden Section Search with F_a, F_b = %5.3f %5.3f" F_a F_b)
    return (a_current, b_current), history

end

function Q1LineSearch(f, d, x_0, desired_interval_size; linesearch_method = "")
    info(LOGGER, "Entering Q1LineSearch...")
    info(LOGGER, @sprintf "Entering with d = %s" d)
    info(LOGGER, @sprintf "Entering with x_0 = %s" x_0)

    one_dimensional_function = alpha -> f((x_0 .+ alpha .* d))

    bracketing_history = undef
    golden_history = undef
    a_l_smaller, a_u_smaller = undef, undef
    if linesearch_method == "SwannsBracketingMethod"
        swanns_step_length = 1 #HARDCODED
        alpha_init = 0 #HARDCODED
        (alpha_lower, alpha_upper), swanns_history = SwannsBracketingMethod(one_dimensional_function, alpha_init, swanns_step_length)
        (a_l_smaller, a_u_smaller), golden_history = GoldenSectionSearch(one_dimensional_function, alpha_lower, alpha_upper, desired_
        bracketing_history = swanns_history
    elseif linesearch_method == "PowellsBracketingMethod"
        alpha_init = 0 #HARDCODED
        powells_delta = 1 #HARDCODED
        powells_delta_max = 16 #HARDCODED
        (alpha_lower, alpha_upper), powell_history = PowellsBracketingMethod(one_dimensional_function, alpha_init, powells_delta, pow
        (a_l_smaller, a_u_smaller), golden_history = GoldenSectionSearch(one_dimensional_function, alpha_lower, alpha_upper, desired_
        bracketing_history = powell_history
    else
        error(LOGGER, "Line Search Method not recognized: $linesearch_method")
    end

    a_mid = (a_l_smaller + a_u_smaller) / 2 #along line
    debug(LOGGER, @sprintf "a_middle %5.3f" a_mid)

    full_middle_point = @. x_0 + a_mid * d #In full N-D space
    info(LOGGER, @sprintf "Exiting Q1LineSearch with middle point in N-D as %s" full_middle_point)
    return full_middle_point, bracketing_history, golden_history
end
```

```
function Q2SteepestDescent(f, grad_f, x_0, tolerance_for_1D_search; linesearch_method = "")
    info(LOGGER, "Entering Q2SteepestDescent...")
    info(LOGGER, @sprintf "Entering with x_0 = %s" x_0)

    x_0 = convert(Array{Float64, 1}, x_0)

    current_point = x_0
    next_point = x_0
    steepest_descent_direction = -1 * grad_f(x_0)

    Q2_history = MVHistory()
    push!(Q2_history, :Nd_point, 0, current_point)

    N_iterations = 0
    while !(norm(steepest_descent_direction) < 10^(-4))
        info(LOGGER, @sprintf "Q2 Start of Loop Iteration... current_point = %s" current_point)
        info(LOGGER, @sprintf "Q2 Start of Loop Iteration... steepest = %s" steepest_descent_direction)
        N_iterations += 1

        next_point, bracketing_history, golden_history = Q1LineSearch(f, steepest_descent_direction, current_point, tolerance_for_1D_
linesearch_method)

        steepest_descent_direction = -1 * grad_f(next_point)
        current_point = next_point

        push!(Q2_history, :Nd_point, N_iterations, current_point)
        push!(Q2_history, :bracketing_history, N_iterations, bracketing_history)
        push!(Q2_history, :golden_history, N_iterations, golden_history)
        debug(LOGGER, @sprintf "End of Loop Iteration... norm of grad %s" norm(steepest_descent_direction))
    end

    info(LOGGER, @sprintf "Exiting with next_point = %s" next_point)
    return next_point, Q2_history

end

function HookeJeeves(f, x_0, big_delta, small_delta, orthogonal_directions)
    info(LOGGER, "Entering Hooke-Jeeves...")
    @assert length(x_0) == length(orthogonal_directions)

