Computational Mathematics for Learning and Data Analysis
Project Report A.Y. 2020/2021

Dawit Anelay, Marco Petix, and Yohannis Telila

Department of Computer science, University of Pisa, Pisa, Italy.

January 17, 2022

Contents

Abstract

(P) is the problem of estimating the matrix norm $\|A\|_2$, for a matrix $A \in R^{m\times n}$, using its definition as an (unconstrained) maximum problem.
(A1) is a conjugate gradient descent algorithm.
(A2) is a quasi-Newton method such as BFGS (one which does not require any approximations of the Hessian of the function).
(A3) is approximating the largest eigenvalue of A^TA (or AA^T) using the Arnoldi process.

1 The 2-Norm of a Matrix as a Maximization Problem

We present the definition of the induced matrix 2-norm:

$$\|\boldsymbol{A}\|_2^2 := \sup_{\boldsymbol{x}\in\mathbb{R}^n_{\neq 0}} \frac{\|\boldsymbol{A}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \tag{1.1}$$

where the 2-norm is defined as:

$$\|\boldsymbol{x}\|_2 := \sqrt{\boldsymbol{x}^T\boldsymbol{x}} \tag{1.2}$$

and $\boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{x} \in \mathbb{R}^n_{\neq 0}$.

By combining the previous definitions we can further detail the equation for $\|\boldsymbol{A}\|_2$ with:

$$\|\boldsymbol{A}\|_2 = \sup_{\boldsymbol{x}\in\mathbb{R}^n_{\neq 0}} \sqrt{\frac{(\boldsymbol{A}\boldsymbol{x})^T\boldsymbol{A}\boldsymbol{x}}{\boldsymbol{x}^T\boldsymbol{x}}} = \sup_{\boldsymbol{x}\in\mathbb{R}^n_{\neq 0}} \sqrt{\frac{\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}}{\boldsymbol{x}^T\boldsymbol{x}}} \tag{1.3}$$

Due to the monotone nature of the square root function, we can write :

$$\|\boldsymbol{A}\|_2 = \sup_{\boldsymbol{x}\in\mathbb{R}^n_{\neq 0}} \frac{\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}}{\boldsymbol{x}^T\boldsymbol{x}} \tag{1.4}$$

in order to express the estimation of the norm as the following unconstrained minimization problem:

$$\|\boldsymbol{A}\|_2 = \inf_{\boldsymbol{x}\in\mathbb{R}^n_{\neq 0}} -f(\boldsymbol{x}) \tag{1.5}$$

Ultimately, we can define our objective function as:

$$f(\boldsymbol{x}) = -\frac{\boldsymbol{x}^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}} \tag{1.6}$$

2 Properties of the Objective Function

2.1 Properties of $\boldsymbol{A}^T\boldsymbol{A}$

Given a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, the followings properties are valid for the matrix $\boldsymbol{A}^T\boldsymbol{A}$ (and $\boldsymbol{A}\boldsymbol{A}^T$):

- The matrix is square: $\boldsymbol{A}^T\boldsymbol{A} \in \mathbb{R}^{n\times n}$
- The matrix is symmetric: $(\boldsymbol{A}^T\boldsymbol{A})^T = \boldsymbol{A}^T(\boldsymbol{A}^T)^T = \boldsymbol{A}^T\boldsymbol{A}$
- The matrix is positive semi-definite for any $\boldsymbol{x} \in \mathbb{R}^n$: $\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = (\boldsymbol{A}\boldsymbol{x})^T\boldsymbol{A}\boldsymbol{x} = \|\boldsymbol{A}\boldsymbol{x}\|_2 \geq 0$

According to both the Spectral Theorem, which deals with the decomposition of the eigenvalues of symmetric matrices, and to the positive semi-definiteness of $\boldsymbol{A}^T\boldsymbol{A}$, we recognize said matrix as possessing only real and not negative eigenvalues.
Ultimately, the properties of norms and matrices define the relationship between the spectral radius of $\boldsymbol{A}^T\boldsymbol{A}$, being the largest eigenvalues possessed by said matrix, and the 2-norm of matrix $\boldsymbol{A}$ as:

$$\|\boldsymbol{A}\|_2 = \sqrt{\rho(\boldsymbol{A}^T\boldsymbol{A})} \tag{2.1}$$

2.2 Continuity, Differentiability and Gradient of the Function

The domain of the objective function is $\mathbb{R}^n_{\neq 0}$ and, being it composed by simple and continuous functions, it results continuous within the same.

In order to compute the partial derivatives of the function at each step i, we express it as

$$f(\boldsymbol{x}) = -\frac{\boldsymbol{x^T M x}}{\boldsymbol{x^T x}} = -\frac{\sum_{i=1}^n \sum_{j=1}^n x_i m_{ij} x_j}{\sum_{i=1}^n x_i^2} \tag{2.2}$$

where m_{ij} are the elements of $\boldsymbol{M = A^T A}$.

$$\begin{aligned}
\frac{\partial f}{\partial x_k} &= -\frac{(\boldsymbol{x^T M x})'(\boldsymbol{x^T x}) - (\boldsymbol{x^T M x})(\boldsymbol{x^T x})'}{(\boldsymbol{x^T x})^2} \\
&= -\frac{(\boldsymbol{2Mx})(\boldsymbol{x^T x}) - (\boldsymbol{x^T M x})2x}{(\boldsymbol{x^T x})^2} \\
&= \frac{2x(\boldsymbol{x^T M x})}{(\boldsymbol{x^T x})^2} - \frac{\boldsymbol{2Mx}}{\boldsymbol{x^T x}}
\end{aligned}$$

We can write our partial derivative result as:

$$\frac{\partial f}{\partial x_k} = \frac{2x_k(\sum_{i=1}^{n}\sum_{j=1}^{n} x_i m_{ij} x_j)}{(\sum_{i=1}^{n} x_i^2)^2} - \frac{\sum_{j=1}^{n} m_{kj} x_j}{\sum_{i=1}^{n} x_i^2} = \frac{2x_k(\boldsymbol{x^T M x})}{(\boldsymbol{x^T x})^2} - \frac{\boldsymbol{2Mx}}{\boldsymbol{x^T x}} \tag{2.3}$$

The derivative of our objective function exist, are continuous and differentiable in all points expect 0. The objective function is therefore differentiable in $\mathbb{R}^n \setminus \{0\}$.
We can write the gradient of the function as

$$\nabla f(\boldsymbol{x}) = \frac{2\boldsymbol{x}(\boldsymbol{x^T M x})}{(\boldsymbol{x^T x})^2} - \frac{\boldsymbol{2Mx}}{\boldsymbol{x^T x}} \tag{2.4}$$

2.3 Complexity of the Function and its Gradient

Computing $\boldsymbol{M}$ would require $O(m^2 n)$ operations but its symmetric nature simplifies this process by halving the amount of elements actually needed, also, such computation is only required once.

Computing $\boldsymbol{x^T M}$ instead requires $O(n^2)$ operations and the same is valid for the computation of $(\boldsymbol{x^T M})\boldsymbol{x}$ too. Finally, computing $\boldsymbol{x^T x}$ and the division take $O(n)$ and 1 operations.
The total complexity of the function is therefore

$$C(f(\boldsymbol{x})) = 2O(n^2) + O(n) + 1 = O(n^2) \tag{2.5}$$

Computing the first term of the gradient takes $O(n^2) + O(n^2) + 1$ operations, computing the second term instead takes $3O(n^2) + 2O(n^2) + 1$. Ultimately, the total complexity of the gradient of the function is

$$C(\nabla f(\boldsymbol{x})) = 7O(n^2) + 3 = O(n^2). \tag{2.6}$$

It's still possible to reduce such complexity by rewriting the gradient as

$$\nabla f(\boldsymbol{x}) = \frac{\boldsymbol{2Mx}}{\boldsymbol{x^T x}} - \frac{2\boldsymbol{x}(f(\boldsymbol{x}))}{\boldsymbol{x^T x}} = \frac{\boldsymbol{Mx} - 2\boldsymbol{x}(f(\boldsymbol{x}))}{\boldsymbol{x^T x}}. \tag{2.7}$$

This enables the use of the values already computed for $f(\boldsymbol{x})$, $\boldsymbol{Mx}$ and $\boldsymbol{x^T x}$ and limits the task of obtaining the gradient to the single computation of $\boldsymbol{x}f(\boldsymbol{x})$.
The complexity of computing the gradient, after applying the aforementioned transformations, is

$$C(\nabla f(\boldsymbol{x})) = O(n). \tag{2.8}$$

2.4 Stationary Points

Another important property we want to know about our function is whether it has a stationary point because we want our algorithm to converge to these points. Lets say our stationary point $\hat{\boldsymbol{x}} \neq \boldsymbol{0}$, we are interested to find where $\nabla f(\hat{x}) = 0$

$$\nabla f(\hat{\hat{\boldsymbol{x}}}) = \frac{\boldsymbol{2M\hat{x}}}{\boldsymbol{\hat{x}^T \hat{x}}} - \frac{2\boldsymbol{\hat{x}}(\boldsymbol{\hat{x}^T M \hat{x}})}{(\boldsymbol{\hat{x}^T \hat{x}})^2} = \frac{\boldsymbol{2M\hat{x}}(\boldsymbol{\hat{x}^T \hat{x}}) - 2\boldsymbol{\hat{x}}(\boldsymbol{\hat{x}^T M \hat{x}})}{(\boldsymbol{\hat{x}^T \hat{x}})^2} = 0 \tag{2.9}$$

$$\mathbf{2}\boldsymbol{M}\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\hat{\boldsymbol{x}}) - 2\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\boldsymbol{M}\hat{\boldsymbol{x}}) = 0 \tag{2.10}$$

$$\mathbf{2}\boldsymbol{M}\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\hat{\boldsymbol{x}}) = 2\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\boldsymbol{M}\hat{\boldsymbol{x}}) \tag{2.11}$$

