Numerical Optimisation and Inverse Problems Project 2

Edward Small 10786391

February 2021

Contents

0.1 Preamble

The following work explores the mathematics and algorithms behind estimating the minimum of multi-dimensional functions. Here, we will make some comments on notation, and note the specific equation we will consider when analysing the minimisation techniques.

0.1.1 Notation

We consider the general quadratic function in the form of

$$Q(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + c \tag{1}$$

where

- $A \in \mathbb{R}^{n\times n}$ is a symmetric, positive definite matrix.
- $\boldsymbol{x} \in \mathbb{R}^n$ is the matrix of variables
- $\boldsymbol{b} \in \mathbb{R}^n$
- $c \in \mathbb{R}$

We denote the Laplacian of Q such that

$$\nabla Q(\boldsymbol{x}) = g(\boldsymbol{x}) = A\boldsymbol{x} + b \tag{2}$$

Worth noting here is that the Hessian matrix $H = \nabla g(\boldsymbol{x}) = \nabla^2 Q(\boldsymbol{x}) = A$. We also say that a set of vectors $\{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_{n-1}\}$ are conjugate with respect to A if

$$\boldsymbol{p}_i^T A\boldsymbol{p}_j = 0 \text{ for all } i \neq j \tag{3}$$

0.1.2 Code

All code for all plots, results, and algorithms will be provided in the appendix. I plan on making a full script that will obtain all necessary results and plots for all questions, which will call on separate functions to run specific algorithms as necessary. Whilst I do plan on implementing every algorithm in Matlab, I will also be doing the work by hand (and then checking the results against the algorithms I have written in Matlab). This is done for two reasons:

1. Completeness
2. My own education and understanding

Furthermore, live code is available to copy from git (URL is available as a footnote[1] on this page). The code is written in a vector/matrix form. Whilst this looks a little confusing, this was done because the nature of the questions require me to know the values of each variable at each step, and this was the easiest way to store them for later analysis.

0.2 Problem Function

For the questions we will consider the following function

$$f(x_1, x_2) = \frac{1}{2}\begin{bmatrix} x_1 & x_2 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4}$$

[1] https://github.com/Teddyzander/NumericalOptimisationandInverseProblems/tree/master/Coursework2

and therefore we can say

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad c = 0 \tag{5}$$

Finally, we can then define the gradient and the Hessian

$$g(\boldsymbol{x}) = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad H = A = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \tag{6}$$

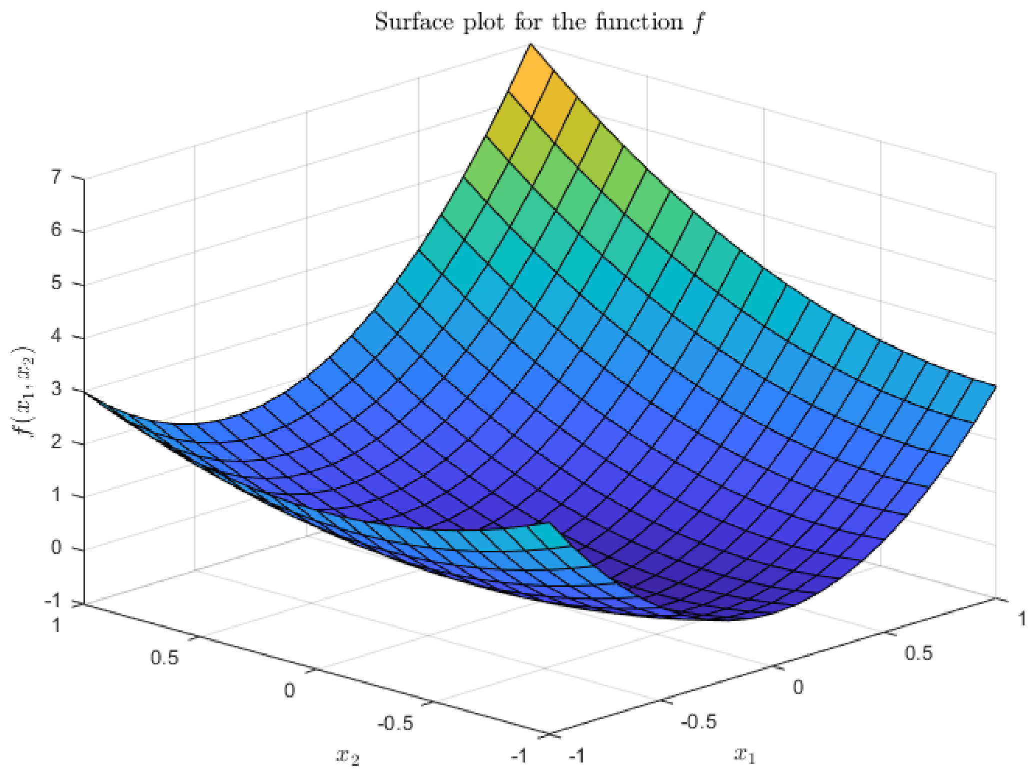

Figure 1: Shows a surface plot of f for $x_1 \in [-1, 1]$ and $x_2 \in [-1, 1]$

1 Steepest Descent with Exact Line-search

1.1 Deriving the Algorithm

For the steepest descent algorithm with exact line search, we start with an initial guess for the minimum and label it $\boldsymbol{x}_0$. We then analyse the gradient at this point to find the steepest direction to get

$$\boldsymbol{p}_0 = -g(\boldsymbol{x}_0) \tag{7}$$

Now that we have found a direction, we need to find how far we should descend before beginning a new iteration. In this method we use *exact line-search*, so we want to find an α_0 such that

$$Q(\boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0) = \min_{\alpha>0} Q(x_0 + \alpha \boldsymbol{p}_0) \tag{8}$$

To do this exactly, we can consider the Taylor expansion of a quadratic form with this specific shift, so

$$Q(\boldsymbol{x} + \alpha \boldsymbol{p}) = Q(\boldsymbol{x}) + \alpha \boldsymbol{p}^T g(\boldsymbol{x}) + \frac{1}{2}\alpha^2 \boldsymbol{p}^T H \boldsymbol{p} \tag{9}$$

We therefore need to minimise this function, with the variable of interest being α. This means

$$\frac{dQ(\boldsymbol{x} + \alpha \boldsymbol{p})}{d\alpha} = \boldsymbol{p}^T g(\boldsymbol{x}) + \alpha \boldsymbol{p}^T H \boldsymbol{p} = 0 \tag{10}$$

We can find the minimum α with a little rearranging to get

$$\alpha = \frac{-\boldsymbol{p}^T g(\boldsymbol{x})}{\boldsymbol{p}^T H \boldsymbol{p}} \tag{11}$$

Computing this for large systems can be computationally expensive, and relies on us knowing the Hessian. In this case, we know that $H = A$, so the k^{th} step in the iteration then becomes

$$\alpha_k = \frac{-\boldsymbol{p}_k^T g(\boldsymbol{x}_k)}{\boldsymbol{p}_k^T A \boldsymbol{p}_k} \tag{12}$$

From here, we update a new estimate for the minimum

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k \tag{13}$$

We can then repeat this process for a fixed number of iterations, or until $||g(\boldsymbol{x}_k)|| < \text{TOL}$, where TOL is some small tolerance larger than 0.

1.2 Applying the Algorithm by Hand

Here we consider two steps of the steepest descent algorithm. Starting with $\boldsymbol{x}_0 = [0, 0]^T$, we first need to evaluate the negative of the gradient of $f(\boldsymbol{x}_0)$ at this point, $\boldsymbol{p}_0$.

$$\begin{aligned} \boldsymbol{p}_0 &= -g(\boldsymbol{x}_0) \\ &= -\left(\begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= -\begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned} \tag{14}$$

Now we require α_0 to find how far to descend in this step

$$\begin{aligned}
\alpha_0 &= \frac{-\boldsymbol{p}_0^T g(\boldsymbol{x}_0)}{\boldsymbol{p}_0^T A \boldsymbol{p}_0} \\
&= \frac{\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}}{-\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}2 & 1\\1 & 6\end{bmatrix}\left(-\begin{bmatrix}1\\1\end{bmatrix}\right)} \\
&= \frac{2}{10} \\
&= \frac{1}{5}
\end{aligned} \tag{15}$$

We can then find a new minimiser estimate $\boldsymbol{x}_1$ by

$$\begin{aligned}
\boldsymbol{x}_1 &= \boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0 \\
&= \begin{bmatrix}0\\0\end{bmatrix} + \frac{1}{5}\left(-\begin{bmatrix}1\\1\end{bmatrix}\right) \\
&= \begin{bmatrix}-\frac{1}{5}\\-\frac{1}{5}\end{bmatrix}
\end{aligned} \tag{16}$$

