Numerical Optimisation: Assignment 5

Student Number:13064947

February 2018

1 Exercise 1a

Submitted via Cody Coursework

2 Exercise 1b

Make your implementation efficient as explained in the lecture i.e. avoid explicitly forming the inverse Hessian matrix H_k and briefly explain what makes the implementation efficient.

I decided to implement the Limited-memory BFGS algorithm described in Chapter 7 of Nocedal. These are useful when the Hessian matrices (or our approximations to them) cannot be computed reasonably and where we do not wish to store dense 'n x n' represenions.

In the code below I will only be storing H as a vector and will be using an iterative method to return the vector comprising the Hessian multiplied by the gradient at each point (needed for the Newton method). Thus, in all cases I only use vectors multiplied by vectors. The first code basic set up is that - the Hessian is a vector of ones, we take a step and rescale.

```
H = ones(size(x0,1),1);
%first step was taken between set up and rescaling the H vector
.......
%This is for H0 which is a vector of ones to be rescaled
H = H_k(H);
```

Whereas previously we calculated one value for s_k and one for y_k, we are now going to store a set of m of these values (I chose m = 10). The if statement merely captures the case where the current number of iterations is not yet m. Our descent direction p_k is now calculated by a function written below. I am passing the gradient at the last x values, the set of m s_k vectors and the set of m y_k vectors. H remains a vector throughout.

```

if (nIter<=m)
    S(:,nIter) = s_k;
    Y(:,nIter) = y_k;
    p_k = -calcHg(F.df(x_k), S(:,1:nIter),Y(:,1:nIter),H);
elseif (nIter>m)
    S(:,1:(m-1))=S(:,2:m);
    Y(:,1:(m-1))=Y(:,2:m);
    S(:,m) = s_k;
    Y(:,m) = y_k;
    p_k = -calcHg(F.df(x_k), S,Y,H);
end
```

The code below returns the vector representing the Hessian multiplied by the gradient. The is equivalent to the L-BFGS two-loop recursion (Algorithm 7.4 - Given by Nocedal). We calculate our rho's for rescaling and use our history of s_k's, and the latest gradient to give a value q, which with an element by element product with our initial H gives a first return value estimate r. We then iterate through the second loop to give a final return value for r, the estimated hessian*gradient value.

```
function r = calcHg(g, S, Y, H)
    [n,k] = size(S);
    alpha =zeros(k,1);
    beta =zeros(k,1);
    r = zeros(n,1);
    q = g;

    for i = 1:k
        rho(i,1) = 1/(Y(:,i)'*S(:,i));
    end
    for i = k:-1:1
        alpha(i) = rho(i)*S(:,i)'*q;
        q = q - alpha(i)*Y(:,i);
    end
    %r = diag(H)*q; We dont need a matrix at all...
    r = H.*q;
    for i = 1:k
        beta(i) = rho(i)*Y(:,i)'*r;
        r = r + S(:,i)*(alpha(i)-beta(i));
    end
```

The key to the efficiency is not storing the 'n x n' hessian explicitly. (Where n is the dimensionality of our xs). By using a diagonal first estimate for the Hessian, we are now only doing vector by vector multiplications. Because there are two loops this is thus 4mn+n multiplications. (With m being a small value of our choosing).

3 Exercise 2

Submitted via Cody coursework

4 Exercise 3a

Minimise the function $f(x, y) = (x3y)^2 + x^4$ using BFGS and SR1 methods starting from $x0 = (10, 10)^T$.

I implemented BFGS with the provided line-search and SR1 with the provided trust region code. For repeatability The parameters were all as provided. But to be specific, for BFGS our starting point $x0 = (10, 10)$. Our alpha parameter for line search, $alpha0 = 1$. Our maximum iterations was 200 and tolerance on our step length was 1e-10. The 'c' optimisation parameters for strong Wolfe line search were 1e-4 and 0.5 respectively. For the SR1 Trust region implementation parameters were identical bar $eta = 0.1$ the acceptance threshold for our steps and the radius of the trust region $Delta = 1$.

Fig 1, shows the optimisation paths taken for BFGS line-search compared to the SR1-Trust region. Here we see quite different paths taken. BFGS took some very large initial steps including a very large initial step where we were yet to rescale and calculate our Hessian estimate reasonably , whereas the trust region method was more slow and steady but of course with steps limited to the parameters chosen for our trust region.

I have also included semi-log charts, fig 2 of the norm of the errors of each method by iteration. Both methods converged in a reasonably similar number of iterations with BFGS being slightly ahead due to its early 'jumps', the SR1 trust region method was slightly slower but with smooth progress.

5 Exercise 3b

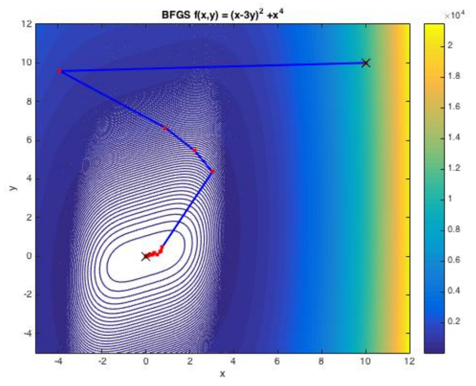
Plot the error of the sequences, $||I - H_k^{BFGS}\nabla^2 f(x_k)||_2$ and $||B_k^{SR1} - \nabla^2 f(x_k)||_2$, with $k \geq 0$. Explain.