We can remove 2 from both sides.

$$\boldsymbol{M}\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\hat{\boldsymbol{x}}) = \hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\boldsymbol{M}\hat{\boldsymbol{x}}) \tag{2.12}$$

$$\frac{\hat{\boldsymbol{x}}(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\boldsymbol{M}\hat{\boldsymbol{x}})}{(\hat{\boldsymbol{x}}^{\boldsymbol{T}}\hat{\boldsymbol{x}})} = \boldsymbol{M}\hat{\boldsymbol{x}} \tag{2.13}$$

$$f(\hat{\boldsymbol{x}})\hat{\boldsymbol{x}} = \boldsymbol{M}\hat{\boldsymbol{x}} \tag{2.14}$$

We can say that $\hat{x}$ is stationary point for our function if and only if $\hat{x}$ is eigen vector for $\boldsymbol{M}$ and its eigenvalue is $f(\hat{\boldsymbol{x}})$.

2.5 Bounds on the Function and descent direction

Another important property of our function is that the function is bounded from below.

$$\exists D \in \mathbb{R}^{+} s.t f(x) \geq D$$

$$\|-f(x)\| = \left\| \frac{x^T M x}{x^T x} \right\|$$

We can expand the norm using Cauchy-Schwarz inequality as follow

$$\left\| \frac{x^T M x}{x^T x} \right\| \leq \frac{\left\|x^T\right\| \|M\| \|x\|}{\|x\|^2} \leq \|M\| \leq D \tag{2.15}$$

Next we discuss the closed formula to compute the step-size α^i at given point x^i at the i^{th} iteration along the direction d^i.

$$\varphi(\alpha^i) = \arg\min f(x + \alpha^i d^i)$$

So we are looking for the minimum of this function $\varphi(\alpha^i)$ and the minimum of the function can be found by solving $\varphi'(\alpha^i) = 0$ equation. We have the equation for $f(x)$ and $\nabla f(x)$ from equation 8 and 10.
If $\mathbf{M}$ is positive semi-definite, $x^T\mathbf{M}x \geq 0$ and $f(x)$ will be a maximization problem.

$$\begin{aligned}
&\varphi(\alpha) = f(x(\alpha)),\ \textit{where}\ x(\alpha) = x + \alpha d \\
&f(x(\alpha)) = \frac{-(x+\alpha d)^T M (x+\alpha d)}{(x+\alpha d)^T (x+\alpha d)} \\
&\varphi(\alpha) = -(\frac{x^T M x + \alpha(d^T M x + x^T M d) + \alpha^2 (d^T M d)}{x(\alpha)^T x(\alpha)})
\end{aligned}$$

But $x^T M d \in \mathbb{R}$ and M is symmetric, means that $x^T M d = (x^T M d)^T = d^T M^T x = d^T M x$. Hence

the above equation can be written as below.

$$\varphi(\alpha) = -(\frac{x^T Mx + 2\alpha d^T Mx + \alpha^2 d^T Md}{x(\alpha)^T x(\alpha)})$$

$$\varphi'(\alpha) = -[\frac{(2d^T Mx + 2\alpha d^T Md)(x(\alpha)^T x(\alpha)) - ((x^T Mx + 2\alpha d^T Mx + \alpha^2 d^T Md)(2(x+\alpha d)^T d))}{(x(\alpha)^T x(\alpha))^2}]$$

$$\begin{aligned} &= -\big[2\alpha^3((d^T Md)d^T d) + 2\alpha^2((d^T Mx)d^T d + 2(d^T Md)d^T x) + \\ &\quad 2\alpha((d^T Md)x^T x) + 2((d^T Mx)d^T x) + 2((d^T Mx)x^T x)\big] + \\ &\quad \big[2\alpha^3((d^T Md)d^T d) + 2\alpha^2((d^T Mx)d^T d + 2(d^T Mx)d^T d) + \\ &\quad \frac{2\alpha(2(d^T Mx)(x^T d)) + (x^T Mx)d^T d) + 2((x^T Mx)x^T d)\big]}{(x(\alpha)^T x(\alpha))^2} \end{aligned}$$

By simplifying the above equation we would have

$$\varphi'(\alpha) = \frac{2\alpha^2((d^T Mx)d^T d - (d^T Md)d^T x) + 2\alpha((x^T Mx)d^T d) - (d^T Mx)x^T x) + 2(x^T Mx)x^T d - (d^T Mx)x^T x}{((x+\alpha d)^T (x+\alpha d))^2}$$

We can write our equation in a short form like this

$$\varphi'(\alpha) = 2.\frac{\alpha^2 a + \alpha b + c}{R(\alpha)}$$

Where,

$$\begin{aligned} a &= (d^T Mx)d^T d \\ b &= ((x^T Mx)d^T d - (d^T Md)(x^T x)) \\ c &= (x^T Mx)x^T d - (d^T Mx)x^T x \\ R(\alpha) &= (x(\alpha)^T x(\alpha))^2 \end{aligned} \tag{2.16}$$

To discuss the complexity of computing this, In general to calculate the parameters a,b and c it takes $\mathcal{O}(n^2)$ because obtaining the value for $x^T Mx$ and $d^T Md$ requires $\mathcal{O}(n^2)$ complexity.

$$\mathcal{C}(\varphi'(\alpha)) = \mathcal{O}(n^2) \tag{2.17}$$

2.6 Non Convexity of the Function

Our function cannot be convex because the only function that is bounded and convex is a constant function; Our function is bounded but not constant for this reason we can say that our function is not convex.

If we make an assumption that $f(x)$ is convex, $\nabla f(x)$ exists almost everywhere and the following

inequality holds.

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle \quad \forall x, y \in R^n$$

If we take two different eigenvalues $\tilde{x}_i$ and $\tilde{x}_j$ with $f(\tilde{x}_i)$ and $f(\tilde{x}_j)$ eigenvectors and if we assume $\tilde{x}_i > \tilde{x}_j$, then $f(\tilde{x}_i) > f(\tilde{x}_j)$. But from the convex inequality assumption stated above, we would get $f(\tilde{x}_i) < f(\tilde{x}_j)$, which contradicts the convexity hypothesis.

3 Conjugate gradient method

Conjugate gradient method is one of the optimization techniques. In this section we will be showing how we can use this method to optimize our function. The algorithm of the conjugate gradient method is defined as below.

Algorithm 1 Pseudo-code for Conjugate Gradient Method

procedure CGM(x_0, ϵ) ▷ The cgm
  $d_0 \leftarrow -\nabla f(x_0)$
  while $\|\nabla \boldsymbol{f(x_k)}\| > \epsilon$ **do**
    $\beta_k = \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})}$
    $d_k = -\nabla f(x_k) + \beta_k d_{k-1}$
    $\varphi(\alpha) = f(x_k + \alpha d_k)$
    $\alpha_k = \min_{\alpha} \varphi'(\alpha)$
    $x_{k+1} = x_k + \alpha_k d_k$
  return x_{k+1}

We used Fletcher-Reeves method to calculate the value for $\boldsymbol{\beta_k}$.

3.1 Exact search along the direction d_k

$\boldsymbol{d_k}$ is our descent direction and the exact search along this direction is given as, $\varphi'(\alpha) = 0$.

$$\varphi'(\alpha) = \nabla f(\boldsymbol{x_{k+1}})^T \boldsymbol{d_k}$$

We can also prove that the gradient and the descent direction are of the opposite direction.

$$\begin{aligned}
&\nabla f(x_k)^T d_k \leq 0 \\
&\nabla f(x_k)^T (-\nabla f(x_k) + \beta_k d_{k-1}) \leq 0 \\
&-\nabla f(x_k)^T \nabla f(x_k) + \beta_k \nabla f(x_k)^T d_{k-1} \leq 0 \\
&-\|\nabla f(x_k)\|_2^2 + \beta_k \nabla f(x_k)^T d_{k-1} \leq 0
\end{aligned}$$

$\nabla f(x_k)$ and d_{k-1} are orthogonal. So, this value will always be less than 0.

