Curtis Dibble

Phys 300

12/7/2018

**Optimization**

**Gradient Descent:**

From Wikipedia, the method of Gradient Descent can be described as a ‘first order iterative optimization algorithm for finding the minimum of a function.’ The entire premise of the algorithm is to take steps proportional to the negative of the gradient of a function, given a set of coordinates. Alternatively, there is the method of Gradient Ascent, in which one moves towards a local maximum by taking positive steps in the direction of the gradient of a function. The key component of the algorithm is an accurate calculation of and successive recalculations of [1]

**Conjugate Gradient Descent:**

The Conjugate Gradient method is an algorithm designed to solve ‘unconstrained optimization’ problems. It is typically used in an iterative fashion, specifically systems with a matrix that is symmetric and positive-definite. The application of the formula is mainly for sparse systems and other systems that are too large to be solved using more direct methods. The method of operation is what gives the algorithm its name. it utilizes steps that are conjugate to one another, eg: it uses steps of appropriate length, which are perpendicular to each other. The graph moves in a pattern of right-angle turns, until it homes in on a local minimum. They key component of this method is that the system of equations takes the form [2].

**Quasi-Newton:**

The Quasi-Newton method is a family of algorithms derived from Newton’s method, to find zeroes of a function of multiple variables. [3] It utilizes the left inverse of the Jacobian matrix [7] of the given function. In large part, the method relies on finding zeroes of the gradient of the function. One of the key features of the method is the use of a symmetric Hessian matrix [8]. The Hessian does not actually need to be computed, which is a major efficiency advantage in larger and more complex systems of equations, but it is updated by analyzing the changes in gradient vectors. The path of descent is determined by using a ‘backtracking line search.’[6] The line search criteria is that the chosen step-size must fulfill the Armijo-Goldstein conditions as detailed in the ‘Formulas’ section, under: ‘SR1: #6.’ The most common flavor of Quasi-Newton methods in use today is the Symmetric Rank 1, which I demo in the ‘code’ section of this report. Other versions of Quasi Newton methods include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, the Davidon-Fletcher-Powell (DFP) formula, Broyden, and Broyden Family methods, and the Symmetric Rank 1 (SR1) method. [5]

In order to compare the efficiency of code I wrote to demonstrate Gradient, Conjugate, and Quasi-Newton methods, I ran each algorithm 100 times, using randomly generated coordinates. The data was identical in both order and format for each algorithm tested. I also ran this same group of coordinates through Python’s timeit() function ten times each, for a total of 1000 loops of each algorithm to generate a reliable sample of timing data for comparison. The function used for testing was the one provided in lecture 16 [24], given as:

I excluded a graph of iteration results for my Conjugate Gradient function, as every loop returned a result of ‘one.’ Data generated by each algorithm and used in the attached graphs, as well as the 100 random coordinates can be found as a .txt file on my GitHub page under the title:

*final project x0y0 data.txt.*

In the graphs appendix, I have included a basic set of output graphs to illustrate the methods of each algorithm, as it moves toward convergence. After these basic I/O graphs, I have included the visual results of the 100 loops of testing mentioned previously. Also included are some comparisons to the built-in scipy.minimize.fmin\_cg to show the advantages of community support for a project, and the difference in speed between natively written python code vs. algorithms written in lower level languages that are called from python. For the following segment, ‘accuracy to zero-convergence’ will be measured as the magnitude of the pair, as the analytic zero for the given function is <0.0,0.0,0.0>. Since the Conjugate Gradient took only one iteration per pass, I will focus more on the differences between the Gradient Descent and SR1 algorithms to begin.