This completes 1 step of the algorithm. We repeat again for step a second step

$$\begin{aligned}
\boldsymbol{p}_1 &= -\left(\begin{bmatrix}2 & 1\\1 & 6\end{bmatrix}\begin{bmatrix}-\frac{1}{5}\\-\frac{1}{5}\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}\right) \\
&= \begin{bmatrix}-\frac{2}{5}\\\frac{2}{5}\end{bmatrix}
\end{aligned} \tag{17}$$

$$\begin{aligned}
\alpha_1 &= \frac{\left(-\begin{bmatrix}-\frac{2}{5} & \frac{2}{5}\end{bmatrix}\right)\begin{bmatrix}\frac{2}{5}\\-\frac{2}{5}\end{bmatrix}}{\begin{bmatrix}-\frac{2}{5} & \frac{2}{5}\end{bmatrix}\begin{bmatrix}2 & 1\\1 & 6\end{bmatrix}\begin{bmatrix}-\frac{2}{5}\\\frac{2}{5}\end{bmatrix})} \\
&= \frac{(\frac{8}{25})}{(\frac{24}{25})} \\
&= \frac{1}{3}
\end{aligned} \tag{18}$$

$$\begin{aligned}
\boldsymbol{x}_2 &= \boldsymbol{x}_1 + \alpha_1 \boldsymbol{p}_1 \\
&= \begin{bmatrix}-\frac{1}{5}\\-\frac{1}{5}\end{bmatrix} + \frac{1}{3}\begin{bmatrix}-\frac{2}{5}\\\frac{2}{5}\end{bmatrix} \\
&= \begin{bmatrix}-\frac{1}{3}\\-\frac{1}{15}\end{bmatrix}
\end{aligned} \tag{19}$$

1.3 Matlab Output

This algorithm was also implemented in Matlab (see section A.2.1). The algorithm was run for two steps to check the results of the hand-calculated section. The results (in four decimal places) for an initial guess of $\boldsymbol{x}_0 = [0, 0]^T$ are as follows.

1.3.1 Results for $k = 0$

$$\boldsymbol{p}_0 = \begin{bmatrix}-1\\-1\end{bmatrix} \qquad \alpha_0 = 0.2 \qquad \boldsymbol{x}_1 = \begin{bmatrix}-0.2\\-0.2\end{bmatrix} \tag{20}$$

which agrees with the hand calculated results.

1.3.2 Results for $k = 1$

Using $\boldsymbol{x}_1$ calculated in the previous section

$$\boldsymbol{p}_1 = \begin{bmatrix} -0.4 \\ 0.4 \end{bmatrix} \qquad \alpha_1 = 0.3333 \qquad \boldsymbol{x}_2 = \begin{bmatrix} -0.3333 \\ -0.0667 \end{bmatrix} \tag{21}$$

which agrees with the hand calculated results.

1.3.3 Path

We can plot the function $f(\boldsymbol{x})$ to see if this is the kind of behaviour we would expect to see.

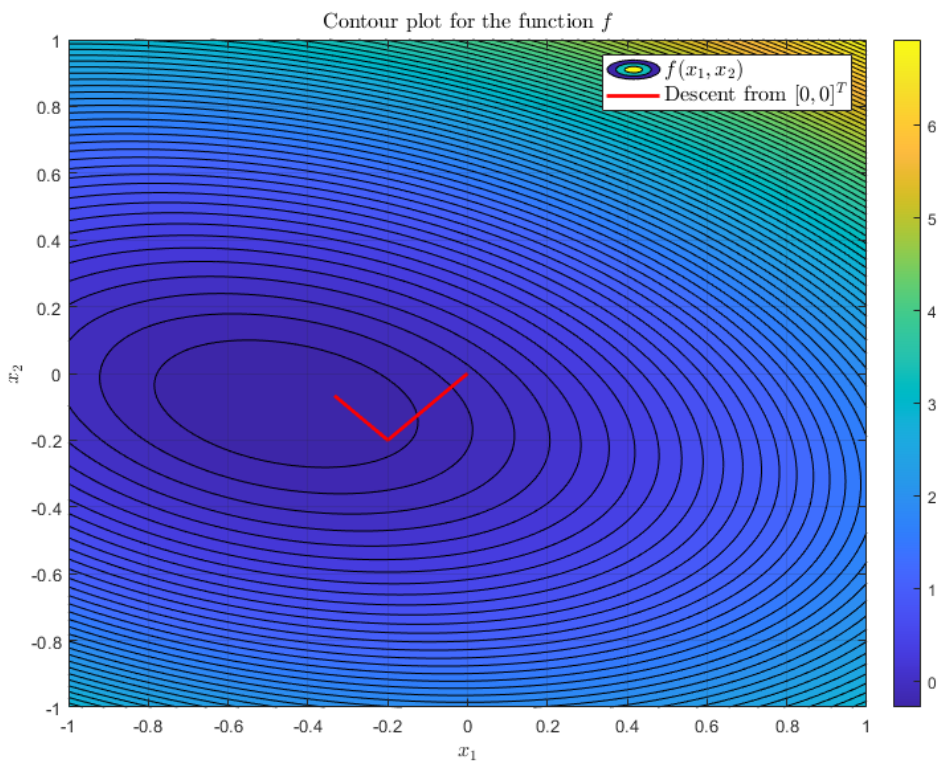

Figure 2: Shows the descent path of two iterations of steepest descent, starting with $x_0 = [0, 0]^T$

We can see from figure 2 that the descent path and behaviour looks as expected. The initial guess for x_0 is already close to the minimum, but we can see that x_2 is closer still. We also get the slight zig-zagging path we would expect to see using steepest descent, and the descent direction is (from visual inspection) perpendicular to the tangent of the contour line at each point (so the steepest direction).

Importantly, this algorithm does not calculate the true minimum in as fewer steps when compared to other algorithms, although each step is computationally cheap compared to some of the others. In fact, running steepest descent for 2 steps was significantly faster than running DFP for 2 steps, according to the Matlab profiler (see section 4 for details on DFP).

For the initial guess $x_0 = [0, 0]^T$, it takes 42 steps to converge to a tolerance of $g(\boldsymbol{x}_k) < 10^{-12}$ (although it does get four decimal places right in only 19 steps). The speed of convergence is also dependent, not just on the accuracy of the initial guess, but the shape of the function itself, especially near the minimum.

2 Newton Method with Natural Step-size

The Newton method is borne from estimating the quadratic function using the following Taylor series expansion

$$f(\boldsymbol{x}_k + \boldsymbol{p}) \approx Q_k(\boldsymbol{p}) = f(\boldsymbol{x}_k) + \boldsymbol{p}^T \boldsymbol{g}_k + \frac{1}{2}\boldsymbol{p}^T H_k \boldsymbol{p} \tag{22}$$

The reason this is just an estimate is because the Taylor series terms continue ad infinitum, but the Taylor series shown in (22) is cut off after 3 terms. We can therefore say that we are at minimum $\boldsymbol{p}$ when any other value increases Q_k, or

$$\nabla Q_k(\boldsymbol{p}) = \boldsymbol{g}_k + H_k \boldsymbol{p} = 0 \tag{23}$$

So, we require

$$\boldsymbol{p}_k = -H_k^{-1} \boldsymbol{g}_k \tag{24}$$

where $\boldsymbol{p}_k$ is the Newton direction. Clearly we require that the Hessian is invertible for each step (and inverting a large system can be computationally expensive). We can then update the estimate for the minimiser so that

$$\boldsymbol{x}_{k+1} = x_k + \boldsymbol{p}_k \tag{25}$$

2.1 Applying the Algorithm by Hand

Here we consider one step of the Newton method, starting with $\boldsymbol{x}_0 = [0, 0]^T$. We begin by defining the gradient and the Hessian at x_0.

$$\boldsymbol{g}_0 = A\boldsymbol{x}_0 + b = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \widetilde{H}_0 = \nabla^2 Q(\boldsymbol{x}_0) = A = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \tag{26}$$

Notice that the Hessian is a constant, and is not a function of $\boldsymbol{x}$. Not only that, but importantly it is equal to A, which is defined to be symmetric positive definite, so it is certainly invertible (an important criterion for this algorithm). We can then calculate the inverse of the Hessian

$$\begin{aligned} \widetilde{H}_0^{-1} &= \frac{1}{2 \times 6 - 1 \times 1} \begin{bmatrix} 6 & -1 \\ -1 & 2 \end{bmatrix} \\ &= \begin{bmatrix} \frac{6}{11} & \frac{-1}{11} \\ \frac{-1}{11} & \frac{2}{11} \end{bmatrix} \end{aligned} \tag{27}$$

and then find the Newton direction

$$\begin{aligned} \boldsymbol{p}_0 &= -\widetilde{H}_0^{-1} \boldsymbol{g}_0 \\ &= -\begin{bmatrix} \frac{6}{11} & \frac{-1}{11} \\ \frac{-1}{11} & \frac{2}{11} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix} \end{aligned} \tag{28}$$

which gives us the value for an estimated minimiser

$$\begin{aligned} \boldsymbol{x}_1 &= \boldsymbol{x}_0 + \boldsymbol{p}_{\mathbf{0}} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix} \end{aligned} \tag{29}$$

2.2 Matlab Output

This algorithm was also implemented in Matlab (see section A.2.2). The algorithm was run for one step to check the results of the hand-calculated section. The results (in four decimal places) for an initial guess of $x_0 = [0, 0]^T$ are as follows.

$$\boldsymbol{g}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \widetilde{H}_0 = \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \qquad \widetilde{H}_0^{-1} = \begin{bmatrix} 0.5455 & -0.0909 \\ -0.0909 & 0.1818 \end{bmatrix} \qquad \boldsymbol{p}_0 = \begin{bmatrix} -0.4545 \\ -0.0909 \end{bmatrix} \tag{30}$$

giving $\boldsymbol{x}_1 = \begin{bmatrix} -0.4545 \\ -0.0909 \end{bmatrix}$. This agrees with the hand calculated results.