As we have proved in the section 2.5 we can obtain the value α along the descent direction d that minimizes the function $\varphi(\alpha) = f(x + \alpha x)$ is given by

$$\frac{\alpha^2 a + \alpha b + c}{R(\alpha)} = \varphi'(\alpha) = 0$$

Where,

$$a = (d^T M x) d^T d \tag{3.1}$$

$$b = (x^T M x) d^T d - (d^T M d)(x^T x)$$

$$c = (x^T M x) x^T d - (d^T M x) x^T x$$

$$R(\alpha) = (x(\alpha)^T x(\alpha))^2$$

One important thing to notice is that $R(\alpha) \neq 0$

$$\begin{aligned} R(\alpha) &\neq 0 \\ ((x + \alpha d)^T (x + \alpha d)) &\neq 0 \\ \|x + \alpha d\|_2^2 &\neq 0 \\ x^T x + \alpha^2 d^T d + 2\alpha < x, d > &\neq 0 \end{aligned}$$

Note : In CGD instead of d we use p. So, $R(\alpha) = \|x_k + \alpha_k p_k\|_2^2$
For $R(\alpha)$ to be 0 all the three equation $x^T x$, $\alpha^2 d^T d$ and $2\alpha < x, d >$ has to be 0 or if $x^T x$, $\alpha^2 d^T d$ are positive, $2\alpha < x, d >$ should not be negative. But, $x^T x = \|x_k\|_2^2$ and $\|p_k\|_2^2$ are always greater or equal to 0. So, If we can show that $2\alpha < x, d >$ is always greater than 0, $R(\alpha)$ is always different from 0. Using Cauchy–Schwarz inequality we can show that

$$| < x_k, p_k > | < \|x_k\|_2^2 \|p_k\|_2^2$$

Since the sign is less than, that mean the cosine angle between the two is not 1(or -1) hence, not linearly dependant.
So now we showed that $R(\alpha) \neq 0$, we now want to find a solution for $\phi'(\alpha) = 0$

$$a\alpha^2 + b\alpha + c = 0 \tag{3.2}$$

3.2 Convergence of the Algorithm

In this section we want to prove that given $\|x_o\|_2 \epsilon D, D = \mathbb{R}^n \backslash B(0, \varepsilon)$, we want to show $x_k \epsilon D, \forall k \geq 0$. We started with the assumption that $\|x_o\|_2^2 \geq \epsilon \geq 0$, and now lets compute for $\|x_1\|_2$.

$$\|x_1\|_2^2 = \|x_0 + \alpha_0 p_0\|_2^2 \geq 0 = \|x_0\|_2^2 + 2\alpha_0 < x_0, p_0 > + \alpha_0^2 \|p_0\|_2^2 \geq 0$$

But this term $2\alpha_0 < x_0, p_0 >$ is 0 since $p_0 = -\nabla f(x_0)$ and $\|x_0\|_2^2 \geq 0$. So its true that $\|x_1\|_2 \geq 0$
We can do the same for $\|x_2\|_2$. If we generalize for $\|x_k\|_2^2$, we would get.

$$\|x_k\|_2 = \|x_{k-1} + \alpha_{k-1} p_{k-1}\|_2^2 \geq 0 = \|x_{k-1}\|_2^2 + 2\alpha_{k-1} < x_{k-1}, p_{k-1} > + \alpha_{k-1}^2 \|p_{k-1}\|_2^2 \geq 0$$

$$< x_k, p_k >= \sum_{i=0}^{k-1} (\prod_{j=k+1-i}^{k+1} b_j) \hat{\alpha_i} \|p_i\|_2^2 \geq 0, \forall i \tag{3.3}$$

Hence, $\forall k, \|x_k\|_2^2 > 0 \Rightarrow \exists \epsilon$ s.t $\|x_k\|_2^2 \geq \epsilon$. That means $\|x_k\|_2^2$ bounded from below. Moreover, using triangle inequality, hölder's inequality and Cauchy–Schwarz inequality it is possible to show that the gradient is bounded by some constant.

$$\|\nabla f(x)\| \leq 4\|M\|^2, \|x\| = 1 \tag{3.4}$$

Proposition 1 *If a function f has bounded derivative, then f is lipschitz continuous.*

Assuming we have α that satisfies Wolfe's condition and using the proof provided by Zoutendijk's theorem with Fletcher–Reeves and Polak–Ribiere methods we can guarantee that our algorithm will converge to a stationary point given that our function is bounded from below in R^n(2.15), continuously differentiable (2.4) and lipschitz continuous(Proposition 1). The complete proof is provided at [1, pg. 127-131].

3.3 Complexity of the Algorithm

For CGD we have computation for β_k and P_k which we can obtain both with $\mathcal{O}(n)$ complexity. We have proved the complexity for $f(x), \nabla f(x)$ and $\phi'(\alpha)$ in the previous sections. So we can say that the complexity of one iteration of the conjugate gradient method is $\mathcal{O}(n^2)$

$$C(CGD) = \mathcal{O}(n^2) \tag{3.5}$$

3.4 Performance of Algorithm

In this section we will provide performance analysis of our algorithm implementation. For this purpose we have generated the following matrices with different properties. x_0 (b_0 in case of arnoldi) is the initial guess for the specified matrix. We ran all our experiments on Apple Macbook Air(Early 2014) 1.4 GHz Dual-Core Intel Core i5 with 4 GB 1600 MHz DDR3 memory.

- **[M1]** is ill-conditioned matrix $A_{1000\times1000}$, where a ∈ [-2,3]and cond. no = 2e15, $x_0 \in \mathbb{R}^{1000}$.
- **[M2]** $A_{10000\times100}$, where a ∈ [-50,50], $x_0 \in \mathbb{R}^{100}$ vector.
- **[M3]** $A_{100\times1000}$, where a ∈ [-50,50], $x_0 \in \mathbb{R}^{1000}$ vector.
- **[M4]** $A_{100\times100}$, where a ∈ [-50,50], $x_0 \in \mathbb{R}^{100}$ vector.
- **[M5]** is sparse matrix $A_{1000\times100}$ and density = 0.3, $x_0 \in \mathbb{R}^{100}$.

For this experiment we randomly generated our initial guess with a range of [-50,50]. We set the value for maximum iteration to be 1000 and 1e-5 for epsilon. For minimum step size α we chose 1e-8. Residual is defined as the norm of the gradient and relative error is defined as the ratio between absolute error and accepted norm. They are defined as follow.

$$
\begin{aligned}
Residual &= \|\nabla f(x^i)\| \\
RelativeError &= (f(x^i) - f^*)/|f^*|
\end{aligned}
\tag{3.6}
$$

Figure 1,2,3,4 and 5 shows relative error and residual generated by the algorithm with different methods on our test matrices.

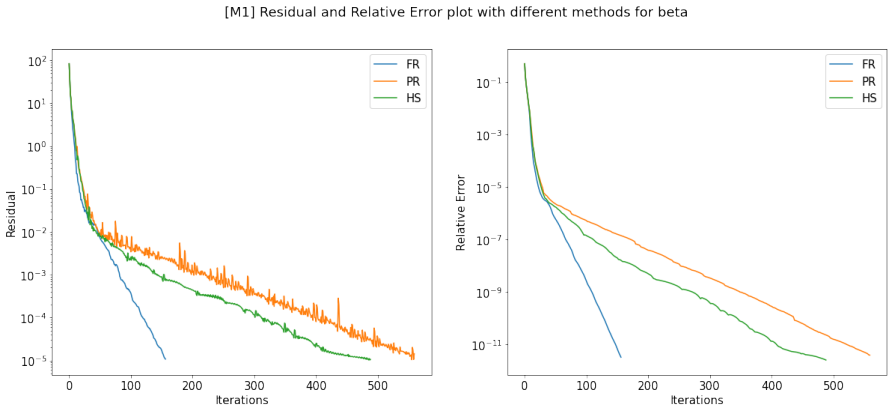


Figure 1: Relative error and residual plot for M1 matrix

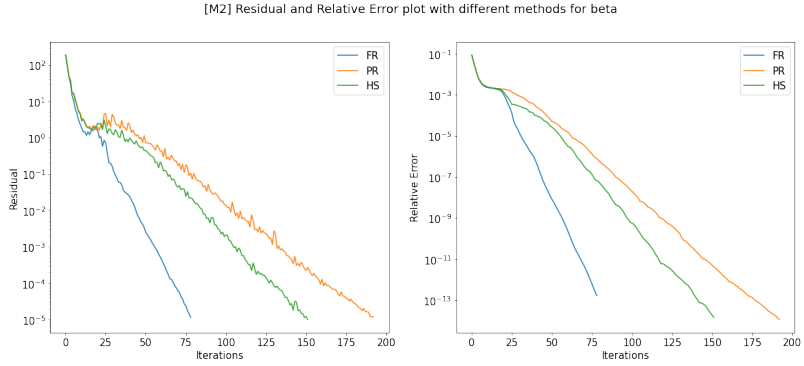


Figure 2: Relative error and residual plot for M2 matrix

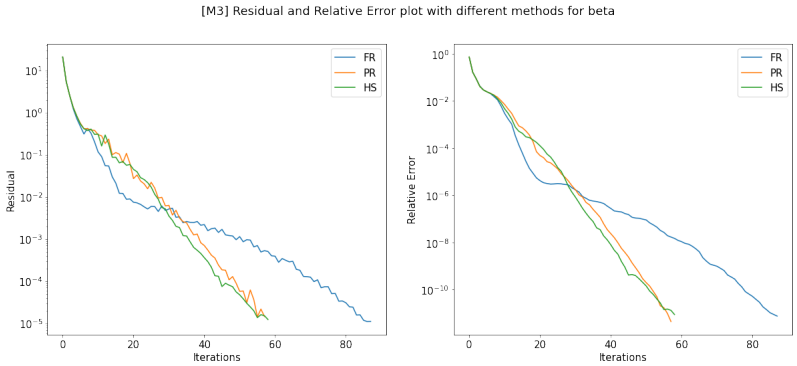


Figure 3: Relative error and residual plot for M3 matrix

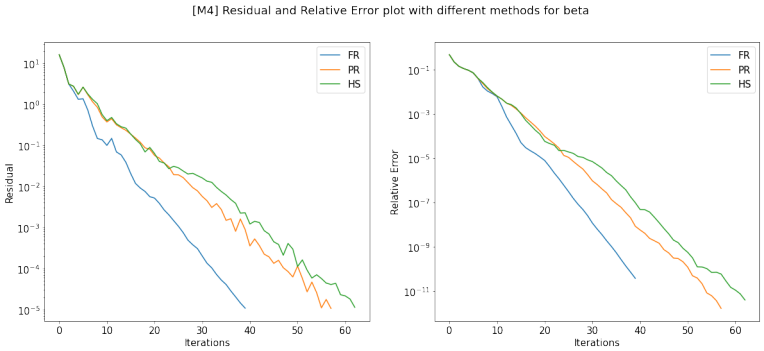


Figure 4: Relative error and residual plot for M4 matrix"