For 100 passes, using random data for each pair, and a desired precision of .0001, the Gradient Descent function took an average of 37.1 iterations to converge, with 5.77 seconds of runtime, and converged within 0.00010489 of the function’s zero. As a contrast to this, the SR1 function averaged 27.16 iterations, 6.33 seconds, and an accuracy of 0.00013761 for each pass. Discounting the iteration count, the Conjugate Descent function took approximately .0577 seconds, with an accuracy (or, *in*accuracy) of 1.53133147 distance from expected zero. Searching for outliers to explain the nearly 10 iteration discrepancy between the SR1 and Gradient Descent functions, I averaged every individual iteration. The results were identical, with a difference of 9.98 iterations per loop, in favor of the SR1 method. To emphasize the importance of using well maintained, and well built library code, I tested the scipy.minimize.fmin\_cg function with much better results than what my hand-built functions provided. The average magnitude of distance from true zero was reduced significantly, to: 9.414237356934639e-08. The minimum point f\_min was brought down to a significant degree on average to 1.858472014830568e-12, and the average number of iterations was not far removed from the SR1 function at 29.52 iterations per loop. I was able to also get runtime data, and it is significantly faster than the native python code which I wrote. The scipy algorithm clocks in with an average of 0.0237 seconds per loop. This is about 267x faster than the SR1 function, and about 243x faster than the Gradient Descent function. There really is no comparison in any category against the Conjugate Descent function. Although it clocks an amazing .0577 seconds per loop, it is wildly inaccurate when held up against the builtin scipy function, it has about 1.6266e-13% the accuracy.

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Avg Runtime | Avg Iterations | Avg Accuracy f\_min |
| Scipy | 0.0237s | 29.52 | 1.85847e-12 |
| Gradient Descent | 5.77 | 37.1 | 0.00010489 |
| Conj. Descent | .0577 | 1 | 1.53133147 |
| SR1 | 6.33 | 27.16 | 0.00013761 |

**Gradient Descent Mathematical Description [1]:**

Note: Gradient Descent relies on a function being differentiable, and works by making steps towards a convergence point, better known as a ‘local maxima/minima’. Mathematically, it can be represented as follows:

1. Multivariable Function defined as , is differentiable in a neighborhood of
2. decreases fastest if following the negative gradient of at , eg:
3. For: , ,

then ,

we move against the gradient towards a minima

1. Beginning with an initial guess of , and using the sequence:

1. We obtain the algorithm:

1. And the monotonic sequence:

1. The sequence is shaped, and obtained via using a line search,

satisfying the Wolfe conditions: [10]

, and:

:

cosine of angle between and , bounded away from zero

or Barzilai-Borwein conditions:

**Conjugate Gradient Descent Mathematical Description (iterative algorithm) [2] :**

1. This algorithm is specifically for matrix equations of the form:
2. Where **A** is a real, symmetric, positive definite matrix
3. Set the following initial conditions:
4. Repeat the following system in order:
5. if is sufficiently small, you’re done
6. Else:

1. The result is

**Preconditioned Conjugate Gradient Descent Mathematical Description [2]:**

**The Preconditioner [11]:**

1. Utilize standard Richardson Iteration -which is quite similar to the standard Gradient Descent formula- for solving:

1. Application to the preconditioned system yields:
2. The Jacobi preconditioner (for utilization of the Jacobian Matrix) is one of the simplest forms of this method.
3. The preconditioner is chosen to be the diagonal of the matrix:
4. Or we can utilize an incomplete Cholesky Factorization via the following [12]:
   * 1. For i, from 1:N,
     2. And then for, from :
5. This provides a matrix A, where A= LL\*, with L being the lower triangular matrix. The incomplete version replaces ‘L’ with ‘K’, to provided A=KK\*, and any entry in ‘A’ that is zero, is also set to zero in ‘K’.
6. This changes the algorithm to:
   * 1. For i, from 1:N,
     2. And then for, from :

**The Algorithm with Preconditioner [2]:**

1. To utilize a preconditioner with the algorithm, eg: to transform the problem into something more suitable for numerical analysis/solution, perform the following:
2. Set the initial conditions to:
3. Where, or from the Preconditioning Algorithm
4. Repeat the following system in order:
5. if is sufficiently small, you’re done:
6. else:

1. As it was with the non-conditioned formula, the result is

**Newton’s Method Mathematical Description [4]:**

This is the algorithm developed by Newton, from which all Quasi-Newton algorithms are derived, this is the implementation in a single variable equation.