2.2.1 Path

We can plot the function $f(\boldsymbol{x})$ to see if this is the kind of behaviour we would expect to see.

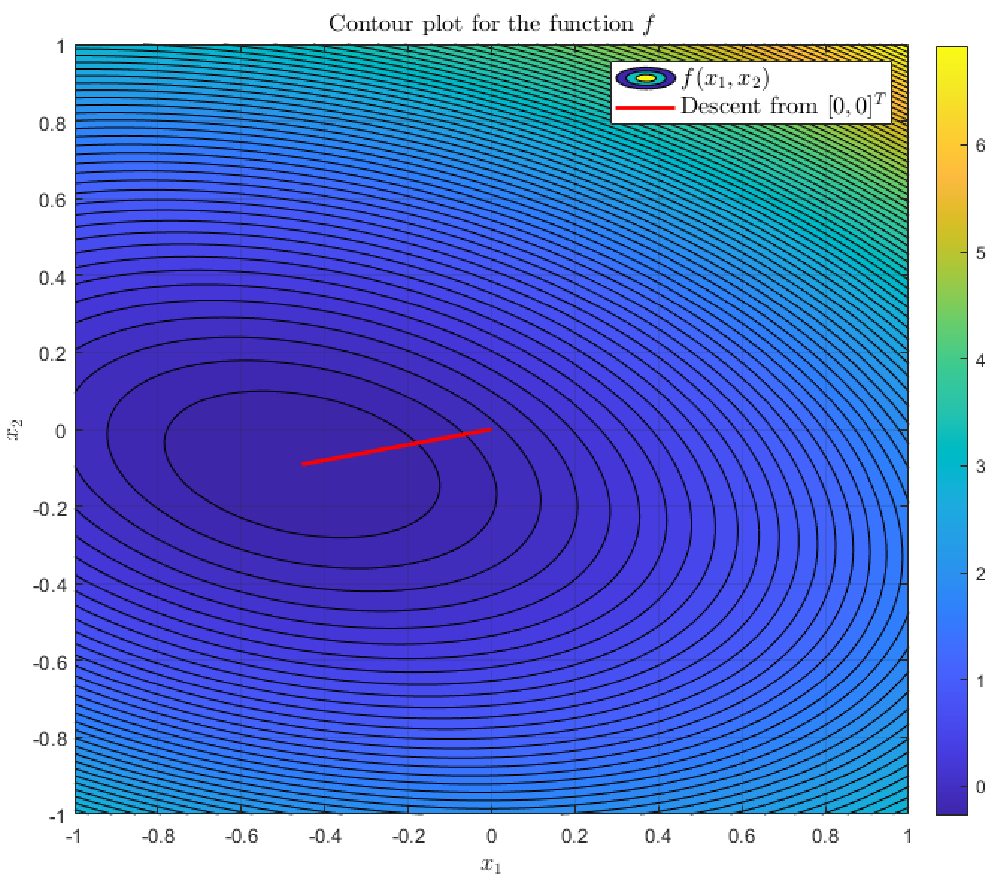

Figure 3: Shows the descent path of one iterations of Newton Method, starting with $x_0 = [0, 0]^T$

Figure 3 shows the descent path. Unsurprisingly, this algorithm manages to get to the actual minimum in a single step. This is because the Taylor series approximation for the function f isn't an approximation at all - it is exact! This is due to the fact that f only has two derivatives, and so a three term Taylor series actually gives an exact solution.

2.3 Analytical Minimum for f

We can show that the Newton method gives an exact minimum for f in one step by comparing the answer from the algorithm to the analytical minimum. To find a minimum we require that

$$\nabla f(\boldsymbol{x}^*) = A\boldsymbol{x}^* + b = 0 \tag{31}$$

which we call the *first-order necessary condition* (note: we already have the second order condition, as H is positive definite). Therefore the minimum is calculated by

$$\begin{aligned}\boldsymbol{x}^* &= -A^{-1}b \\ &= -\begin{bmatrix} \frac{6}{11} & \frac{-1}{11} \\ \frac{-1}{11} & \frac{2}{11} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix}\end{aligned} \tag{32}$$

This is the exact answer we get from Matlab and from the hand-calculated section. The reason for this is given in the previous section, but we will re-iterate it here. The Taylor series approximation for $f(\boldsymbol{x}_k + \boldsymbol{p}) \approx Q_k(\boldsymbol{p})$ is actually $f(\boldsymbol{x}_k + \boldsymbol{p}) = Q_k(\boldsymbol{p})$ because the function f has no further derivatives beyond H. Therefore, when we minimise $Q_k(\boldsymbol{p})$ we find the value of $\boldsymbol{p}$ that will take us directly from any initial guess for $\boldsymbol{x}_0$ to $\boldsymbol{x}^*$.

3 Conjugate Gradient Method with Exact Line-search

The conjugate gradient method relies on the property of conjugacy. A set of non-zero vectors $\boldsymbol{p}$ are said to be conjugate to a symmetrix positive definite matrix $A \in \mathbb{R}^{n\times n}$ if

$$\boldsymbol{p}_i^T A \boldsymbol{p}_j = 0 \text{ for all } i \neq j \tag{33}$$

where

- $\boldsymbol{p} = \{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_{n-1}\}$
- The vectors in $\boldsymbol{p}$ are linearly independent
- The set of eigenvectors of A are always conjugate

We then pick an initial guess for the minimum $\boldsymbol{x}_0$ and consider the vector $\boldsymbol{p}$ as search directions. We set the first direction $\boldsymbol{p}_0 = -g(\boldsymbol{x}_0)$. We want to update the guess for the minimum $\boldsymbol{x}$ such that $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$, where α_k satisfies

$$\min_{\alpha>0} Q(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \tag{34}$$

This minimum was derived in section 1.1. The final derivation shows that

$$\alpha_k = \frac{-\boldsymbol{p}_k^T g(\boldsymbol{x}_k)}{\boldsymbol{p}_k^T A \boldsymbol{p}_k} \tag{35}$$

We can use this value to update $\boldsymbol{x}_{k+1}$ as shown above, and find a value for $\boldsymbol{g}_{k+1} = g(\boldsymbol{x}_{k+1})$. To ensure that the value of $\boldsymbol{p}_{k+1}$ is conjugate with respect to A, we find a scalar β_k and apply it such that

$$\beta_k = \frac{\boldsymbol{g}_{k+1}^T \boldsymbol{g}_{k+1}}{\boldsymbol{g}_k^T \boldsymbol{g}_k} \qquad \boldsymbol{p}_{k+1} = -\boldsymbol{g}_{k+1} + \beta_k \boldsymbol{p}_k \tag{36}$$

This algorithm converges in, at most, n steps, as we only have n amount of conjugate vectors. This is because each iteration of x_{k+1} is a minimum of Q in the subspace $x_0 + \text{span}\{\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_k\}$, so when $k = n-1$ x_{k+1} is no longer the minimum of a subspace of Q, it is the minimum of the space Q.