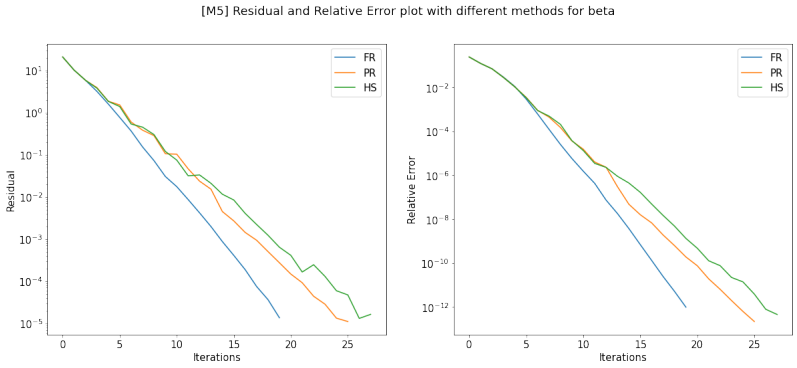


Figure 5: Relative error and residual plot for M5 matrix

Looking at all Fig. 1, Fig. 2 and Fig. 3 we can understand that the algorithm converges faster and doesn't show significant change in both error and residual. So, it would be possible to do an early stopping even before $1e{-}5$ without affecting the final result significantly. We can also observe that our algorithm converges slower on ill conditioned matrices.

3.4.1 Algorithm time performance

We evaluated the running time performance of our algorithm and compared the accuracy of the result with numpy linear algebra implementations of a method norm [1]. The following table summarises the running time of our implementation compared to the numpy tool. The error column represents the absolute difference between the solution provided by our algorithm and the numpy norm library.

Matrix	CGD(Time)	np.linalg.norm(Time)	Relative Error	Iterations
M1	834 ms ± 146 ms	537 ms ± 23.7 ms	2.87e-12	157
M2	23.9 ms ± 1.53 ms	79.3 ms ± 2.24 ms	1.14e-13	79
M3	336 ms ± 22.4 ms	12.7 ms ± 2.23 ms	5.24e-12	88
M4	14.1 ms ± 491 µs	1.83 ms ± 401 µs	2.01e-11	40
M5	7.57 ms ± 1.46 ms	5.74 ms ± 691µs	2.57e-13	20

Table 1: Time performance, epsilon = 1e-5 and beta = "FR"

Matrix	CGD(Time)	np.linalg.norm(Time)	Relative Error	Iterations
M1	2.41 s ± 472ms	585 ms ± 42.3 ms	3.78e-12	560
M2	86.2 ms ± 35.1 ms	81.8 ms ± 1.98 ms	0.0069	193
M3	226 ms ± 14 ms	11.5 ms ± 2.17ms	2.58e-12	89
M4	17.2 ms ± 1.2 ms	1.41 ms ± 60.9µs	1.17e-12	58
M5	7.75 ms ± 295 µs	8.78 ms ± 2.31 ms	6.33e-14	26

Table 2: Time performance, epsilon = 1e-5 and beta = "PR"

Matrix	CGD(Time)	np.linalg.norm(Time)	Relative Error	Iterations
M1	3.11 s ± 311ms	585 ms ± 42.3 ms	2.44e-12	490
M2	54.9 ms ± 13.8 ms	81.8 ms ± 1.98 ms	1.23e-14	153
M3	285 ms ± 25.4 ms	11.5 ms ± 2.17ms	6.048e-12	60
M4	26.1 ms ± 1.63 ms	1.41 ms ± 60.9µs	2.20e-12	64
M5	10.1 ms ± 295 µs	8.78 ms ± 2.31 ms	1.2e-13	29

Table 3: Time performance, epsilon = 1e-5 and beta = "HS"

Comparing the running time and the number of iterations, Fletcher–Reeves method is faster compared to the other methods and converges in a fewer iterations. The performance of **HS** method is comparatively similar to that of **PR** method. Interestingly, the same can be observed from the relative error and residual plot. Another observation we could take is that our algorithm takes longer time to converge on matrix type **M1**(ill-conditioned) and comparatively faster time on other type of matrices. The high relative error occurred on **M2** is caused by a convergence to a local minimum. The test has been carried out by providing different starting point and we were able to get significantly low relative error.

[1] https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

Since the performance our algorithm is dependant on the initial guess of vector $\boldsymbol{x_0}$, it would be difficult to get the exact performance. To avoid this problem we compared our implementation with a Conjugate Gradient method implementation of scipy[2] providing the same initial vector.

Matrix	Scipy CG(Time)	Relative Error	Iterations
M1	455 ms ± 29.3 ms	3.44e-13	65
M2	33.4 ms ± 1.72 ms	3.11e-15	89
M3	317 ms ± 15.7 ms	4.93e-12	36
M4	15.6 ms ± 2.27 ms	1.09e-13	32
M5	7.28 ms ± 1.48 ms	2.76e-14	23

Table 4: Time performance of Scipy CG

Table 4 shows the summary of running time, error and number of iterations for each matrices. Interestingly our Fletcher-Reeves implementation of Conjugate Gradient was able to match the performance of scipy implementation except for $\boldsymbol{M1}$(which is ill-conditioned) matrix which took longer iterations to converge.

4 Quasi-Newton method: BFGS

The following section describes our use of one of the most popular quasi-Newton methods, the BGFS method, to solve the optimization problem defined in Section 1.
Quasi-Newton methods, unlike proper Newton methods, don't require the computation of the Hessian and thus, while still providing a lower convergence speed, are known to be a cheaper alternative to the latter.

We start by introducing the BFGS algorithm, named after Broyden, Fletcher, Goldfarb, and Shanno, together with its proprieties.

Algorithm 2 BFGS

procedure BFGS($\boldsymbol{H}_0 \in \mathbb{R}^{n\times n}, \boldsymbol{x}_0 \in \mathbb{R}^n, \epsilon \in \mathbb{R}$)
  while $\|\nabla f_k\| > \epsilon$ **do**
    $p_k = -H_k \nabla f_k$ ▷ Computes the search direction
    $x_{k+1} = x_k + \alpha_k p_k$ ▷ Where α_k satisfies either the Wolfe or the Armijo conditions
    $s_k = x_{k+1} - x_k, \quad y_k = \nabla f_{k+1} - \nabla f_k$
    $\rho_k = \frac{1}{y_k^T s_k}$
    $H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$
    $k = k + 1$

The $\boldsymbol{H}_k$ matrix is the inverse of the $\boldsymbol{B}_k$ matrix, $\boldsymbol{H}_k = \boldsymbol{B}_k^{-1}$, an $n \times n$ symmetric positive definite matrix, acting as an approximate Hessian, that is updated at every iteration.

[2]https://docs.scipy.org/doc/scipy/reference/optimize.minimize-cg.html#optimize-minimize-cg

Instead of recomputing the matrix from scratch at every iteration, we want to update it by applying simple modifications accounting for the curvature measured during the most recent step. For such incremental refinement to work, some requirements are imposed on the computation of $\boldsymbol{H}_{k+1}$, these are: the satisfaction of the *secant equation* and, in order to uniquely identify a single matrix as the new approximation, the satisfaction of a *condition of closeness* to the previous iterate $\boldsymbol{H}_k$.

The secant equation:

$$\boldsymbol{H}_{k+1}\boldsymbol{y}_k = \boldsymbol{s}_k$$

requires the matrix $\boldsymbol{H}_{k+1}$ to map the displacement $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \alpha_k \boldsymbol{p}_k$ into the difference between the gradients $\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k$. This is possible only for $\boldsymbol{s}_k$ and $\boldsymbol{y}_k$ satisfying the *curvature condition*

$$\boldsymbol{s}_k^T \boldsymbol{y}_k > 0.$$

The fulfillment of this condition, which ensures the inheritance by $\boldsymbol{H}_{k+1}$ of the positive definiteness of $\boldsymbol{H}_k$, is not always guaranteed for non-convex functions like our objective function.

In these cases, the curvature condition needs to be enforced explicitly, by imposing on the line search procedure that chooses the step length α the satisfaction of the Wolfe conditions:

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f_k^T \boldsymbol{p}_k,$$

$$\nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla f_k^T \boldsymbol{p}_k,$$

or the Armijo conditions:

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f_k^T \boldsymbol{p}_k,$$

in both cases with $0 < c_1 < c_2 < 1$.