1. Begin with a real-valued function, along the lines of
2. Will usually converge if is close enough to unknown zero and
3. The function is then approximated by its tangent line, and the x-intercept is computed
4. Taking a function , that is differentiable across the interval , with a current approximation , a better approximation can be found at .
5. The tangent line for at is:
6. With an x intercept such that , is used to approximate the root
7. With:
8. Repeat until a zero is found

Difficulties can include, but are not limited to:

1. Can be difficult if derivative is not easily attainable
2. Can overshoot and diverge if derivative is not well behaved near a zero, derivative is discontinuous, no second derivative, or initial point is not a good choice
3. Can terminate unexpectedly if stationary point is found
4. Can loop infinitely if equation enters a cycle

**Quasi-Newton Mathematical Description [5]:**

1. The basic method looks like this, using a Taylor Series, and is the approximate Hessian:
2. is:
3. Set to obtain:
4. Approximate B carefully to satisfy this equation, or the ‘secant equation’:
5. Aim for a symmetric solution , with being positive-definite, but will usually suffice for rapid convergence
6. Update using by:

to compute or

1. My example follows the Symmetric Rank 1 method, which utilizes the following to update the Hessian, and inverse Hessian approximation matrices:

**Symmetric Rank 1 Mathematical Description [3]:**

The algorithm which I followed to write my own Quasi-Newton optimizer follows this pattern [19]:

1. Given , initial positive matrix (Identitiy)
2. Begin with the following initial values:
3. Use the termination test to determine convergence:
4. If #3 == True, stop
5. Compute direction, :
6. Find steplength using a backtracking line search, such as the Armijo-Goldstein algorithm [6]:
7. Return as the steplength
8. Set by:
9. Set the inverse Hessian approximation by:
10. Where:
11. And

**Gradient Descent Code:**

This method is the simplest of the three to implement.

It can usually find a convergence point, but has a tendency to get stuck, especially in more complex functions such as the minima in the Ackley function.

Here is my version in python:

# Conjugate Gradient Descent

# Accept the function, its limits,

# and the expected value provided by benchmark

# x(n+1) = x[n] - gamma[n]\*GradF(x[n])

# variable stepsize

def GD\_min1(f,xlims, ylims):

# N = Number of intervals

# xa = storage array for values to return

# i = counter

# init(ax,bx,ay,by) = function call to gen random

# values for initial guess

# prec = .0001

# gamma = prec is initial step size so algo

# doesn’t overshoot on the first run

N = 1000

xa = []

i = 0

ax,bx = xlims[0],xlims[1]

ay,by = ylims[0],ylims[1]

x\_now = init(ax,bx,ay,by)

gamma = prec

converged = False

xa.append(x\_now)

# loop the algo until the term\_test conditions

# are satisfied and it returns a ‘True’ value

while converged == False or i < N:

converged = term\_test(x\_now,f)

if converged == True:

break

else:

df = nd.Gradient(f)

x\_next = x\_now - gamma\*df(x\_now)

a = (x\_next - x\_now)

b = a.T

c = (df(x\_next)- df(x\_now))

gamma = b\*c/np.linalg.norm(c)\*\*2

x\_now = x\_next

xa.append(x\_now)

i += 1

# reshape ‘xa’ as a (-1,2) array to return

# f\_min: return the minimum value, or how close

# the algo got to zero

# return the number of iterations

# for comparison later

xa = np.array(xa)

f\_min = f(x\_now)

return xa, i, f\_min

**Conjugate Gradient Descent Code:**

Ideal method for sparse matrices and systems of non-linear equations. Incredibly efficient when the logic is correct. This implementation claims convergence in a single iteration and converges in about 10% of the time that the Gradient Descent and SR1 algorithms take.

def CD\_min1(f,xlims, ylims):

# init x value statements,

# and the return value array

ax,bx = xlims[0],xlims[1]

ay,by = ylims[0],ylims[1]

x\_now = init(ax,bx,ay,by)

xa = []

np.asarray(xa)

# counters

k = 0

# inital values

x0 = x\_now

A = nd.Hessian(f)

A = A(x\_now)

# internal functions

def CD\_algo\_init(x):

r = -A\*x

p = r

return r,p

def a\_now(rk,pk):

ak = rk.T\*rk/(pk.T\*A\*pk)

return ak

def x\_k1(xk,ak,pk):

xk1 = xk+ak\*pk

return xk1

def r\_k1(rk,A,ak,pk):

rk1 = rk - ak\*A\*pk

return rk1

def Beta\_k(rk1,rk):