3.1 Applying the Algorithm by Hand

Here we consider two steps of the conjugate gradient method, starting with $x_0 = [0,0]^T$. We start by finding the values for $\boldsymbol{g}_0$ and $\boldsymbol{p}_0$, so

$$\begin{aligned} \boldsymbol{g}_0 &= g(\boldsymbol{x}_0) \\ &= \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned} \tag{37}$$

$$\boldsymbol{p}_0 = -\boldsymbol{g}_0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{38}$$

We then find the minimum α_0 that minimises $f(\boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0)$

$$\begin{aligned} \alpha_0 &= \frac{-\boldsymbol{p}_0^T \boldsymbol{g}_0}{\boldsymbol{p}_0^T A \boldsymbol{p}_0} \\ &= \frac{\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}}{\begin{bmatrix} -1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix}} \\ &= \frac{1}{5} \end{aligned} \tag{39}$$

Now we can update $\boldsymbol{x}$ to find a new estimated minimiser

$$\begin{aligned}
\boldsymbol{x}_1 &= \boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0 \\
&= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \frac{1}{5} \begin{bmatrix} -1 \\ -1 \end{bmatrix} \\
&= \begin{bmatrix} -\frac{1}{5} \\ -\frac{1}{5} \end{bmatrix}
\end{aligned} \tag{40}$$

Next we prepare for the next iteration by finding the gradient at the new minimiser, and using it to create a new vector $\boldsymbol{p}_1$ that is conjugate with respect to A.

$$\begin{aligned}
\boldsymbol{g}_1 &= g(\boldsymbol{x}_1) \\
&= \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} -\frac{1}{5} \\ -\frac{1}{5} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{2}{5} \\ -\frac{2}{5} \end{bmatrix}
\end{aligned} \tag{41}$$

$$\begin{aligned}
\beta_0 &= \frac{\boldsymbol{g}_1^T \boldsymbol{g}_1}{\boldsymbol{g}_0^T \boldsymbol{g}_0} \\
&= \frac{\begin{bmatrix} \frac{2}{5} & -\frac{2}{5} \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ -\frac{2}{5} \end{bmatrix}}{\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}} \\
&= \frac{(\frac{8}{25})}{2} \\
&= \frac{4}{25}
\end{aligned} \tag{42}$$

and therefore

$$\begin{aligned}
\boldsymbol{p}_1 &= -\boldsymbol{g}_1 + \beta_0 \boldsymbol{p}_0 \\
&= \begin{bmatrix} -\frac{2}{5} \\ \frac{2}{5} \end{bmatrix} + \frac{4}{25} \begin{bmatrix} -1 \\ -1 \end{bmatrix} \\
&= \begin{bmatrix} -\frac{14}{25} \\ \frac{6}{25} \end{bmatrix}
\end{aligned} \tag{43}$$

We can then apply this vector in a new iteration to find a new estimate for the minimiser.

$$\begin{aligned}
\alpha_1 &= \frac{-\boldsymbol{p}_1^T \boldsymbol{g}_1}{\boldsymbol{p}_1^T A \boldsymbol{p}_1} \\
&= \frac{\begin{bmatrix} \frac{14}{25} & -\frac{6}{25} \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ -\frac{2}{5} \end{bmatrix}}{\begin{bmatrix} -\frac{14}{25} & \frac{6}{25} \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} -\frac{14}{25} \\ \frac{6}{25} \end{bmatrix}} \\
&= \frac{(\frac{8}{25})}{(\frac{88}{125})} \\
&= \frac{5}{11}
\end{aligned} \tag{44}$$

which gives

$$\begin{aligned}\boldsymbol{x}_2 &= \boldsymbol{x}_1 + \alpha_1 \boldsymbol{p}_1 \\ &= \begin{bmatrix} -\frac{1}{5} \\ -\frac{1}{5} \end{bmatrix} + \frac{5}{11} \begin{bmatrix} -\frac{14}{25} \\ \frac{6}{25} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix}\end{aligned} \tag{45}$$

This value should be the true minimum, since $A \in \mathbb{R}^{2\times 2}$ and we have completed two steps of the algorithm. We can confirm this is the true minimum by comparing it to the analytical solution in section 2.3.

3.2 Matlab Output

This algorithm was also implemented in Matlab (see section A.2.3). The algorithm was run for two steps to check the results of the hand-calculated section. The results (in four decimal places) for an initial guess of $x_0 = [0, 0]^T$ are as follows.

$$\boldsymbol{x}_1 = \begin{bmatrix} -0.2 \\ -0.2 \end{bmatrix} \qquad \boldsymbol{g}_1 = \begin{bmatrix} 0.4 \\ -0.4 \end{bmatrix} \qquad \beta_0 = 0.16 \qquad \boldsymbol{p}_1 = \begin{bmatrix} -0.56 \\ 0.24 \end{bmatrix} \qquad \boldsymbol{x}_2 = \begin{bmatrix} -0.4545 \\ -0.0909 \end{bmatrix} \tag{46}$$

This agrees with the hand calculated results.

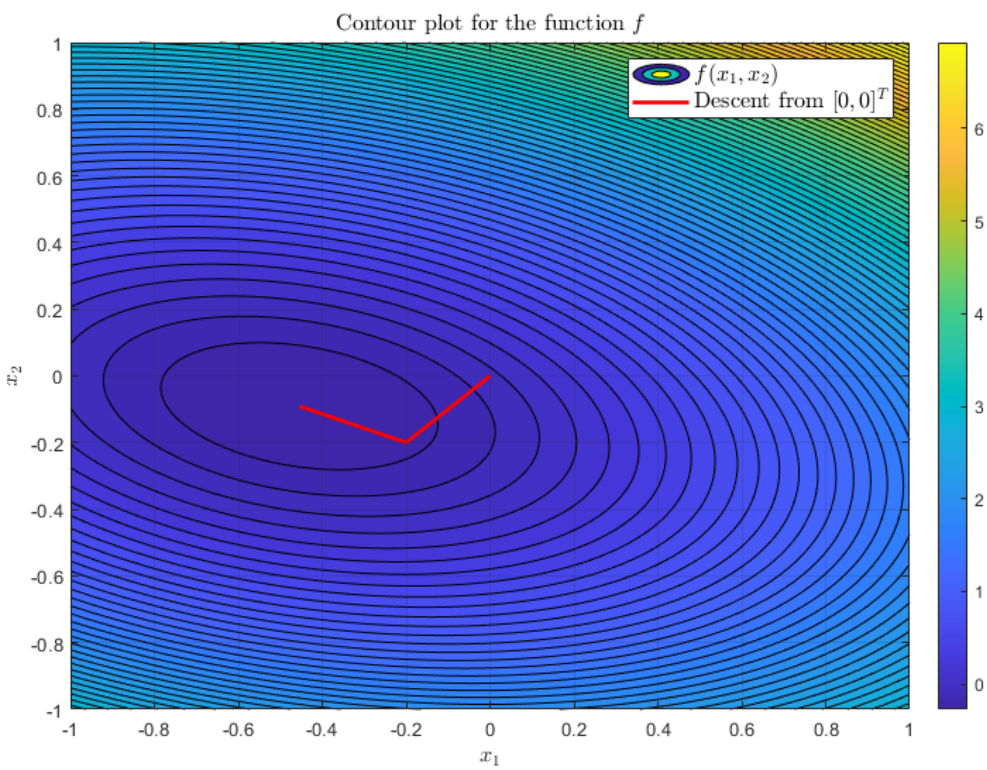

Figure 4: Shows the descent path of two iterations of conjuate method, starting with $x_0 = [0, 0]^T$. On visual inspection, it seems to go to the minimum, agreeing with the theory

4 DFP Method with Exact Line-search

The DFP Quasi-Newton method differs to the other algorithms in that it uses an estimate of the Hessian to create the vectors $\boldsymbol{p}_k$. We then run an exact line search to minimise Q with some shift in the $\tau\boldsymbol{p}_k$ direction. Lastly, we update the minimum in the usual way, and create a new estimated Hessian before running another iteration. Equations of note here, then, are

- The construction of $\boldsymbol{p}$
$$\boldsymbol{p}_{k+1} = -H_k^{-1}\boldsymbol{g}_k \tag{47}$$
- The calculation of τ
$$\begin{aligned}\tau_{k+1} &= \text{argmin}_{\tau>0} Q(\boldsymbol{x}_k + \tau\boldsymbol{p}_{k+1}) \\ &= \frac{-\boldsymbol{p}_{k+1}^T\boldsymbol{g}_k}{\boldsymbol{p}_{k+1}^T A\boldsymbol{p}_{k+1}}\end{aligned} \tag{48}$$
  which is exact line search, similar to previous sections but instead using $\boldsymbol{p}_{k+1}$. Some algorithms use an estimate H_k in place of A performing a line-search. However, we want exact line-search, so we will be using the actual value of A, not an estimate.
- The construction of the new Hessian estimate
$$H_{k+1} = \left(I - \frac{\boldsymbol{y}_k\boldsymbol{s}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k}\right)H_k\left(I - \frac{\boldsymbol{s}_k\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k}\right) + \frac{\boldsymbol{y}_k\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k} \tag{49}$$
  where
  - $\boldsymbol{s_k} = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
  - $\boldsymbol{y_k} = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$

  Importantly, this construction guarantees that each Hessian estimate is symmetric positive definite, and therefore invertible (which is crucial for finding the $\boldsymbol{p}$ vectors). We also need an initial guess for H_0. Some literature suggests using the identity matrix I, but the initial guess must be symmetric positive definite, and preferably close to the value of the actual Hessian.