To determine $\boldsymbol{H}_{k+1}$ uniquely, we search, among all symmetric matrices satisfying the secant equation, the closest matrix to the current $\boldsymbol{H}_k$. The condition of closeness is specified by:

$$\min_{\boldsymbol{H}_{k+1}} \|\boldsymbol{H}_{k+1} - \boldsymbol{H}_k\|$$

$$\text{subject to } \boldsymbol{H} = \boldsymbol{H}^T \text{ and } \boldsymbol{H}\boldsymbol{y}_k = \boldsymbol{s}_k.$$

Different matrix norms can be used, each one giving rise to a different quasi-Newton method, the one utilized in the BFGS method is the weighted Frobenius norm:

$$\|\boldsymbol{A}\|_W = \|\boldsymbol{W}^{1/2} \boldsymbol{A} \boldsymbol{W}^{1/2}\|_F,$$

where the weight matrix $\boldsymbol{W}$ is any matrix satisfying $\boldsymbol{W}\boldsymbol{s}_k = \boldsymbol{y}_k$.

The unique solution $\boldsymbol{H}_{k+1}$ to the previous optimization problem is given by

$$\boldsymbol{H}_{k+1} = (\boldsymbol{I} - \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^T)\boldsymbol{H}_k(\boldsymbol{I} - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^T) + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^T, \tag{4.1}$$

with

$$\rho_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}.$$

Regarding the initialization of the $\boldsymbol{H}_0$ matrix, there is not a fixed heuristic solving the issue. It could be either be defined as the inverse approximate Hessian calculated by finite differences at $\boldsymbol{x}_0$, the identity matrix $\boldsymbol{I}$, or even one of its multiples $\beta\boldsymbol{I}$.

4.0.1 Cautious BFGS

We also introduce a variant of the BFGS algorithm based on a more cautious approach to the update of the $\boldsymbol{B}_k$ matrix, or in our case, the $\boldsymbol{H}_k$ one. The following cautious update rule was designed by Li and Fukushima in [2]:

$$\boldsymbol{B}_{k+1} = \begin{cases} \boldsymbol{B}_k - \frac{\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k}{\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k} + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}, & \text{if } \frac{\boldsymbol{y}_k^T \boldsymbol{s}_k}{\|\boldsymbol{s}_k\|^2} > \epsilon \|\nabla f_k\|^\alpha, \\ \boldsymbol{B}_k, & \text{otherwise,} \end{cases}$$

with both ϵ and α as positive constants. As probably expected, we will modify such rule so to accomodate the use of the $\boldsymbol{H}_k$ matrix instead of $\boldsymbol{B}_k$, such modification results in the following update formula:

$$\boldsymbol{H}_{k+1} = \begin{cases} (\boldsymbol{I} - \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^T)\boldsymbol{H}_k(\boldsymbol{I} - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^T) + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^T, & \text{if } \frac{\boldsymbol{y}_k^T \boldsymbol{s}_k}{\|\boldsymbol{s}_k\|^2} > \epsilon \|\nabla f_k\|^\alpha, \\ \boldsymbol{H}_k, & \text{otherwise,} \end{cases} \tag{4.2}$$

From now on we will refer to the version of the BFGS algorithm exploiting such rule as C-BFGS.

Algorithm 3 C-BFGS

procedure C-BFGS($\boldsymbol{H}_0 \in \mathbb{R}^{n \times n}, \boldsymbol{x}_0 \in \mathbb{R}^n, \epsilon \in \mathbb{R}$)
  while $\|\nabla f_k\| > \epsilon$ **do**
    $p_k = -H_k \nabla f_k$
    $x_{k+1} = x_k + \alpha_k p_k$
    $s_k = x_{k+1} - x_k, \quad y_k = \nabla f_{k+1} - \nabla f_k$
    Computes H_{k+1} according to 4.2
    $k = k + 1$

4.1 Convergence of the Algorithm

While the convergence theory regarding the use of the BFGS method for convex minimization problems has been mostly established, the same is not true for the nonconvex case. Global convergence for nonconvex objective functions is not yet guaranteed for the original version of the BFGS method, even when enforcing either exact[3] or Wolfe[4] line search strategies.

The application of small changes to the original algorithm has provided, however, several modified versions of the BFGS method displaying global convergence behaviour for nonconvex objective functions. Such changes usually revolve around a more cautious approach to the modification of the $\boldsymbol{B}_k$ matrix, so to preserve the positive definiteness of the Hessian approximation, and/or the use of more or less specifically crafted line search strategies[5][6][7]. The base for the C-BFGS algorithm[2] we described in the previous section stems from such family of modified methods.

In order to provide a more detailed context for the assumption above, we now refer to:
Lemma 4.1 (Lemma 2.1 in [2]): If BFGS method with Wolfe-type line search is applied to a continuously differentiable function f that is bounded below, and if there exists a constant $\boldsymbol{M} > 0$ such that the inequality

$$\frac{\|\boldsymbol{y}_k\|^2}{\boldsymbol{y}_k^T \boldsymbol{s}_k} \leq \boldsymbol{M} \tag{4.3}$$

holds for all k, then

$$\liminf_{k\to\infty} \|g(\boldsymbol{x}_k)\| = 0. \tag{4.4}$$

According to said lemma, the global convergence assumption comes easily for the application of BFGS to a convex function due to how a twice continuously differentiable and uniformly convex function would always satisfies the 4.3 inequality. For the application to a nonconvex function f however, the satisfaction of the same inequality may be hard to guarantee.

While such obstacle is impossible for the original version of BFGS to overcome, its variant equipped with a more cautious update rule for matrix $\boldsymbol{B}_k$, or in our case $\boldsymbol{H}_k$, guarantees an approximate Hessian that is symmetric and positive definite for all values of k. This characterizes $\{f(\boldsymbol{x}_k)\}$ as a decreasing sequence whenever employing either Wolfe- or Armijo-type line searches.

We now refer to the main assumption at the base of the convergence of C-BFGS:
Assumption 4.1 (Assumption A in [2]): The level set

$$\Omega = \{\boldsymbol{x} \in \mathbb{R}^n | f(\boldsymbol{x}) \leq f(\boldsymbol{x}_0)\}$$

is contained in a bounded convex set D. The function f is continuously differentiable on D and there exists a constant $L > 0$ such that

$$\|g(\boldsymbol{x}) - g(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|, \qquad \forall \boldsymbol{x}, \boldsymbol{y} \in D.$$

Having already delved on the nature of $\{f(\boldsymbol{x}_k)\}$ as a decreasing sequence it appears clear how $\{\boldsymbol{x}_k\}$ is guaranteed to be contained Ω. Being not our goal to provide a fully detailed proof of convergence, however, we conclude this section by reporting the two main theorems from the source.

Theorem 4.1 (Theorem 3.2 in [2]): Let Assumption 4.1 holds and $\boldsymbol{x}_k$ be generated by Algorithm 3 with α_k being generated by Armijo-type line search. Then

$$\liminf_{k\to\infty} \|g(\boldsymbol{x}_k)\| = 0. \tag{4.5}$$

Theorem 4.2 (Theorem 3.3 in [2]): Let Assumption 4.1 holds and $\boldsymbol{x}_k$ be generated by Algorithm 3 with α_k being generated by Wolfe-type line search. Then (4.5) holds.

4.2 Complexity of the Algorithm

We now compute the complexity of running an iteration of the BFGS algorithm without accounting for the cost of calculating the value of the function and its gradient.

By directly computing, and updating, the matrix $\boldsymbol{H}_k$ as the inverse of the approximated Hessian, instead of computing the actual approximation $\boldsymbol{B}_K$ and then inverting it, the search direction $\boldsymbol{p}_k = -\boldsymbol{H}_k \nabla f_k$ can be calculated by means of a simple matrix–vector multiplication. This entails just $O(n^2)$ operations and thus saves us from the computational burden, of $O(n^3)$ more operations, of performing the inversion.

The algorithm is therefore free from costly matrix-matrix operations and most of its internal processes are managed through matrix-vector, vector-vector or scalar operations.

The overall computational complexity of running an iteration of the BFGS algorithm, just as for C-BFGS, is:

$$C(\text{BFGS}) = C(\text{C-BFGS}) = O(n^2)$$

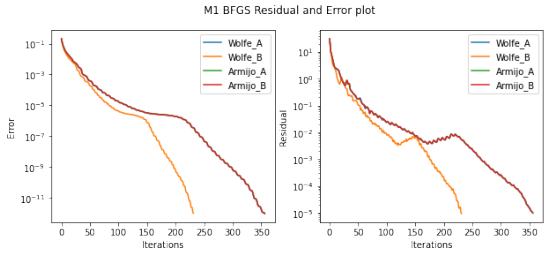
4.3 Performance of the Algorithm

In this section we provide an analysis of the performance displayed by both the BFGS and C-BFGS algorithms when applied to the experimental setup described in Section 3.4.