Bk = rk1.T\*rk1/(rk.T\*rk)

return Bk

def p\_k1(rk1,Bk,pk):

pk1 = rk1 +Bk\*pk

return pk1

# iterate through algo until a zero appears

rk, pk = CD\_algo\_init(x0)

while np.abs(f(x\_now)) > prec or k < N:

ak = a\_now(rk,pk)

x\_next = x\_k1(x\_now,ak,pk)

rk1 = r\_k1(rk,A,ak,pk)

if np.linalg.norm(rk1) <= prec:

xa.append(x\_next)

k += 1

break

else:

Bk = Beta\_k(rk1,rk)

pk = p\_k1(rk1,Bk,pk)

x\_now = x\_next

xa.append(x\_now)

k += 1

# reshape the data from a mixed list

# of floats and arrays to a list of

# useable values

for i in range(k):

x = xa[i]

x1 = x[0]

x1a = x1[0]

x1b = x1[1]

x2 = x[1]

x2a = x2[0]

x2b = x2[1]

x0.append(x1a)

x0.append(x1b)

x0.append(x2a)

x0.append(x2b)

# reshape list into an array of coordinates

x0 = np.asarray(x0, dtype = np.float32)

x0.shape = (-1,2)

return x0, k

**Symmetric Rank 1 – Quasi-Newton Code:**

This is the SR1 implementation of the Quasi-Newton method. Relies on calculating and updating inverse-Hessian matrix of function, and a backtracking line-search to determine distance to step, based on direction of step towards minima. Very accurate and efficient.

# Quasi-Newton

# Symmetric Rank-1

# Accept the function, its limits,

# and the expected value

# provided by benchmark

def Rank1\_min(f, xlims, ylims):

xa = []

ax,bx = xlims[0],xlims[1]

ay,by = ylims[0],ylims[1]

x\_now = init(ax,bx,ay,by)

x\_now = np.asarray(x\_now)

xa.append(x\_now)

converged = False

G = nd.Gradient(f)

H = nd.Hessian(f)

# initial function values

x0 = x\_now

f0 = f(x0)

g0 = G(x0)

H0 = np.identity(2)

k = 0

L = 1

# termination condition

epsilon = prec

# Direction d(k)

def dk(H,xk):

return -H\*G(xk)

# Step Size a(k)

def a\_now(H,xk,f):

a = np.linalg.norm(f(xk))

tau = np.linspace(1,0,N)

c = np.linspace(0,1,N)

c = c[np.random.randint(0,N)]

m = dk(H,xk)

m = array\_fix(m)

p = m

m = np.dot(np.transpose(m),G(xk))

t = -c\*m

j = 0

a\_converge = False

# Backtracking Line Search to find

# appropriate value of ‘a’

while a\_converge == False or j <= N:

ajt\_test = f(xk) - f(xk +np.multiply(a,p))

j += 1

if a\*t <= ajt\_test:

a\_converge = True

return a

if j == N:

return a

else:

a = a\*tau[j]

# Compute x(k+1)

def x\_next(xk,f,H):

direction = dk(H,xk)

direction = array\_fix(direction)

return xk + a\_now(H,xk,f)\*direction

# Delta x(k)

def delta\_xk(xk1, xk):

return xk1 - xk

# Delta x(k), or change in the gradient

def delta\_gk(xk,xk1):

return H(xk1 - xk)

# H(k+1) = H + some stuff

def H\_formula(H,xk,xk1,f):

a = delta\_gk(xk,xk1)

b = delta\_xk(xk,xk1)

c = H\*a

d = b - c

e = (H + d\*d.T/a.T/c)

return e

def algo(xk,H,f):

xk1 = x\_next(xk,f,H)

Hk1 = H\_formula(H,xk,xk1,f)

return xk1, Hk1

x\_now, H\_now = algo(x\_now,H0,f)

xa.append(x\_now)

converged = term\_test(x\_now,f)

k += 1

while converged == False:

x\_now, H\_now = algo(x\_now, H\_now,f)

xa.append(x\_now)

k += 1

converged = term\_test(x\_now,f)

xa = np.array(xa)

xa.shape = (-1,2)