4.1 Applying the Algorithm by Hand

Here we consider two steps of the DFP method, starting with $x_0 = [0,0]^T$ and $H_0 = \begin{bmatrix}2 & 0\\0 & 6\end{bmatrix}$. We start by finding the setting
$$\boldsymbol{g}_0 = g(\boldsymbol{x}_0) = \begin{bmatrix}1\\1\end{bmatrix} \tag{50}$$
to find
$$\begin{aligned}\boldsymbol{p}_1 &= -H_0^{-1}\boldsymbol{g}_0 \\ &= -\frac{1}{6}\begin{bmatrix}6 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} \\ &= \begin{bmatrix}-\frac{1}{2}\\-\frac{1}{6}\end{bmatrix}\end{aligned} \tag{51}$$

Now we minimise

$$\begin{aligned}\tau_1 &= \text{argmin}_{\tau>0} Q(\boldsymbol{x}_k + \tau \boldsymbol{p}_{k+1}) \\ &= \frac{-\boldsymbol{p}_1^T \boldsymbol{g}_0}{\boldsymbol{p}_1^T A \boldsymbol{p}_1} \\ &= \frac{\begin{bmatrix}\frac{1}{2} & \frac{1}{6}\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}}{\begin{bmatrix}-\frac{1}{2} & -\frac{1}{6}\end{bmatrix}\begin{bmatrix}2 & 1\\1 & 6\end{bmatrix}\begin{bmatrix}-\frac{1}{2}\\-\frac{1}{6}\end{bmatrix}} \\ &= \frac{(\frac{4}{6})}{(\frac{5}{6})} \\ &= \frac{4}{5}\end{aligned} \tag{52}$$

This information allows for an updated minimum to be calculated

$$\begin{aligned}x_1 &= x_0 + \tau_1 \boldsymbol{p}_1 \\ &= \begin{bmatrix}0\\0\end{bmatrix} + \frac{4}{5}\begin{bmatrix}-\frac{1}{2}\\-\frac{1}{6}\end{bmatrix} \\ &= \begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix}\end{aligned} \tag{53}$$

and the gradient at this point

$$\begin{aligned}\boldsymbol{g}_1 &= g(\boldsymbol{x}_1) \\ &= \begin{bmatrix}2 & 1\\1 & 6\end{bmatrix}\begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix} \\ &= \begin{bmatrix}\frac{1}{15}\\-\frac{1}{5}\end{bmatrix}\end{aligned} \tag{54}$$

and a new Hessian estimate. First, we need to establish the difference between gradients and minimum estimates from this iteration and the previous iteration.

$$\boldsymbol{s}_0 = \boldsymbol{x}_1 - \boldsymbol{x}_0 = \begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix} - \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix} \tag{55}$$

$$\boldsymbol{y}_0 = \boldsymbol{g}_1 - \boldsymbol{g}_0 = \begin{bmatrix}\frac{1}{15}\\-\frac{1}{5}\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}-\frac{14}{15}\\-\frac{6}{5}\end{bmatrix} \tag{56}$$

Now we can construct a new Hessian estimate

$$\begin{aligned}H_1 &= \left(I - \frac{\boldsymbol{y}_0 \boldsymbol{s}_0^T}{\boldsymbol{y}_0^T \boldsymbol{s}_0}\right) H_0 \left(I - \frac{\boldsymbol{s}_0 \boldsymbol{y}_0^T}{\boldsymbol{y}_0^T \boldsymbol{s}_0}\right) + \frac{\boldsymbol{y}_0 \boldsymbol{y}_0^T}{\boldsymbol{y}_0^T \boldsymbol{s}_0} \\ &= \left(\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} - \frac{\begin{bmatrix}-\frac{14}{15}\\-\frac{6}{5}\end{bmatrix}\begin{bmatrix}-\frac{2}{5} & -\frac{2}{15}\end{bmatrix}}{\begin{bmatrix}-\frac{14}{15} & -\frac{6}{5}\end{bmatrix}\begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix}}\right)\begin{bmatrix}2 & 0\\0 & 6\end{bmatrix}\left(\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} - \frac{\begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix}\begin{bmatrix}-\frac{14}{15} & -\frac{6}{5}\end{bmatrix}}{\begin{bmatrix}-\frac{14}{15} & -\frac{6}{5}\end{bmatrix}\begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix}}\right) + \frac{\begin{bmatrix}-\frac{14}{15}\\-\frac{6}{5}\end{bmatrix}\begin{bmatrix}-\frac{14}{15} & -\frac{6}{5}\end{bmatrix}}{\begin{bmatrix}-\frac{14}{15} & -\frac{6}{5}\end{bmatrix}\begin{bmatrix}-\frac{2}{5}\\-\frac{2}{15}\end{bmatrix}} \\ &= \begin{bmatrix}\frac{1}{3} & -\frac{7}{30}\\-\frac{9}{10} & \frac{7}{10}\end{bmatrix}\begin{bmatrix}2 & 0\\0 & 6\end{bmatrix}\begin{bmatrix}\frac{1}{3} & -\frac{9}{10}\\-\frac{7}{30} & \frac{7}{10}\end{bmatrix} + \begin{bmatrix}\frac{49}{30} & \frac{21}{100}\\\frac{21}{100} & \frac{27}{100}\end{bmatrix} \\ &= \begin{bmatrix}\frac{107}{50} & \frac{29}{50}\\\frac{29}{50} & \frac{363}{50}\end{bmatrix}\end{aligned} \tag{57}$$

Having a new estimate for the Hessian allows us to repeat the calculations again to find a better estimate of the minimiser (in fact, because $A \in \mathbb{R}^{2\times 2}$, $k = 2$ should take us to the minimum exactly!). We utilise the newly found H_1 to calculate

$$\begin{aligned} \boldsymbol{p}_2 &= -H_1^{-1}\boldsymbol{g}_1 \\ &= \begin{bmatrix} -\frac{363}{760} & \frac{29}{760} \\ \frac{29}{760} & -\frac{107}{760} \end{bmatrix} \begin{bmatrix} \frac{1}{15} \\ -\frac{1}{5} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{3}{76} \\ \frac{7}{228} \end{bmatrix} \end{aligned} \tag{58}$$

which is used to minimise

$$\begin{aligned} \tau_2 &= \frac{-\boldsymbol{p}_2^T\boldsymbol{g}_1}{\boldsymbol{p}_2^T H_1 \boldsymbol{p}_2} \\ &= \frac{76}{55} \end{aligned} \tag{59}$$

This can then be applied to calculate the new minimum

$$\begin{aligned} x_2 &= x_1 + \tau_2 \boldsymbol{p}_2 \\ &= \begin{bmatrix} -\frac{2}{5} \\ -\frac{2}{15} \end{bmatrix} + \frac{76}{55} \begin{bmatrix} -\frac{3}{76} \\ \frac{7}{228} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{5}{11} \\ -\frac{1}{11} \end{bmatrix} \end{aligned} \tag{60}$$

This matches the analytical solution (as well as answers found from other algorithms). If we continued on to calculate H_2 we would find that

$$H_2 = A \tag{61}$$

4.2 Matlab Output

This algorithm was also implemented in Matlab (see section A.2.4). The algorithm was run for two steps to check the results of the hand-calculated section. The results (in four decimal places) for an initial guess of $x_0 = [0, 0]^T$ and $H_0 = \begin{bmatrix} 2 & 0 \\ 0 & 6 \end{bmatrix}$ are as follows.

$$\boldsymbol{p}_1 = \begin{bmatrix} -0.5 \\ -0.1667 \end{bmatrix} \qquad \boldsymbol{x}_1 = \begin{bmatrix} -0.4 \\ -0.1333 \end{bmatrix} \qquad H_1 = \begin{bmatrix} 2.14 & 0.58 \\ 0.58 & 7.26 \end{bmatrix} \tag{62}$$

$$\boldsymbol{p}_2 = \begin{bmatrix} -0.0395 \\ 0.0307 \end{bmatrix} \qquad \boldsymbol{x}_2 = \begin{bmatrix} -0.4545 \\ -0.0909 \end{bmatrix} \tag{63}$$

This agrees with the hand calculated results. Contnuing the calculation a little further confirms that $H_2 = A$