    N = length(orthogonal_directions)
    x_i_array = zeros(length(x_0), length(orthogonal_directions)+1)

    x_0_current = x_0
    x_i_array[:, 1] = x_0
    current_big_delta = big_delta

    history = MVHistory()
    push!(history, :x_1, 0, x_i_array[:, 1])
    push!(history, :x_0_current, 0, x_0_current)
    push!(history, :current_big_delta, 0, current_big_delta)

    iteration_number = 0
    while !(current_big_delta < small_delta)
        iteration_number += 1
        debug(LOGGER, "Start of Iteration Loop... ($iteration_number)")


        #Exploratory Moves
        debug(LOGGER, "Performing Exploratory Moves... (delta = $current_big_delta)")
        prev_x = []
        for (index, direction) in enumerate(orthogonal_directions)
            debug(LOGGER, "Trying out direction ($index): $direction")
            debug(LOGGER, @sprintf "Trying from %s" x_i_array[:, index])
            x_current = x_i_array[:, index] .+ current_big_delta * direction

            F_x_i =  f(x_i_array[:, index])
            F_x_current_forward = f(x_current)
            if F_x_current_forward < F_x_i
                x_i_array[:, index+1] = x_current
                debug(LOGGER, "Forward is good to $x_current")
            else
                x_current = x_i_array[:, index] .- current_big_delta * direction
                F_x_current_backward = f(x_current)

                if F_x_current_backward < F_x_i
                    x_i_array[:, index+1] = x_current
                    debug(LOGGER, "Backward is good to $x_current")

                else
```

```
                    x_i_array[:, index+1] = x_i_array[:, index]
                    debug(LOGGER, @sprintf "Stay here at %s" x_i_array[:, index])
                    @assert (F_x_current_forward >= F_x_i)
                    @assert (F_x_current_backward >= F_x_i)


                end
            end
        end

        # Pattern Move
        debug(LOGGER, @sprintf "Performing Pattern Move Branching... with x_i_array[:, N+1] = %s" x_i_array[:, N+1])
        if f(x_i_array[:, N+1]) < f(x_0_current)
            debug(LOGGER, "Performing Pattern Move...")
            x_i_array[:, 1] = x_i_array[:, N+1] .+ (x_i_array[:, N+1] .- x_0_current)
            x_0_current = x_i_array[:, N+1]
            debug(LOGGER, @sprintf "x_i_array[:, 1] updated to %s" x_i_array[:, 1])
            debug(LOGGER, @sprintf "x_0_current updated to %s" x_0_current)

        elseif x_i_array[:, 1] == x_0_current
            debug(LOGGER, "Pattern move not better x_i_array[:, 1] == x_0_current")
            current_big_delta = current_big_delta / 10
            debug(LOGGER, "Reduced big_delta to $current_big_delta")

        else
            debug(LOGGER, "Pattern Move not better: Set x_i_array[:, 1] to x_0_current")
            x_i_array[:, 1] = x_0_current
            debug(LOGGER, @sprintf "x_i_array[:, 1] updated to %s" x_i_array[:, 1])

        end

        push!(history, :x_1, iteration_number, x_i_array[:, 1])
        push!(history, :x_0_current, iteration_number, x_0_current)
        push!(history, :x_end, iteration_number, x_i_array[:, N+1])
        push!(history, :current_big_delta, iteration_number, current_big_delta)
        debug(LOGGER, "End of Iteration Loop... ($iteration_number)")
    end

    info(LOGGER, "Exiting Hooke-Jeeves...")
    return x_i_array[:, 1], history
end

end
```

A.6 A2Module.jl

```
module A2Module

#Importing A1Module
include("A1Module.jl") # Module written by me, Kim Laberinto
using .A1Module: SwannsBracketingMethod, GoldenSectionSearch

# Other Imports
using LinearAlgebra #External module for taking norm, condition number, and identity
using Memento #Invenia Module for logging
using Printf #External module for formatting strings
using ValueHistories #External Package for keeping track of values

# Exports
export conjugateGradient
export secantLineSearch
export powellsConjugateGradientMethod
export nelderMeadSimplexSearch
export originalNewtonsMethod
export modifiedNewtonsWithLMMethod