Fig 3 shows semi-log plots of the errors based upon estimated Hessian matrices for the BFGS and the SR1 methods. Note that in SR1 we are directly calculating B and so subtract this from the Hessian at each point and take the norm. For BFGS we are taking the inverse Hessian and so multiply by the Hessian and subtract by the identity matrix.

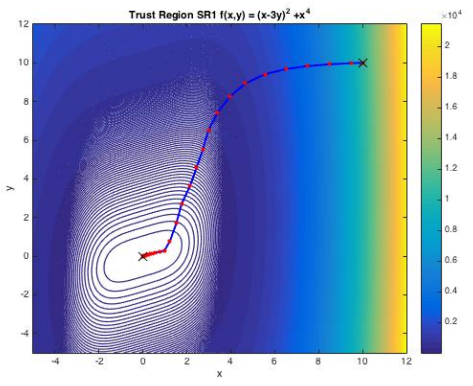
First of all for SR1 we know from Theorem 6.2 of Nocedal that

$$lim_{k\to\infty}||B_k^{SR1} - \nabla^2 f(x_k)||_2 = 0$$

Thus, I was actually expecting a smooth reduction in the norm to zero or close to it. There was a smooth reduction, but this flattened out close to a norm of 1. (It did go below 1, but progress was slow). Honestly this didn't tie in with my intuition. I thus looked at the later estimates for B_k and for the Hessian

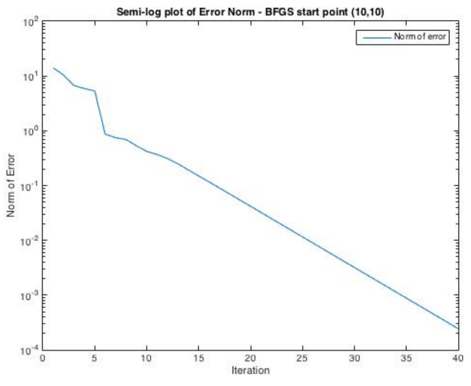


(a) BFGS Optimisation path

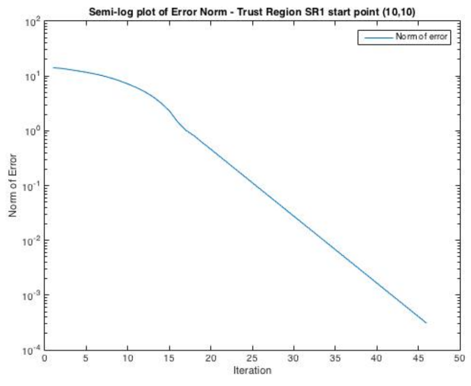


(b) TR-SR1 Optimisation path

Figure 1: BFGS v TR-SR1



(a) BFGS Norm of Errors

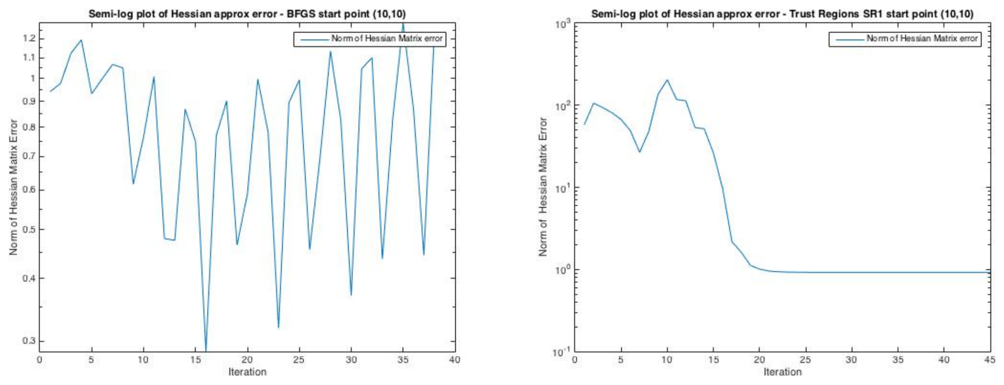


(b) TR-SR1 Norm of Errors

Figure 2: BFGS v TR-SR1 Errors by Iteration

for each x_k taking care to choose the correct B_k. The values were actually very close to the actual Hessian which itself had a norm close to 20 at the end. (The norm of the estimate to the Hessian by this point was 20.9), so perhaps I am being overly pessimistic about the flattening of the estimate error, and it simply means we achieved convergence (of course we are also constrained by the trust region in this case, and indeed we always update but may skip steps for SR1).

BFGS was also not really what I expected. The norm of the matrix estimate errors immediately reduced to one or below, but hopped around between about 0.3 and 1. My intuition again expected some sort of convergence. From my reading Nocedal, I see that BFGS does have self-correction properties when the approximation to the Hessian is poor and an adequate line search is performed. I understand the update to BFGS, and there is actually no reason why the inverse Hessian should actually exactly match each iteration or over several iterations as we go step by step. The key point may be that the error tends to self correct after a relatively poor estimate (where the error norm is close to 1 in this case), but of course as we step forward this can happen again.



(a) BFGS Norm of Hessian Matrix Errors (b) TR-SR1 Norm of Hessian Matrix Errors

Figure 3: BFGS v TR-SR1 Hessian Approximations