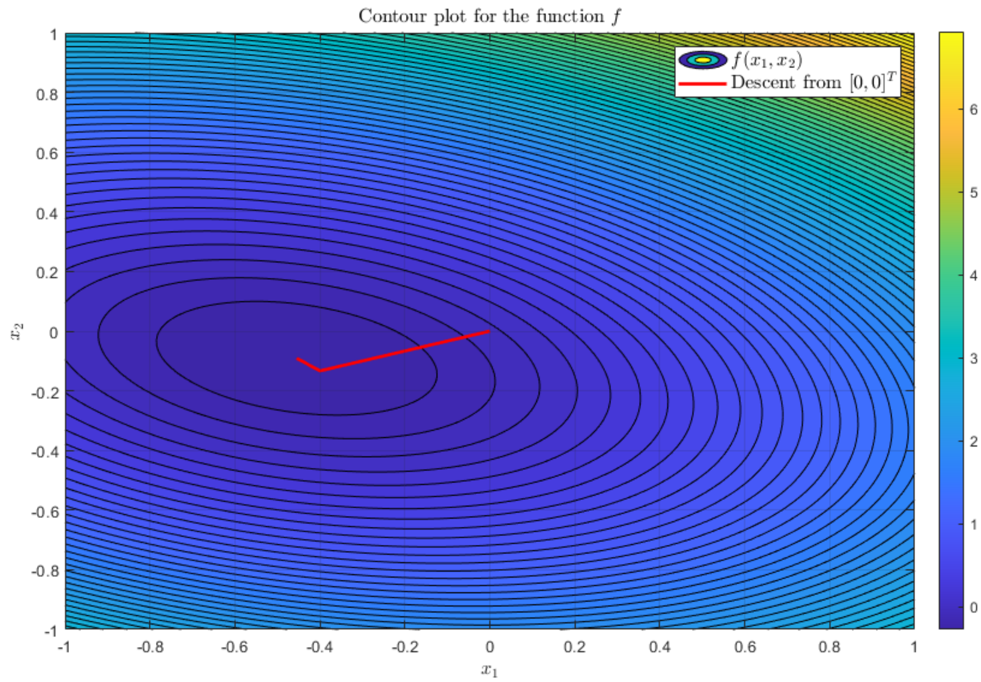

Figure 5: Shows the descent path of two iterations of DFP method, starting with $x_0 = [0,0]^T$, $H_0 = \begin{bmatrix} 2 & 0 \\ 0 & 6 \end{bmatrix}$. On visual inspection, it seems to go to the minimum, agreeing with the theory, and the first step takes us very close

5 Conjugacy of $\boldsymbol{p}$ with respect to A in DFP

5.0.1 Assumptions

In this section we consider the vector space $\{\boldsymbol{p}_0, \boldsymbol{p}_1, ...\}$ generated by the DFP algorithm, specifically its relationship with the symmetric postive definite matrix $A \in \mathbb{R}^{n \times n}$. We will assume we are minimising a general quadratic $Q(\boldsymbol{x})$. We will also assume that the initial guess for the Hessian H_0, as well as all further estimates of the Hessian H_k for $k = 1, 2, ...$ obtained from the algorithm are symmetric positive definite, and satisfy the secant equation

$$H_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k \tag{64}$$

where

- $\boldsymbol{s_k} = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$
- $\boldsymbol{y_k} = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$

5.0.2 Claim

For all values of $\boldsymbol{p}$ generated by this algorithm,

$$\boldsymbol{p}_{k+1}^T A \boldsymbol{p}_k = 0 \tag{65}$$

5.0.3 Proof

We must first note that each vector in $\boldsymbol{p}$ is made such that

$$\boldsymbol{p}_k = -H_{k-1}^{-1}\boldsymbol{g}_{k-1}, \qquad \boldsymbol{p}_{k+1} = -H_k^{-1}\boldsymbol{g}_k, \qquad \boldsymbol{p}_{k+2} = -H_{k+1}^{-1}\boldsymbol{g}_{k+1}, \qquad \ldots \tag{66}$$

where

$$H_{k+1} = \left(I - \frac{\boldsymbol{y}_k\boldsymbol{s}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k}\right)H_k\left(I - \frac{\boldsymbol{s}_k\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k}\right) + \frac{\boldsymbol{y}_k\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k} \tag{67}$$

which means, by construction, that $\boldsymbol{p}_{k+1}^T H_{k+1}\boldsymbol{p}_k = 0$. Using the definitions of $\boldsymbol{s}_k$, $\boldsymbol{y}_k$, and $\boldsymbol{g}_k$ we can see that

$$\begin{aligned} \boldsymbol{y}_k &= \boldsymbol{g}_{k+1} - \boldsymbol{g}_k \\ &= A\boldsymbol{x}_{k+1} + b - A\boldsymbol{x}_k - b \\ &= A\boldsymbol{x}_{k+1} - A\boldsymbol{x}_k \\ &= A(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) \\ &= A\boldsymbol{s}_k \end{aligned} \tag{68}$$

which means that $H_{k+1}\boldsymbol{s}_k = A\boldsymbol{s}_k$. Importantly, this does **not** imply that $H_{k+1} = A$, but it does mean that both matrices apply the same transformation on s_k. From the formation of the algorithm, we can also see that $x_k = x_{k-1} + \tau_k\boldsymbol{p}_k$, so a little rearranging and substituting gives

$$\boldsymbol{p}_k = \frac{\boldsymbol{s}_{k-1}}{\tau_k} \tag{69}$$

At this point, we are in a position where we can prove our claim. Start with our conjugate equation and make the substitutions using (66) and (69)

$$\begin{aligned} \boldsymbol{p}_{k+1}^T A\boldsymbol{p}_k &= (-H_k^{-1}\boldsymbol{g}_k)^T A\frac{\boldsymbol{s}_{k-1}}{\tau_k} \\ &= -H_k^{-1}\boldsymbol{g}_k^T A\frac{\boldsymbol{s}_{k-1}}{\tau_k} \end{aligned} \tag{70}$$

We can now substitute in the value for $H_{k+1}\boldsymbol{s}_k = A\boldsymbol{s}_k$ from (68), giving

$$\boldsymbol{p}_{k+1}^T A\boldsymbol{p}_k = -H_k^{-1}\boldsymbol{g}_k^T H_k\frac{\boldsymbol{s}_{k-1}}{\tau_k} \tag{71}$$

Applying a transformation and then applying the inverse is the same as not doing anything at all, so

$$\begin{aligned} \boldsymbol{p}_{k+1}^T A\boldsymbol{p}_k &= -\boldsymbol{g}_k^T\frac{\boldsymbol{s}_{k-1}}{\tau_k} \\ &= -\boldsymbol{g}_k^T\boldsymbol{p}_k \end{aligned} \tag{72}$$

However, we know from the construction of each $\boldsymbol{p}_k$ that

$$\boldsymbol{g}_k^T\boldsymbol{p}_k = 0 \tag{73}$$

Which therefore implies that

$$\boldsymbol{p}_{k+1}^T A\boldsymbol{p}_k = 0 \tag{74}$$

6 Behaviours Plot

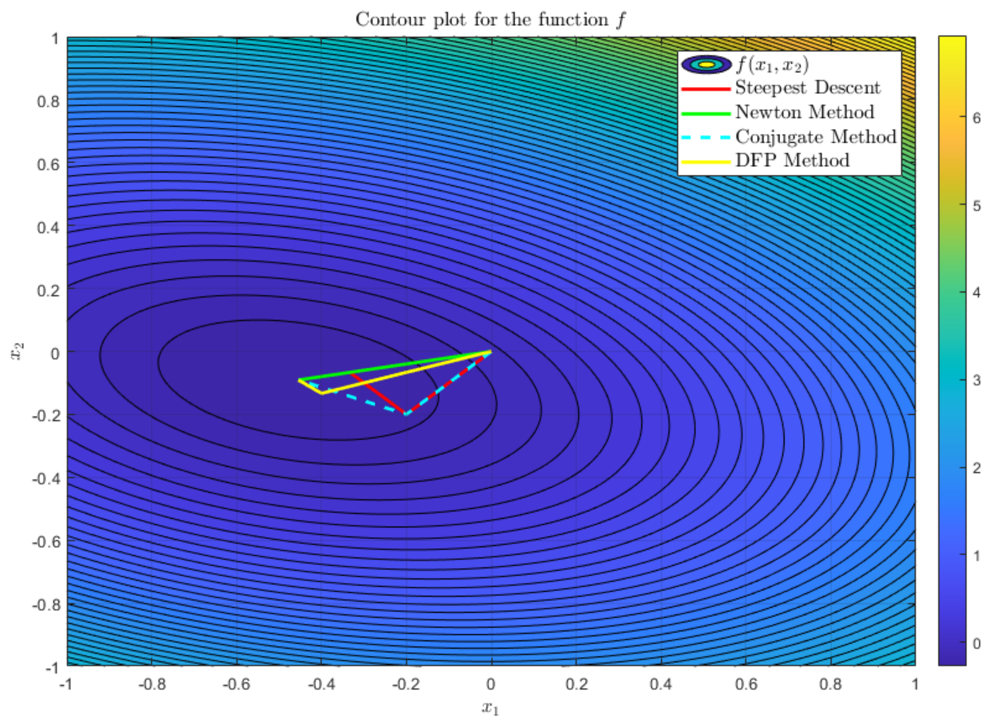

Figure 6: Shows the descent path of all the algorithms from the initial guess of $\boldsymbol{x}_0 = [0, 0]^T$

References

[1] Geraldine E. Myers (1968) 'Journal of Optimization Theory and Applications: Properties of the Conjugate-Gradient and Davidon Methods', *Plenum Publishing Corporation*, pp. 209–219.