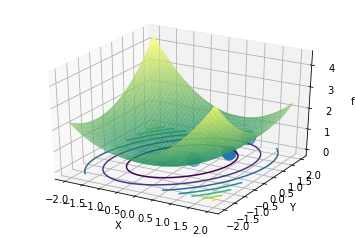
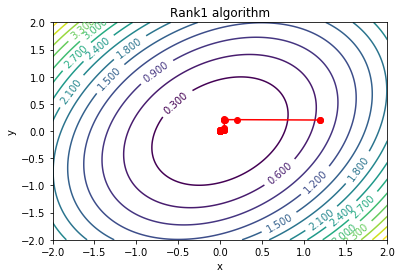
f\_min = f(x\_now)

return xa, k

**Basic Comparison of three algorithms using the same coordinates:**

**Initial coordinates:** [1.1992234483300366, 0.20307928229770278]

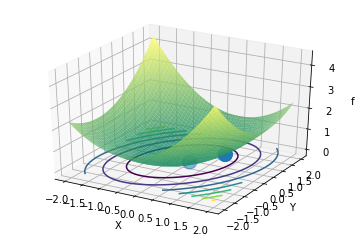
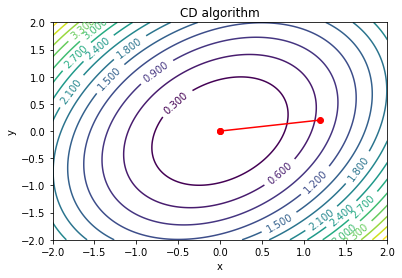
**Gradient Descent**



# of iterations: 27

f\_min found: 1.576510746288387e-07

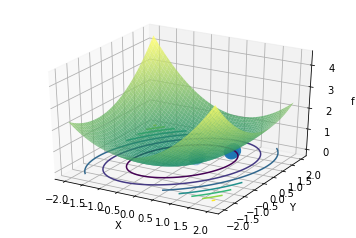
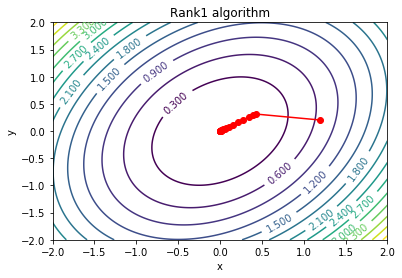
**Conjugate Gradient Descent**



# of iterations: 1

f\_min found: 0.6719311451781153

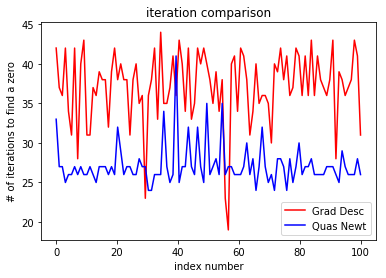
**Symmetric Rank 1**

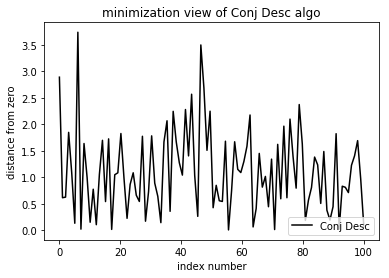
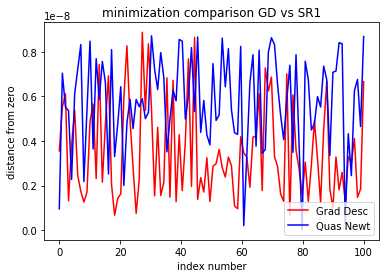


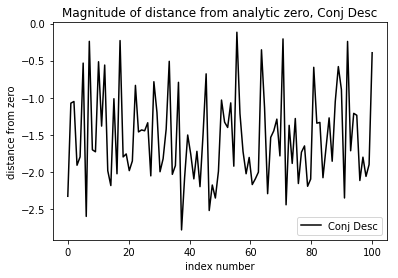
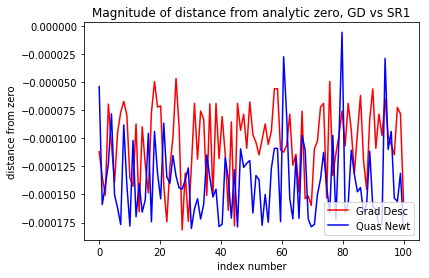
# of iterations: 21