# Set up Memento Logger
const LOGGER = getlogger(@__MODULE__)
function __init__()
    Memento.register(LOGGER)
end

function secantLineSearch(grad_f::Function, x_0::Array, d::Array, linesearch_tol::T;
        max_iter::Integer = 100) where T <: Real
    debug(LOGGER, "Entering Secant Line Search")
```

```
    alpha_current = 0.0;
    alpha = 0.1; #Larger initial alpha. No more NaNs
    dphi_zero = grad_f(x_0)' * d
    dphi_current = dphi_zero

    i = 0
    while abs(dphi_current) > linesearch_tol*abs(dphi_zero)
        alpha_old = alpha_current;
        alpha_current = alpha;
        dphi_old = dphi_current;
        dphi_current = grad_f(x_0 .+ alpha_current.*d)' * d;
        alpha = (dphi_current * alpha_old - dphi_old * alpha_current) / (dphi_current - dphi_old);
        i += 1

        if i >= max_iter && abs(dphi_current) < abs(dphi_zero)
            debug(LOGGER, "Secant Line Search Terminating with i=$i")
            break
        end
    end

    full_Nd_point = x_0 .+ alpha_current.*d
    debug(LOGGER, "Exiting Secant Line Search")
    return full_Nd_point
end

function powellsConjugateGradientMethod(f::Function, x_0::Array, tol::T;
    max_iter::Integer = 1000, linesearch_tol = 1e-3) where T <: Real
    info(LOGGER, "Entering Powells Conjugate Gradient")

    search_dir_array = Array{Array}(undef, length(x_0)+1)
    for i in 1:length(x_0)
        search_dir_array[i] = zeros(length(x_0))
        search_dir_array[i][i] = 1
    end

    full_Nd_minimizer, _, _ = A1Module.Q1LineSearch(f, search_dir_array[length(x_0)], x_0, linesearch_tol;
        linesearch_method = "SwannsBracketingMethod")
    X = full_Nd_minimizer;
    C = false;
    k = 0;

    history = MVHistory()
    push!(history, :x_current, 0, x_0)

    while C == false
        Y = X;
        k += 1;
        for i in 1:length(x_0)
            #Keep updating X using line searches in the s_i directions
            # @show search_dir_array[i]
            # @show X
            try
                full_Nd_minimizer, _, _ = A1Module.Q1LineSearch(f, search_dir_array[i], X,
                    linesearch_tol; linesearch_method = "SwannsBracketingMethod")
                X = full_Nd_minimizer;
            catch e
                warn(LOGGER, "Caught an error during Powells Conjugate Gradient with x_0=$x_0. Aborting and returning history. $e")
                push!(history, :x_current, k, X)
                return X, history
            end
        end
        search_dir_array[end] = X .- Y;

        try
            full_Nd_minimizer, _, _ = A1Module.Q1LineSearch(f, search_dir_array[end], X,
                linesearch_tol; linesearch_method = "SwannsBracketingMethod")
            X = full_Nd_minimizer;
        catch e
            warn(LOGGER, "Caught an error during Powells Conjugate Gradient with x_0=$x_0. Aborting and returning history. $e")
            push!(history, :x_current, k, X)
            return X, history
        end

        f_X = f(X)
        f_Y = f(Y)
        if k > max_iter || (abs(f_X - f_Y) / max(abs(f_X), 1e-10)) < tol
            C = true
        else
            for i in 1:length(x_0)
                search_dir_array[i] = search_dir_array[i+1]
            end
```