[2] R. Fletcher and M. J. D. Powell (August 1963) 'A rapidly convergent descent method for minimization', *Oxford Academic*, pp. 163–168 .

A Matlab Code

A.1 General Scripts

A.1.1 Script to Run Results for all Questions

```
%%
% main script for coursework 2
% Focuses on minimising general quadratic functions in the form of
%       Q(x) = 0.5*x^T*A*X+b^T + c
% numerically

%% Define function variables. Plot the function over some interval

% define variables for the specific function Q
A = [2, 1;
    1, 6];
b = [1; 1];
c = 0;

% set up quadratic function
Q = @(x) (1/2)*x.'*A*x + b.'*x + c;

% set up gradient of quadratic g
g = @(x) A*x + b;

H = @(x) A;

% set up bounds for the plot
lower = -1;
upper = 1;
step = 0.01;

% plot the function, and get the surface/contour
[f_mesh, Q_plot] = f(Q, lower, upper, step, true, 'c', 0.1);

x_0 = [0;0]; % initial guess

%% Q1 find an estimated minimiser of Q using steepest descent for 2 steps

steps = 02; % number of steps
tol = 0.0001; % some tolerance for acceptable gradient at x_min

% apply steepest descent
[SD_x_min, SD_p, SD_alpha] = SteepestDescent(x_0, A, g, steps, tol);

% display values for each step
disp('------------------------------------------------------------------')
disp('STEEPEST DESCENT')
disp('Initial guess for x_0: [' + string(x_0(1)) + ' ' + ...
    string(x_0(2)) + ']')
for k=1:length(SD_alpha)
    disp('Values for step ' + string(k-1))
    disp('p: [' + string(SD_p(1, k)) + ' ' + string(SD_p(2, k)) + ']')
    disp('alpha: ' + string(SD_alpha(k)))
    disp('new min estimate: [' + string(SD_x_min(1, k+1)) + ' ' + ...
        string(SD_x_min(2, k+1)) + ']')
end
% make a plot to show how the descent behaved
figure(1)
hold on
plot(SD_x_min(1, :), SD_x_min(2, :), '-r', 'linewidth', 2);
legend('$f(x_1, x_2)$', 'Descent from $[0, 0]^T$', 'interpreter','latex', ...
    'FontSize',12,'FontWeight', 'bold');
grid on
```

```matlab

%% Q2 find an estimated minimiser of Q using Newton Method for 1 step

steps = 1; % number of steps
tol = 0.0001; % some tolerance for acceptable gradient at x_min

% Apple the Newton Method
[NM_x_min, NM_g, NM_H, NM_p, NM_alpha] = NewtonMethodNSS(x_0, g, H, ...
    steps, tol);

% display values for each step
disp('-----------------------------------------------------------------')
disp('NEWTON METHOD')
disp('Initial guess for x_0: [' + string(x_0(1)) + ' ' + ...
    string(x_0(2)) + ']')
for k=1:length(NM_alpha)
    disp('Values for step ' + string(k-1))
    disp('g: [' + string(NM_g(1, k)) + ' ' + string(NM_g(2, k)) + ']')
    disp('H:')
    NM_H
    disp('p: [' + string(NM_p(1, k)) + ' ' + string(NM_p(2, k)) + ']')
    disp('new min estimate: [' + string(NM_x_min(1, k+1)) + ' ' + ...
        string(NM_x_min(2, k+1)) + ']')
end
% make a plot to show how the descent behaved
figure(1)
hold on
plot(NM_x_min(1, :), NM_x_min(2, :), '-g', 'linewidth', 2);
legend('$f(x_1, x_2)$', 'Descent from $[0, 0]^T$', 'interpreter','latex', ...
    'FontSize',12,'FontWeight', 'bold');
grid on

%% Q3 find an estimated minimiser of Q using Conjugate Gradient Method
% for 2 steps

steps = 2; % number of steps
tol = 0.0001; % some tolerance for acceptable gradient at x_min

% Apple the Conjugate Gradient Method
[CG_x_min, CG_g, CG_p, CG_beta, CG_alpha] = ConjugateGradient(x_0, A, ...
    b, steps, tol);

% display values for each step
disp('-----------------------------------------------------------------')
disp('CONJUGATE GRADIENT METHOD')
disp('Initial guess for x_0: [' + string(x_0(1)) + ' ' + ...
    string(x_0(2)) + ']')
for k=1:length(CG_alpha)
    disp('Values for step ' + string(k-1))
    disp('g: [' + string(CG_g(1, k)) + ' ' + string(CG_g(2, k)) + ']')
    disp('p: [' + string(CG_p(1, k)) + ' ' + string(CG_p(2, k)) + ']')
    disp('alpha: ' + string(CG_alpha(k)))
    disp('beta: ' + string(CG_beta(k)))
    disp('new min estimate: [' + string(CG_x_min(1, k+1)) + ' ' + ...
        string(CG_x_min(2, k+1)) + ']')
end
% make a plot to show how the descent behaved
figure(1)
hold on
plot(CG_x_min(1, :), CG_x_min(2, :), '--c', 'linewidth', 2);
legend('$f(x_1, x_2)$', 'Descent from $[0, 0]^T$', 'interpreter','latex', ...
    'FontSize',12,'FontWeight', 'bold');
grid on

%% Q4 find an estimated minimiser of Q using DFP Method
```

```
% for 2 steps

steps = 2; % number of steps
tol = 0.0001; % some tolerance for acceptable gradient at x_min
H_0 = [2, 0; 0, 6]; % initial guess of hessian

% Apple the DFP Quasi-Newton method
[DFP_x_min, DFP_g, DFP_H, DFP_p] = DFP(x_0, H_0, A, ...
    g, steps, tol);

% display values for each step
disp('-----------------------------------------------------------------')
disp('DFP METHOD')
disp('Initial guess for x_0: [' + string(x_0(1)) + ' ' + ...
    string(x_0(2)) + ']')
for k=1:length(CG_alpha)
    disp('Values for step ' + string(k))
    disp('g: [' + string(DFP_g(1, k)) + ' ' + string(DFP_g(2, k)) + ']')
    disp('p(' + string(k) + '): [' + string(DFP_p(1, k+1)) + ' ' + ...
        string(DFP_p(2, k+1)) + ']')
    disp('H(' + string(k) + '): ')
    DFP_H(:, :, k+1)
    disp('new min estimate: [' + string(DFP_x_min(1, k+1)) + ' ' + ...
        string(DFP_x_min(2, k+1)) + ']')
end
% make a plot to show how the descent behaved
figure(1)
hold on
plot(DFP_x_min(1, :), DFP_x_min(2, :), '-y', 'linewidth', 2);
legend('$f(x_1, x_2)$', 'DFP Method', 'interpreter','latex', ...
    'FontSize',12,'FontWeight', 'bold');
grid on

%% If all functions have been run at once, make plot with legend for
% all descents
figure(1)
hold on
legend('$f(x_1, x_2)$', 'Steepest Descent', 'Newton Method', ...
    'Conjugate Method', 'DFP Method', 'interpreter','latex', ...
    'FontSize',12,'FontWeight', 'bold');
grid on
```

A.1.2 Script to Plot the Function f

```
% In this script we simply sketch, over some given space, a mesh
% of the 2D function we are minimising. This gives us a general idea of
% values we expect to see

% INPUTS
% func is the quadratic form of f with given matrices,
% l and u are the lower and upper bounds for the area to plot
% plot is a true/false bool for producing a plot of the function
% [if mesh is large, shading is interpolated (so it isn't just black!)]
% type is to choose between surface or contour ('s' for surface)
% consize is to change to levels in the contour plot

% OUTPUTS
% surface is the mesh for f evalulated between the bounds
% plot is the function plotted

function [surface, plot] = f(func, l, u, step, plot, type, consize)

% set up interval of interest, and allocate memory for mesh
x_int=[l:step:u];
y_int=[l:step:u];
```

```
surface=zeros(length(x_int), length(y_int));

% find the value of the function for
for i=1:length(x_int)
    for j=1:length(y_int)
        surface(i, j) = func([x_int(i); y_int(j)]);
    end
end

if plot == true
    if type == 's'
        plot = surf(y_int, x_int, surface);
        title('Surface plot for the function $f$', ...
            'interpreter','latex','FontSize',12);
        xlabel('$x_1$','interpreter','latex','FontSize',12,'FontWeight', 'bold');
        ylabel('$x_2$','interpreter','latex','FontSize',12,'FontWeight', 'bold');
        zlabel('$f(x_1, x_2)$','interpreter','latex','FontSize',12,'FontWeight', 'bold');

        if length(x_int) > 100 || length(y_int) > 100
            shading interp;
        end
    else
        [Y,X]=meshgrid(x_int, y_int);
        v=[min(surface(:)):consize:max(surface(:))];
        plot = contourf(X,Y,surface,v);
        title('Contour plot for the function $f$', ...
            'interpreter','latex','FontSize',12);
        xlabel('$x_1$','interpreter','latex','FontSize',12,'FontWeight', 'bold');
        ylabel('$x_2$','interpreter','latex','FontSize',12,'FontWeight', 'bold');
        colorbar;
    end
end
end
```