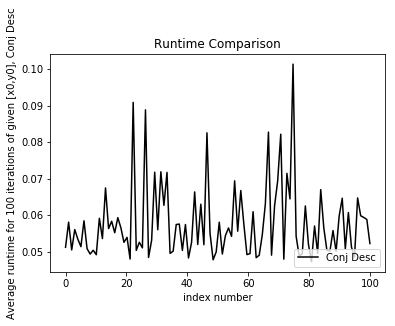
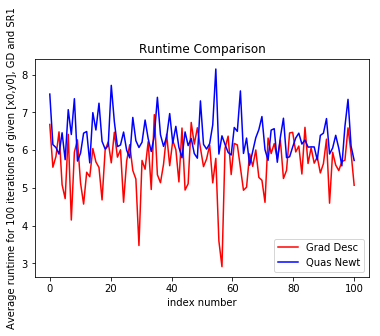
f\_min found: 3.5916250519422733e-07

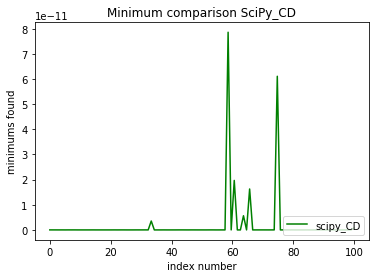
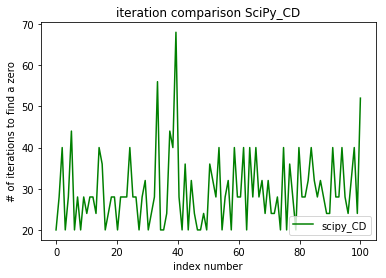
**Comparison across 100 Loops:**