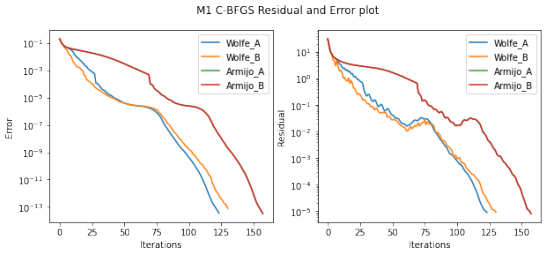
The identity matrix $\boldsymbol{I}$ was feed to the algorithms as initial guess for the inverse of the approximated Hessian matrix $\boldsymbol{H}_0$. The comparison also accounts for the application of either Wolfe- or Armijo-type line searches for the computation of the step-length. The labels $Wolfe\_A$, $Wolfe\_B$, $Armijo\_A$ and $Armijo\_B$ refer to the set of values utilised for the c_1 and c_2 constants, respectively: $(1e-4, 0.9), (0.1, 0.49), (1e-4, //)$ and $(0.1, //)$.

4.3.1 Error and residual plots

The following plots display the relative error and residual generated by both algorithms.



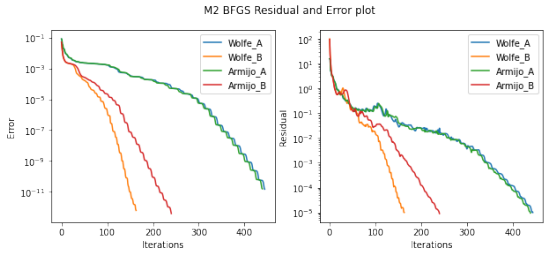
(a) BFGS



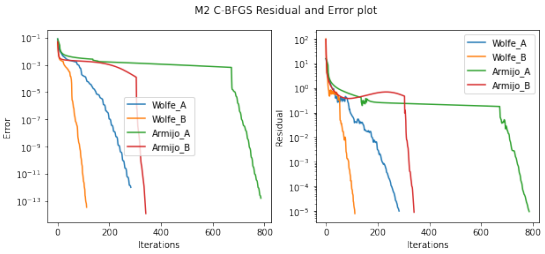
(b) C-BFGS

Figure 6: Relative error and residual plot for M1 matrix

In the case of the ill-conditioned matrix $\boldsymbol{M1}$, whose plots are displayed in Figure 6, the cautious version of the BFGS algorithm seems to greatly outperform the original one in both the number of iterations need for convergence and the ultimate magnitude of the error. Again, the original BFGS appears to be *mostly* invariant with respect to the type of line-search it employs, said invariance stands mostly evident for C-BFGS with respect to the employment of Armijo-type line search.



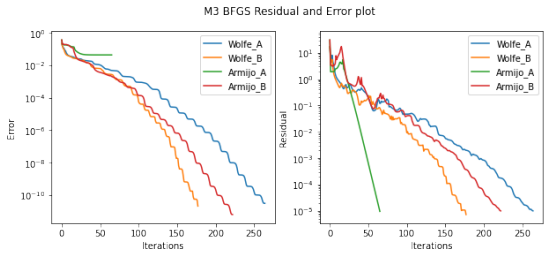
(a) BFGS



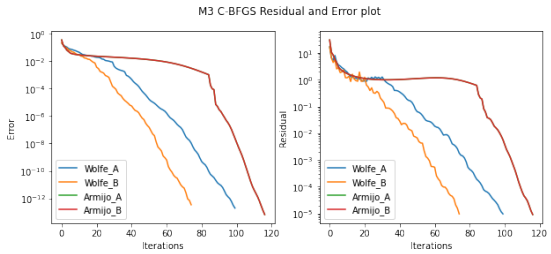
(b) C-BFGS

Figure 7: Relative error and residual plot for M2 matrix

Figure 7 displays the plots for the **M2** matrix, While C-BFGS seems to outperform the original BFGS when emploing a Wolfe-type line search, the opposite of this phenomenon is observed when emploing an Armijo-type one.

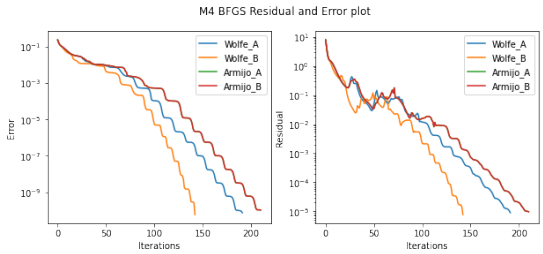


(a) BFGS

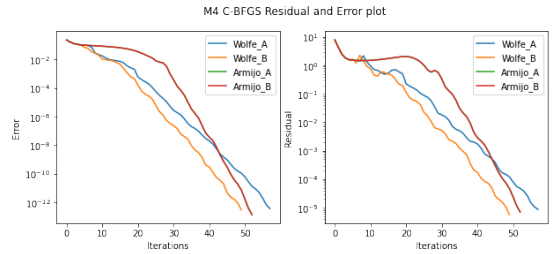


(b) C-BFGS

Figure 8: Relative error and residual plot for M3 matrix

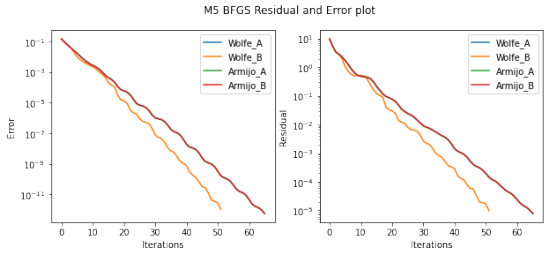


(a) BFGS



(b) C-BFGS

Figure 9: Relative error and residual plot for M4 matrix



(a) BFGS

(b) C-BFGS

Figure 10: Relative error and residual plot for M5 matrix

8, 9 and 10 confirm the better performance of C-BFGS regarding its application to the three randomly generated matrices. A relevant trend is also the one identifying Wolfe-type line search, especially the one employing the set of values for the c_1 and c_2 constants suggested by [2], as the most efficient among its peers.

One of the runs shown on Figure 8a, specifically the one using an Armijo-type line search with $c_1 = 0.0001$ in the search for **M3**'s norm, has shown to stop at a point of unusually high relative error. After performing additional tests with different starting points generated randomly and having

actually verified their convergence to lower-error points, we established that the first outcome had to be dictated by an unlucky convergence at a local minimum.

4.3.2 Algorithms time and accuracy performance

We now present the results of our comparison of the average running time and accuracy performance of both our BFGS and C-BFGS algorithms with respect to the Scipy's implementation of the former. Numpy's implementation of the matrix-norm function has been used to compute the error as the absolute difference between the value provided by said function and the solution provided by our algorithms, or the implementation from Scipy.

The set of values utilized for the positive constants c_1 and c_2 are presented in the respective columns. Scipy's implementation of BFGS is based on an application of the Wolfe-type line search with default values for $c_1 = 1e-4$ and $c_2 = 0.9$.

Matrix	Line-search	c_1	c_2	Average time	Relative Error	Iterations
M1	Wolfe	0.0001	0.9	50.2 s ± 276 ms	1.01e-12	357
M1	Wolfe	0.1	0.49	32.1 s ± 283 ms	1.03e-12	232
M1	Armijo	0.0001	//	47.8 s ± 250 ms	9.87e-13	357
M1	Armijo	0.1	//	47.6 s ± 1.5 s	9.87e-13	357
M2	Wolfe	0.0001	0.9	243 ms ± 11.3 ms	1.46e-11	445
M2	Wolfe	0.1	0.49	109 ms ± 18.3 ms	6.11e-13	169
M2	Armijo	0.0001	//	232 ms ± 19.1 ms	1.55e-11	440
M2	Armijo	0.1	//	128 ms ± 12.7 ms	3.86e-13	241
M3	Wolfe	0.0001	0.9	38.2 s ± 2.67 s	3.08e-11	265
M3	Wolfe	0.1	0.49	20.9 s ± 183 ms	2.09e-11	172
M3	Armijo	0.0001	//	7.75 s ± 113 ms	0.04	66
M3	Armijo	0.1	//	27.3 s ± 2.52 s	6.32e-12	223
M4	Wolfe	0.0001	0.9	112 ms ± 17.3 ms	7.39e-11	192
M4	Wolfe	0.1	0.49	91.6 ms ± 4.62 ms	5.89e-11	143
M4	Armijo	0.0001	//	109 ms ± 17 ms	1.05e-10	211
M4	Armijo	0.1	//	118 ms ± 17.5 ms	1.05e-10	211
M5	Wolfe	0.0001	0.9	37.4 ms ± 2 ms	5.73e-13	66
M5	Wolfe	0.1	0.49	34.3 ms ± 3.48 ms	1.09e-12	52
M5	Armijo	0.0001	//	34.1 ms ± 1.91 ms	5.73e-13	66
M5	Armijo	0.1	//	35.2 ms ± 3.07 ms	5.73e-13	66

Table 5: Time and accuracy performance for our implementation of the BFGS algorithm

Matrix	Average time	Relative Error	Iterations
M1	49.5 s ± 507 ms	9.79e-13	422
M2	46.7 ms ± 11 ms	0.0002	78
M3	34.4 s ± 489 ms	4.44e-11	294
M4	33 ms ± 1.81 ms	0.04	55
M5	33.1 ms ± 1.86 ms	7.26e-14	57

Table 6: Time and accuracy performance for the Scipy's implementation of the BFGS algorithm