A.2 Minimisation Algorithms

A.2.1 Steepest Descent with Exact Line-search

```
% function to apply steepest descent (with exact alpha)
% ------------------------------------------------------------------
% INPUTS
% x_0 is an initial guess at the minimum
% A is the matrix of values from the standard quadratic form of Q
% g is the derivate of Q, the function we are evaluating
% steps is the maximum number of iterations
% tol is the tolerance for saying we have found a minimum
% ------------------------------------------------------------------
% OUTPUTS
% x is a matrix that holds every estimated minimiser
% p is a matrix hat holds the direction taken at each step
% alpha is a vector that holds the step size taken at each step

function [x, p, alpha] = SteepestDescent(x_0, A, g, steps, tol)

% allocate memory to save all minimums
x = zeros(length(x_0), length(1:steps));
p = zeros(length(x_0), length(1:steps)-1);
alpha = zeros(1, length(1:steps)-1);
% initialise first guess at minimum
x(:, 1) = x_0;
% start counter
iter = 0;

% estimate minimiser
for k = 1:steps

    % get the search direction
    p(:, k) = -g(x(:, k));
    % calculate how far to descend
    alpha(k) = (-p(:, k).'*g(x(:, k)))/(p(:, k).'*A*p(:, k));
    % update the minimiser
    x(:, k+1) = x(:, k) + alpha(k) * p(:, k);

    %increase iterator (so we can cut off excess if we find the min early)
    iter = iter + 1;
    % check to see if the gradient at the minimiser is small enough
    if norm(g(x(:, k+1))) < tol
        break;
    end
end

% ensure we only keep vectors from steps we have actually taken!
x = x(:, 1:iter+1);
p = p(:, 1:iter);
alpha = alpha(1:iter);
end
```

A.2.2 Newton Method with Natural Step-size

```
% function to apply Newton Method (with natural alpha)
% ------------------------------------------------------------------
% INPUTS
% x_0 is an initial guess at the minimum
% grad is the gradient function of the quadratic Q
% Hess is the Hessian of Q, the grad of grad
% steps is the maximum number of iterations
% tol is the tolerance for saying we have found a minimum
% ------------------------------------------------------------------
% OUTPUTS
```

```
% x is a matrix that holds every estimated minimiser
% g is a matrix that holds the gradient at each step
% H is a matrix that holds the value of the Hessian at each step
% p is a matrix that holds the direction at each step
% alpha is a vector that holds the step size taken at each step

function [x, g, H, p, alpha] = NewtonMethodNSS(x_0, grad, Hess, steps, tol)

% allocate memory to save all minimums
x = zeros(length(x_0), length(1:steps));
g = zeros(length(x_0), length(1:steps));
p = zeros(length(x_0), length(1:steps));
H = zeros(length(x_0), length(x_0), length(1:steps));
alpha = zeros(1, length(1:steps));
% initialise first guess at minimum
x(:, 1) = x_0;
% start counter
iter = 0;

% estimate minimiser
for k = 1:steps

    % get the search direction
    g(:, k) = grad(x(:, k));
    H(:, :, k) = Hess(x(:, k));
    % calculate the newton step
    p(:, k) = -inv(H(:, :, k)) * g(:, k);
    % natural step size
    alpha(k) = 1;
    % update the minimiser
    x(:, k+1) = x(:, k) + alpha(k) * p(:, k);

    % increase count
    iter = iter + 1;
    % check to see if the gradient at the minimiser is small enough
    if norm(grad(x(:, k+1))) < tol
        break;
    end
end

% ensure we only keep vectors from steps we have actually taken!
x = x(:, 1:iter+1);
g = g(:, 1:iter);
H = H(:, :, 1:iter);
p = p(:, 1:iter);
alpha = alpha(1:iter);
end
```

A.2.3 Conjugate Method with Exact Line-search

```
% function to apply Conjugate Method (with exact line search)
% -----------------------------------------------------------------
% INPUTS
% x_0 is an initial guess at the minimum
% A is the squared coeffecient from the standard Q
% b is the linear coeffeicent from the standard Q
% steps is the maximum number of iterations
% tol is the tolerance for saying we have found a minimum
% -----------------------------------------------------------------
% OUTPUTS
% x is a matrix that holds every estimated minimiser
% g is a matrix that holds the gradient at each step
% p is a matrix that holds the conjugate vectors with respect to A
% beta is a vector that holds scalars for adjusting minimiser
% alpha is a vector that holds the step size taken at each step
```

```
function [x, g, p, beta, alpha] = ConjugateGradient(x_0, A, b, steps, tol)

% allocate memory to save all minimums
x = zeros(length(x_0), length(1:steps));
g = zeros(length(x_0), length(1:steps));
p = zeros(length(x_0), length(1:steps));
beta = zeros(1, length(1:steps)-1);
alpha = zeros(1, length(1:steps)-1);
% initialise first guess at minimum, including gradient
x(:, 1) = x_0;
g(:, 1) = A * x_0 + b;
p(:, 1) = -g(:, 1);
% start counter
iter = 0;

% estimate minimiser
for k=1:steps

    % perform exact line search
    alpha(k) = -(g(:, k).'*p(:,k))/(p(:, k).'*A*p(:, k));
    % update gradient for next step
    g(:, k+1) = A*(x(:, k) + alpha(k)* p(:, k)) + b;
    %update the minimum
    x(:, k+1) = x(:, k) + alpha(k)*p(:, k);
    % calculate beta
    beta(k) = (g(:, k+1).'*g(:, k+1))/(g(:, k).'*g(:, k));
    % update direction
    p(:, k+1) = -g(:, k+1) + beta(k) * p(:, k);
    % increase count
    iter = iter + 1;
    % check to see if the gradient at the minimiser is small enough
    if norm(g(:, k+1)) < tol
        break;
    end
end

% ensure we only keep vectors from steps we have actually taken!
x = x(:, 1:iter+1);
g = g(:, 1:iter+1);
p = p(:, 1:iter+1);
alpha = alpha(1:iter);
beta = beta(1:iter);
end
```

A.2.4 DFP Method with Exact Line-search

```
% function to apply DFP Quasi-Newton Method (with exact line search)
% ----------------------------------------------------------------
% INPUTS
% x_0 is an initial guess at the minimum
% H_0 is an initial guess at the Hessian for Q
% grad is the gradient function of the quadratic Q
% A is the matrix coeffecient from Q
% steps is the maximum number of iterations
% tol is the tolerance for saying we have found a minimum
% ----------------------------------------------------------------
% OUTPUTS
% x is a matrix that holds every estimated minimiser
% g is a matrix that holds the gradient at each step
% p is a matrix that holds the conjugate vectors with respect to A
% beta is a vector that holds scalars for adjusting minimiser
% alpha is a vector that holds the step size taken at each step

function [x, g, H, p] = DFP(x_0, H_0, A, grad, ...
```

```
    steps, tol)

% allocate memory to hold values at each step
x = zeros(length(x_0), length(1:steps)+1);
H = zeros(length(x_0), length(x_0), length(1:steps)+1);
g = zeros(length(x_0), length(1:steps)+1);
p = zeros(length(x_0), length(1:steps)+1);
tau = zeros(1, length(1:steps)+1);

% initialise guesses
x(:, 1) = x_0;
H(:, :, 1) = H_0;
g(:, 1) = grad(x_0);

% initialise counter
iter = 0;
% get identity matrix
I = eye(length(x_0));

% Apply
for k=1:steps

    % compute Quasi-Newton step
    p(:, k+1) = -inv(H(:, :, k))* g(:, k);
    % get minimum of Q(x + tau * p)
    tau(k+1) = (-p(:, k+1).'*g(:, k))/(p(:, k+1).'*A*p(:, k+1));
    % estimate new minimum
    x(:, k+1) = x(:, k) + tau(k+1)*p(:, k+1);
    % find gradient at new minimum
    g(:, k+1) = grad(x(:, k+1));
    %find new estimate for the hessian
    s = x(:, k+1) - x(:, k);
    y = g(:, k+1) - g(:, k);
    H(:, :, k+1) = (I - (y * s.')/(y.'*s)) * ...
        H(:, :, k) * (I - (s*y.')/(y.'*s)) + (y*y.')/(y.'*s);
    iter = iter + 1;

    if norm(grad(x(:, k+1))) < tol
        break;
    end
end

% ensure we only keep vectors from steps we have actually taken!
x = x(:, 1:iter+1);
g = g(:, 1:iter+1);
H = H(:, :, 1:iter+1);
p = p(:, 1:iter+1);
end
```