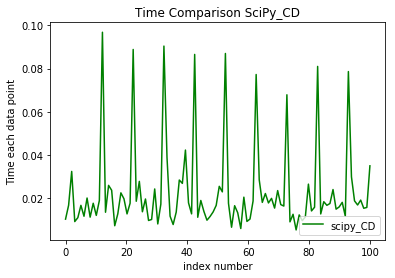












**References and Works Cited:**

1. “Gradient Descent.” Wikipedia, Wikimedia Foundation, 12 Nov. 2018, en.wikipedia.org/wiki/Gradient\_descent. Algorithm, Background, Formulas
2. “Conjugate Gradient Method.” Wikipedia, Wikimedia Foundation, 29 Nov. 2018, en.wikipedia.org/wiki/Conjugate\_gradient\_method. Algorithm, Formulas, Reference, Theory
3. “Symmetric Rank-One.” Wikipedia, Wikimedia Foundation, 1 Dec. 2018, en.wikipedia.org/wiki/Symmetric\_rank-one. SR1 Algorithm, Formulas, Theory
4. “Newton's Method.” Wikipedia, Wikimedia Foundation, 5 Dec. 2018, en.wikipedia.org/wiki/Newton's\_method. Formulas, Description, Theory, History of Quasi-Newton methods
5. “Quasi-Newton Method.” Wikipedia, Wikimedia Foundation, 11 Sept. 2018, en.wikipedia.org/wiki/Quasi-Newton\_method. List of Various QN methods
6. “Backtracking Line Search.” Wikipedia, Wikimedia Foundation, 16 Mar. 2017, en.wikipedia.org/wiki/Backtracking\_line\_search. Armijo-Goldstein conditions and algorithm
7. “Jacobian Matrix and Determinant.” Wikipedia, Wikimedia Foundation, 29 Nov. 2018, en.wikipedia.org/wiki/Jacobian\_matrix\_and\_determinant. Formulas and background
8. “Hessian Matrix.” Wikipedia, Wikimedia Foundation, 3 Dec. 2018, en.wikipedia.org/wiki/Hessian\_matrix. Reference, Formulas
9. “Test Functions for Optimization.” Wikipedia, Wikimedia Foundation, 22 Oct. 2018, en.wikipedia.org/wiki/Test\_functions\_for\_optimization. Ackley, Beale, Cross-in-Tray, Eggholder, Holder Table functions and details
10. “Wolfe Conditions.” Wikipedia, Wikimedia Foundation, 8 May 2017, en.wikipedia.org/wiki/Wolfe\_conditions. Algorithm, Formulas, Background
11. “Preconditioner.” Wikipedia, Wikimedia Foundation, 21 July 2018, en.wikipedia.org/wiki/Preconditioner. Preconditioner Algorithm
12. “Cholesky Decomposition.” Wikipedia, Wikimedia Foundation, 13 Nov. 2018, en.wikipedia.org/wiki/Cholesky\_decomposition. Formulas and reference
13. Brockhoff, Dimo. “Introduction to Optimization Introduction to Continuous Optimization III / Gradient-Based Algorithms.” Cmapx Polytechnique, Cmapx Polytechnique, 20 Nov. 2015, www.cmapx.polytechnique.fr/~dimo.brockhoff/introoptimization/2015/slides/09-introductiontocontopt3.pdf. Pseudocode and Formulas
14. Brodtkorb, Per A, and John D’Errico. “Numdifftools Documentation.” Readthedocs.org, Readthedocs.org, 1 Nov. 2018, media.readthedocs.org/pdf/numdifftools/latest/numdifftools.pdf. Used to simplify computaton of Hessian/Gradient, rapidly compute derivatives
15. Chu, Wei-Ta. “Chapter 11 Quasi-Newton Methods - An Introduction to Optimization.” National Chung Chang University, Department of Computer Science and Information Engineering, 2014, www.cs.ccu.edu.tw/~wtchu/courses/2014s\_OPT/Lectures/Chapter 11 Quasi-Newton Methods.pdf. Approximating Inverse Hessian, Formulas, Algorithms, Reference Material
16. Dennis, John, and Jorge Moré. “Quasi-Newton Methods, Motivation and Theory.” Hal.archives-Ouvertes, SIAM Review, Society for Industrial and Applied Mathematics, 26 Mar. 2017, hal.archives-ouvertes.fr/hal-01495720/document. HAL Id: hal-01495720 Algorithms, Formulas, Theory
17. “Design Optimization.” Optimization Method: FOPDT to Data, Brigham Young University, 26 Feb. 2013, apmonitor.com/me575/index.php/Main/QuasiNewton. Code Examples of QN, GD, CD, and visualization of graphs
18. Kuzovkin, Ilya. “The Concept of Conjugate Gradient Descent in Python.” The Concept of Conjugate Gradient Descent in Python, 15 Apr. 2015, ikuz.eu/2015/04/15/the-concept-of-conjugate-gradient-descent-in-python/. Code Examples, Graph Examples, Theory
19. Modarres, Farzin, et al. “Multi-Steps Symmetric Rank-One Update for Unconstrained Optimization.” Semanticscholar.org, Department of Mathematics, Faculty of Science, University Putra Malaysia, 2009, pdfs.semanticscholar.org/d6e4/a8bae8136e869717230eb8a38d143394e02a.pdf. Basis for my SR1 Algorithm - page 613
20. “Quasi-Newton Methods.” NIntegrate-Wolfram Language Documentation, Wolfram, reference.wolfram.com/language/tutorial/UnconstrainedOptimizationQuasiNewtonMethods.html. Quasi-Newton Variant comparisons - BFGS method
21. Refsnæs, Runar Heggelien. “A BRIEF INTRODUCTION TO THE CONJUGATE GRADIENT METHOD.” NTNU , Dept of Computer Science, 2009, www.idi.ntnu.no/~elster/tdt24/tdt24-f09/cg.pdf. Broad based description and comparison
22. “Scipy.optimize.fmin\_cg¶.” Scipy.stats.trim\_mean - SciPy v1.1.0 Reference Guide, docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.fmin\_cg.html. Used function as a benchmarkSingh, Aarti.
23. “Conjugate Gradient Descent .” Carnegie Mellon University, Carnegie Mellon University CS Dept, www.cs.cmu.edu/~pradeepr/convexopt/Lecture\_Slides/conjugate\_direction\_methods.pdf. Algos/reference.Line Search
24. Zhu, Qiang. “qzhu2017/2017-Cmp.” GitHub, Oct. 2018, github.com/qzhu2017/2017-cmp/blob/master/Lec\_16\_optimization2.ipynb. Assorted Graphs, derivative2 function