Matrix	Line-search	c_1	c_2	Average time	Relative Error	Iterations
M1	Wolfe	0.0001	0.9	12.1 s ± 148 ms	3.53e-14	124
M1	Wolfe	0.1	0.49	15.5 s ± 157 ms	7.65e-14	131
M1	Armijo	0.0001	//	12.9 s ± 140 ms	3.21e-14	158
M1	Armijo	0.1	//	12.7 s ± 147 ms	3.21e-14	158
M2	Wolfe	0.0001	0.9	149 ms ± 14.6 ms	9.70e-13	284
M2	Wolfe	0.1	0.49	64.2 ms ± 1.74 ms	3.34e-14	113
M2	Armijo	0.0001	//	198 ms ± 10.4 ms	1.55e-13	787
M2	Armijo	0.1	//	81.2 ms ± 2.29 ms	1.15e-14	342
M3	Wolfe	0.0001	0.9	9.32 s ± 205 ms	1.94e-13	100
M3	Wolfe	0.1	0.49	7.84 s ± 93.2 ms	3.40e-13	75
M3	Armijo	0.0001	//	4.83 s ± 60.9 ms	6.59e-14	117
M3	Armijo	0.1	//	4.76 s ± 75.6 ms	6.59e-14	117
M4	Wolfe	0.0001	0.9	32.7 ms ± 1.68 ms	3.62e-13	58
M4	Wolfe	0.1	0.49	31.3 ms ± 3.1 ms	2.97e-13	50
M4	Armijo	0.0001	//	21.5 ms ± 1.25 ms	1.41e-13	53
M4	Armijo	0.1	//	22.2 ms ± 1.79 ms	1.41e-13	53
M5	Wolfe	0.0001	0.9	33.7 ms ± 8.77 ms	3.63e-13	49
M5	Wolfe	0.1	0.49	30.3 ms ± 2.67 ms	4.50e-13	40
M5	Armijo	0.0001	//	35.2 ms ± 14.5 ms	2.25e-13	55
M5	Armijo	0.1	//	29.4 ms ± 1.66 ms	2.25e-13	55

Table 7: Time and accuracy performance for our implementation of the C-BFGS algorithm

5 Largest Eigenvalue Estimation via Arnoldi Iteration

The following section describes our use of the Gram-Schmidt-style Arnoldi iteration to reduce a matrix $\boldsymbol{A}^T\boldsymbol{A}$ to Hessenberg form and approximate its largest eigenvalue. The relationship between the norm of a matrix $\boldsymbol{A}$ and the spectral radius of its scalar product matrix $\boldsymbol{M} = \boldsymbol{A}^T\boldsymbol{A}$ is exploited in order to compute the former.

We start by introducing the Arnoldi Iteration algorithm with its proprieties.

Algorithm 4 Arnoldi Iteration

procedure ARNOLDI($\boldsymbol{M} \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^n, k \in \mathbb{N}$)
  H = matrix of zeros $\in \mathbb{R}^{m\times k+1}$
  $q_1 = b/\|b\|$
  for $i = 1, ..., k$ **do**
    $v = Mq_i$
    for $j = 1, ..., i$ **do**
      $H_{ji} = q_j^* v$
      $v = v - H_{ji}q_j$
    $H_{i+1,i} = \|v\|$
    $q_{i+1} = v/H_{i+1,i}$
  return H

The matrix $\boldsymbol{H}$ is an upper Hessenberg matrix while the $\boldsymbol{Q} = [q_1, ..., q_m]$ matrix contains the basis vectors of the Krylov subspace, both are incrementally constructed by the algorithm. The initial vector $\boldsymbol{b}$, soon normalized, is instead randomly built as is usual when applying the Arnoldi Iteration to the problem of estimating eigenvalues.
At each new iteration, the algorithm construct a new basis vector $\boldsymbol{q}_{k+1}$, orthogonal to its predecessors, by using a modified Gram-Schmidt process.

By assuming $k < m$, we can consider $\boldsymbol{H}_k$, the upper Hessenberg matrix generated via k iterations of the Arnoldi algorithm, as a low-rank approximation of matrix $\boldsymbol{M} = \boldsymbol{A}^T\boldsymbol{A}$.
Its eigenvalues, usually called Ritz Values, are known to converge to the largest eigenvalues of $\boldsymbol{M}$, as we will soon prove.
By using the Arnoldi Iteration for the construction of increasingly more complex approximations of $\boldsymbol{A}^T\boldsymbol{A}$, we can identify an exact, if not nearly so, value for the largest eigenvalue of said matrix and therefore compute the norm of the original matrix $\boldsymbol{A}$ according to the spectral radius property analyzed in (2.1).
What follows is our propose for the algorithm performing the aforementioned norm estimation.

Algorithm 5 Norm Estimation based on the Arnoldi Iteration

procedure ARNOLDI-NORM($\boldsymbol{M} \in \mathbb{R}^{m\times n}$)
  b = arbitrary vector
  $k = 1$
  $H_1 = \textbf{ARNOLDI}(M, b, 1)$
  $\theta_1 = \max(\text{eig}(H_1))$
  repeat
    $k = k + 1$
    $H_k = \textbf{ARNOLDI}(M, b, k)$
    $\lambda_k = \max(\text{eig}(H_k))$ ▷ Largest Ritz-Value Estimation
  until $\|\lambda_k - \lambda_{k-1}\| < \epsilon$ ▷ Check for Convergence
  return $\sqrt{\lambda_k}$ ▷ The norm of A is computed from λ_k according to (2.1)

5.1 Convergence of the Algorithm

The following section describes the Convergence of the Arnoldi method for our Eigenvalue Estimation . We saw the Convergence of the Arnoldi method in two cases. In the first case If there is a lucky breaks down at step $\boldsymbol{n}$. Recall that given a Matrix $\boldsymbol{M}$ the Arnoldi method produce

$$MQ_n = Q_{n+1}H_n \tag{5.1}$$

we can rewrite equation 5.1 as

$$MQ_n = Q_nH_n + q_{n+1}\beta_{n+1,n}\epsilon_n^T \tag{5.2}$$

H_n is without the last row and ϵ_n^T is a vector with all element 0 expect the last entity at n is 1.
We said luck break down happen when $\beta_{n+1,n} = 0$ and then equation 5.2 reduce to

$$MQ_n = Q_nH_n \tag{5.3}$$

in such a case when luck break down happen the eigenvalue value of H_n is a subset of the eigenvalue M.
proof as follows:
For every eigenvalue eigenvector pair $H_nV = \lambda V$, $Mx = \lambda x$, $x = Q_nV$ i.e (λ,x) eigenvalue eigenvector pair M.

$$MQ_n = Q_nH_n \tag{5.4}$$

$$MQ_nV = Q_nH_nV = Q_n\lambda V \tag{5.5}$$

$$MQ_nV = \lambda Q_nV \tag{5.6}$$

where (Q_nV, λ) are eigen pair of M which implies the eigenvalue of $\boldsymbol{H}$ is subset of the eigenvalue of M.

The second case is when there is no lucky break down at $\boldsymbol{n}$ steps, eigenvalue and eigenvector of H is a good approximation of our matrix M and we can check this by computing the difference of eigenvector and eigenvalue of M and taking the norm should equal to 0 as such $\|Mx - \lambda x\| = 0$.

5.2 Complexity of the Arnoldi Iteration

We now compute the complexity of running k steps of the Arnoldi Iteration when applied to a matrix $\boldsymbol{P} \in \mathbb{R}^{m\times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^m$.
The Arnoldi Iteration entails k products with the $\boldsymbol{M}$ matrix. The cost of these products, assuming the sparsity of matrix $\boldsymbol{M}$, is $O(k \cdot nnz(\boldsymbol{M}))$, where $nnz(\boldsymbol{M})$ represents the total number of non-zero elements in $\boldsymbol{M}$.

The scalar product $h_{ji} = q_j^* v$ and linear combination of vectors $v = v - h_{ji}q_j$ are instead performed

$O(k^2)$ times for a total cost of $O(m \cdot k^2)$.

Additionally, the vector-norm operation $h_{i+1,i} = \|v\|$ and the vector scalar division $q_{i+1} = v/h_{i+1,i}$ are performed k times for a total cost of $O(2 \cdot k \cdot 2m) \approx O(k \cdot m)$ operations.

Finally, the total cost of performing k steps of the Arnoldi Iteration, in the case of a sparse $\boldsymbol{A}^T\boldsymbol{A}$ matrix, is:

$$C(ARNOLDI) = O(k \cdot m \cdot nnz(\boldsymbol{M})) + (m \cdot k^2) + O(k \cdot m) = O(m \cdot k^2) \tag{5.7}$$

5.3 Performance of Algorithm

In this section we provide an analysis of the performance of Arnoldi algorithms when applied to the experimental setup described in Section 3.4. In addition we perform $\boldsymbol{A}^T\boldsymbol{A}$ on **M3** ,**M4** and $\boldsymbol{A}\boldsymbol{A}^T$ on M2,M5 to make them symmetric.

5.3.1 Algorithm time performance

We evaluated the running time performance of our algorithm and compared the accuracy of the result with numpy linear algebra implementation of a method norm [3]. The following table summarises the running time of our implementation compared to the numpy tool. The error column represents the absolute difference between the solution provided by our algorithm and the numpy norm library.