```
        end

        push!(history, :x_current, k, X)
    end

    info(LOGGER, "Exiting Powells Conjugate Gradient")
    return X, history
end

function conjugateGradient(f::Function, grad_f::Function, x_0::Array,
        g_tol::T, k_max::Integer, n_resetsearchdir::Integer; method = "",
        tol_for_linesearch = 1e-3) where T <: Real
    info(LOGGER, "Entering Conjugate Gradient ($method)")

    k_current = 0;
    x_current = x_0;
    num_resetsearchdir = 0;

    g_old = grad_f(x_current);
    g_new = g_old;

    s_old = g_old;
    s_new = g_old;

    history = MVHistory()
    push!(history, :x_current, 0, x_current)
    push!(history, :grad_norm, 0, norm(g_new))
    push!(history, :num_resets, 0, num_resetsearchdir)

    while (norm(g_new) > g_tol) && (k_current < k_max)
        for i in 1:n_resetsearchdir
            k_current += 1 # NOTE: k moved to inner loop to track all iterations
            if i == 1
                debug(LOGGER, "Reseting Search Direction at k_current=$k_current")
                s_new = -1 * g_new
                num_resetsearchdir += 1
            else
                if method == "FletcherReeves"
                    gamma = norm(g_new)^2 / norm(g_old)^2
                elseif method == "HestenesStiefel"
                    gamma = (g_old' * g_new) / (g_old' * s_old)
                elseif method == "PolakRibiere"
                    gamma = (g_old' * g_new) / norm(g_old)^2
                else
                    error("Undefined method '$method' for Conjugate-Gradient Descent")
                end

                s_new = -g_new + gamma*s_old;
            end

            debug(LOGGER, "Entering Line Search...")
            # full_Nd_minimizer, _, _ = A1Module.Q1LineSearch(f, s_new, x_current,
            #    tol_for_linesearch; linesearch_method = "SwannsBracketingMethod")
            full_Nd_minimizer = secantLineSearch(grad_f, x_current, s_new, tol_for_linesearch)
            # full_Nd_minimizer is already x_current + lambda*s_new
            x_current = full_Nd_minimizer;
            g_old = g_new
            g_new = grad_f(x_current);

            s_old = s_new;

            push!(history, :x_current, k_current, x_current)
            push!(history, :grad_norm, k_current, norm(g_new))
            push!(history, :num_resets, k_current, num_resetsearchdir)
        end
    end

    info(LOGGER, "Exiting Conjugate Gradient ($method)")
    return x_current, history
end

function originalNewtonsMethod(grad_f::Function, hessian_f::Function,
        x_0::Array{T}; g_tol::T = 1e-3, max_iter::Integer = 1000) where T <: Real
    info(LOGGER, "Entering Original Newtons Method")

    k_current = 0;
    num_linsys_solves = 0;

    x_current = x_0;

    g_current = grad_f(x_current);
```

```
    history = MVHistory()
    push!(history, :x_current, 0, x_current)
    push!(history, :g_current, 0, g_current)

    while norm(g_current) > g_tol && k_current < max_iter
        k_current += 1

        num_linsys_solves += 1
        hessian_current = hessian_f(x_current)
        g_current = grad_f(x_current)
        d = hessian_current \ (-g_current)
        x_current += d

        push!(history, :x_current, k_current, x_current)
        push!(history, :g_current, k_current, g_current)
        push!(history, :hessian_current, k_current, hessian_current)
    end

    info(LOGGER, "Exiting Original Newtons Method")
    return x_current, history, num_linsys_solves
end

function modifiedNewtonsWithLMMethod(grad_f::Function, hessian_f::Function,
    x_0::Array{T}; linesearch_tol::T = 1e-3, mu_param::T = 1.0, g_tol::T = 1e-3,
    max_iter::Integer = 1000) where T <: Real
    info(LOGGER, "Entering Modified Newtons Method with LM")

    k_current = 0;
    num_linsys_solves = 0;

    x_current = x_0;

    g_current = grad_f(x_current);

    history = MVHistory()
    push!(history, :x_current, 0, x_current)
    push!(history, :g_current, 0, g_current)

    while norm(g_current) > g_tol && k_current < max_iter
        k_current += 1

        num_linsys_solves += 1
        hessian_current = hessian_f(x_current)
        LM_matrix = (hessian_current + I*mu_param)
        g_current = grad_f(x_current)
        d = LM_matrix \ (-g_current)