Matrix	Arnoldi	np.linalg.norm	Error	Iterations
M1	23.9 s ± 4.26 s	592 ms ± 18.5 ms	2.86e-11	58
M2	1.84 s ± 384 ms	201 ms ± 273 ms	0.21	14
M3	5.83 s ± 208 ms	11.8 ms ± 1.78 ms	1.52e-12	35
M4	636 ms ± 62.9 ms	1.5 ms ± 241 µs	3.6e-11	22
M5	952 ms ± 152 ms	6.2 ms ± 198 µs	3.52e-11	19

Table 8: Arnoldi Time performance

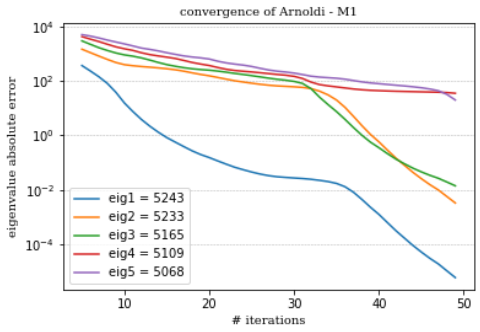
From table 7, we can observe that the error, which is difference between the algorithm result and off the shelf solver solution, indicates that the result provided by the algorithm is very close to the result provided by the off the shelf solver except for **M2** . We can observe that the algorithm failed to reach convergence on $\boldsymbol{M}2$ due to not enough number of iterations. We analyzed by increasing the number of iterations and after taking too much time, the algorithm failed to converge. This could be due to the matrix eigenvalues being not well separated.

[3] https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

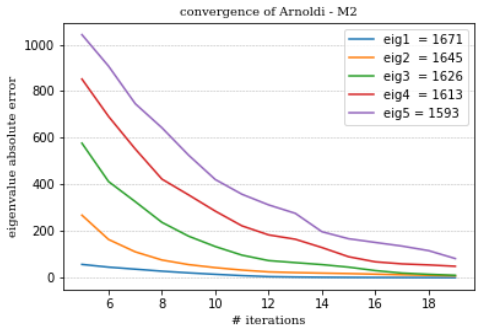
5.3.2 Algorithm convergence rates

We examined the convergent rate of the algorithm by comparing the Ritz values to the exact eigenvalues. We took the first 5 Ritz values of our algorithm and compared them to the corresponding exact eigenvalues calculated with numpy linear algebra implementation of a method eig [4]. The following plots Figure 11 show the convergent rate (as the absolute error $|\lambda^{(n)} - x|$) as a function of the number of iterations [8, pg. 261-265]. The algorithm converges fast for all matrices and we observe that the biggest eigenvalue is converging the fastest.

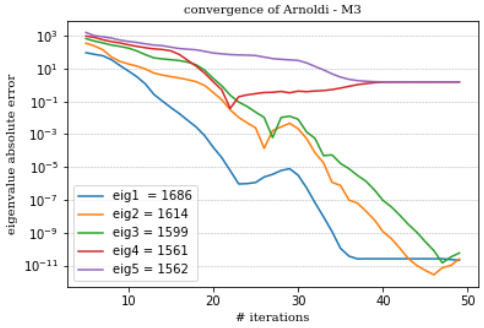
[4] https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html



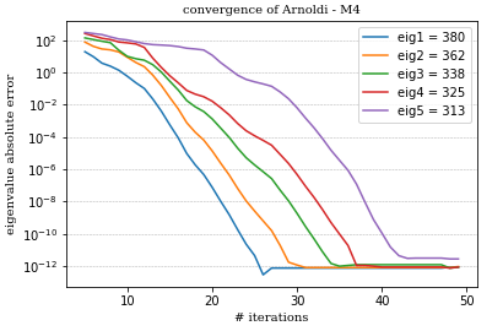
(a) M1



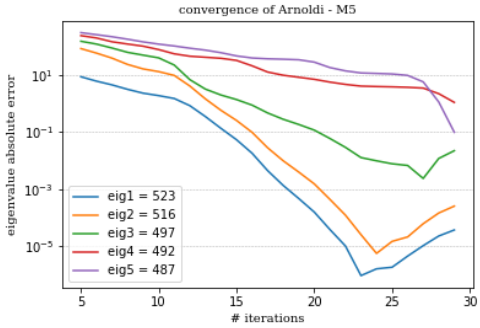
(b) M2



(c) M3



(d) M4



(e) M5

Figure 11: Arnoldi convergence rates

6 Arnoldi future improvement

We observed that Arnold algorithm is the slowest compared to BFGS and CGD. Theoretically the method suffers from slow convergence when the eigenvalues are not well separated. The following section provides suggestions on how to accelerate the convergence of the method. The first method is based on restarting, it is a way of repeating a k-step arnoldi iteration after improving the starting vector v_0 . A rapid convergence should occurs if we choose the starting vector v_0 of the arnoldi

process carefully.

Lemma 6.1 (Lemma 2.1 in [9]): if $v_0 \in \{\boldsymbol{W}_k\}$,where $\boldsymbol{S}_k \in \mathrm{C}^{m\times n}$ and $\boldsymbol{A}\boldsymbol{W}_k = \boldsymbol{W}_k\boldsymbol{G}_k$ for some $\boldsymbol{G}_k \in \mathrm{C}^{k\times k}$ then

$$\mathcal{K}(\boldsymbol{A}, \boldsymbol{v}_0; \boldsymbol{k}) \supset Span\{\boldsymbol{W}_k\} \tag{6.1}$$

and an arnoldi process with v_0 as the staring vector terminate in k or fewer steps.

Avoiding building a large Krylov subspace is another way to improve the speed of convergence, this again implies to initially carry out only k steps of Arnoldi iteration. If the k-dimension Krylov subspace fails to provide an accurate approximations, as it happened in the case of $\boldsymbol{M2}$, then re-computing an k-step Arnoldi factorization using a modified starting vector actually improves the convergence. This version of Arnoldi based on modifying the starting vector and refactorization procedure is called Restarted Arnoldi.

A strategy to implementing the new starting vector v_0 is to replace v_0 with $\Psi(\mathrm{A})v_0$ where $\Psi(\mathrm{A})$ is a function constructed to filter out the unwanted eigencomponents from v_0.

$$v_0 \leftarrow \Psi(A)v_0 \tag{6.2}$$

Restarted Arnoldi requires us to explicitly compute v_0 but the Implicitly Restarted Arnoldi algorithm (IRA) avoids to compute $\Psi(\mathrm{A})v_0$ directly and the new starting vector emerges as a by-product of a sequence of implicit QR updates.

Another method we can consider is a spectral transformation which is based on the idea of transforming the original eigenvalue problem, with a polynomial or rational transformation, into one that is easier to solve.

7 Conclusion

We now provide an overall comparison among the three methods and their variants.

In terms of speed the Conjugate Gradient method has showed the best performance, with the Fletcher-Reeves variant slightly outperforming the Rolak-Ribière one. The original version of the BFGS algorithm instead, while still being the slowest method for application concerning ill-conditioned matrices, like **M1**, has shown better speed than Arnoldi's take on the problem. Lastly, C-BFGS, the more cautious version of the algorithm, has showed to outperform its inspiration in all scenarios.

In order to provide a comparison of the results of the application of the methods on ill-conditioned matrices, we may need to account for the application-specific trade-off between accuracy and speed. C-BFGS has shown to be the most accurate but the CGD method has proved to be the fastest, on ill-conditioned matrix as on the other ones, while still providing a relatively good accuracy.

References

[1] J. Nocedal and S. J. Wright, *Numerical Optimization*, second. New York, NY, USA: Springer, 2006.

[2] D.-h. Li and M. Fukushima, "On the global convergence of bfgs method for nonconvex unconstrained optimization problems," *SIAM Journal on Optimization*, vol. 11, May 2000. DOI: 10.1137/S1052623499354242.

[3] W. F. Mascarenhas, "The bfgs method with exact line searches fails for non-convex objective functions," *Mathematical Programming*, vol. 99, pp. 49–61, 2004.

[4] Y.-H. Dai, "Convergence properties of the bfgs algoritm," *SIAM Journal on Optimization*, vol. 13, pp. 693–701, Jan. 2002. DOI: 10.1137/S1052623401383455.

[5] D.-H. Li and M. Fukushima, "A modified bfgs method and its global convergence in nonconvex minimization," *Journal of Computational and Applied Mathematics*, vol. 129, no. 1, pp. 15–35, 2001, Nonlinear Programming and Variational Inequalities, ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00540-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042700005409.

[6] G. Yuan, Z. Sheng, B. Wang, W. Hu, and C. Li, "The global convergence of a modified bfgs method for nonconvex functions," *Journal of Computational and Applied Mathematics*, vol. 327, pp. 274–294, 2018, ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2017.05.030. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042717302753.

[7] P. Li, J. Lu, and H. Feng, "The global convergence of a modified bfgs method under inexact line search for nonconvex functions," *Mathematical Problems in Engineering*, vol. 2021, pp. 1–9, May 2021. DOI: 10.1155/2021/8342536.

[8] L. N.Trefeethen and D. Bau, *Numerical Linear Algebra*. OXford University: Society for Industrial and Applied Mathematics, 1997.

[9] C. Yang, *Accelerating the Arnoldi Iteration - Theory and Practice*. RICE University, 1998.