        #Do a line search to do the update
        x_current = secantLineSearch(grad_f, x_current, d, linesearch_tol)

        push!(history, :x_current, k_current, x_current)
        push!(history, :g_current, k_current, g_current)
        push!(history, :hessian_current, k_current, hessian_current)
        push!(history, :LM_matrix, k_current, LM_matrix)
    end

    info(LOGGER, "Exiting Modified Newtons Method with LM")
    return x_current, history, num_linsys_solves

end

# https://c.mql5.com/31/43/garch-improved-nelder-mead-mt4-screen-9584.png
function nelderMeadSimplexSearch(f::Function, x_0::Array{T},
    initial_sidelength::T; max_iter::Integer = 1000, stuck_max::Integer = 10,
    stuck_coef::T = 0.5) where T <: AbstractFloat
    info(LOGGER, "Entering Nelder Mead")
    current_vertices = generateSimplex(x_0, initial_sidelength)
    @assert length(current_vertices) == (length(x_0) + 1)

    k_current = 0;

    x_l_old = x_0;
    stuck_counter = 0;

    current_sidelength = initial_sidelength

    history = MVHistory()
    push!(history, :x_best, 0, x_0)

    ALPHA = 1;
```

```
    BETA = 0.5;
    GAMMA = 2;
    while k_current < max_iter
        k_current += 1
        #Evaluate and sort into ascending order.
        #current_vertices[1] will be the best (lowest) vertex
        f_evals = map(f, current_vertices)
        indices_for_sorting = sortperm(f_evals)
        current_vertices = current_vertices[indices_for_sorting]
        f_evals = f_evals[indices_for_sorting]

        x_h = current_vertices[end] #Highest (to replace)
        f_h = f_evals[end]

        x_g = current_vertices[end-1] #Second Highest
        f_g = f_evals[end-1]

        x_l = current_vertices[1] #Lower
        f_l = f_evals[1]

        push!(history, :x_best, k_current, current_vertices[1])

        if x_l_old == x_l
            stuck_counter += 1
        end

        #Define centroid based on other vertices
        x_c = zeros(length(x_0))
        for (i, vertex) in enumerate(current_vertices)
            if vertex != x_h
                x_c += vertex
            end
        end
        x_c /= length(x_0)

        # Do a normal reflection
        x_r = 2*x_c - x_h
        f_r = f(x_r)

        if stuck_counter < stuck_max
            if f_l < f_r < f_g
                theta = ALPHA;
            elseif f_r < f_l
                theta = GAMMA;
            elseif f_r > f_h
                theta = -1 * BETA;
            else # f_g < f_r < f_h
                theta = BETA;
            end
        else
            info(LOGGER, "Shrinking in Nelder Mead")
            stuck_counter = 0 #reset
            for (index, x_old) in enumerate(current_vertices)
                if index >= 2 #only modify the non-x_l vertices
                    current_vertices[index] = (x_old - x_l)*stuck_coef + x_l
                end
            end

            x_l_old = x_l;
            continue #skip to next iteration
        end

        x_l_old = x_l;
        x_new = x_h + (1+ theta)*(x_c - x_h)

        # Replace the largest with x_new
        current_vertices[end] = x_new
    end

    info(LOGGER, "Exiting in Nelder Mead")
    return current_vertices[1], history
end

function generateSimplex(basePoint::Array{T}, side_length::T) where T <: AbstractFloat
    n = length(basePoint)

    a = side_length * ( (sqrt(n+1) + (n - 1)) / (n * sqrt(2)))
    b = side_length * ( (sqrt(n+1) - 1) / (n * sqrt(2)) )

    list_of_points = Array{Array{T}}(undef, n+1)
    for i in 1:n #Generate each point
```

```
        newPoint = Array{T}(undef, n)
        for j in 1:n # Define coordinates of new point
            if i == j
                newPoint[j] = basePoint[j] + a;
            else
                newPoint[j] = basePoint[j] + b;
            end
        end
        list_of_points[i] = newPoint
    end
    list_of_points[end] = basePoint

    @assert length(list_of_points) == (n + 1)
    return list_of_points
end

end
```