Report: Optimization Methods
INF 272 - Nonlinear Optimization

Mahmood Jokar

April 2020

Implementation of optimization methods in ADMB

Abstract

Automatic differentiation model builder (ADMB) is a framework for performing statistical parameter estimation, that includes a complete suite of tools for developing nonlinear mathematical and statistical models. The main aim of this report is to explore implemented algorithms and evaluate the performance of optimization tool employed in ADMB. In order to establish numerical comparison with ADMB, three algorithms have been implemented as line search directions which can be combined with three linesearch steps. A matlab toolbox function *fminunc* has also been used for the numerical comparisons. This report uses three different mathematical problems to investigate the efficiency of the optimization tools in ADMB, matlab toolbox and also my implemented algorithms.

1 Introduction

ADMB [1] is a programming framework based on automatic differentiation, aimed at highly nonlinear models with a large number of parameters. This framework integrates function minimizer which is used to minimize an objective function (e.g. a negative log-likelihood function). The general numerical function minimizer that has been used in the ADMB is *fmin(const double &f, const dvector &x, const dvector &g)* fuction where it's three input elements consist of

f as objective function value, x is a initial vector and g is a vector of gradient information. The function source code can be found in the Appendix. Function *fmin* contains Quasi-Newton function minimizer with inexact line search using Wolfe conditions and BFGS correction formula for Hessian update. The current report explores details of the algorithms for linesearch directions and linesearch step sizes used in the ADMB.

Algorithms for three search directions steepest descent, modified Newton and BFGS will be described and the corresponding matlab codes are implemented. Three linesearch steps backtracking Armijo, Wolfe conditions and strong Wolfe conditions are implemented to be combined with the mentioned linesearch directions. The report intend to compare the numerical results of the ADMB, implemented matlab algorithms and one of the optimization toolbox matlab which is called *fminunc*. To evaluate the performance of the mentioned optimization tools three mathematical problems with different dimensions have been considered. The fraction of number of function evaluation divided by iterations is used as a criterion of the efficiency of the linesearch procedures.

In order to create a consistency in the report and employed notations we first give a brief description of the methods to be discussed. All the definitions and the methods can be found in the reference [2]. First we start with the definition of an optimization problem. An optimization problem is a problem of the form

$$\min_{\text{s.t. } x\in\mathcal{R}^n,} f(x) \tag{1}$$

where f is a scalar-valued function called the objective function, x is the decision variable, and $\mathcal{R}^n$ is the constraint set (or feasible set). The abbreviations min. and s.t. are short for minimize and subject to respectively. The gradient of the function $f : \mathcal{R}^n \to \mathcal{R}$ (which is two times continuously differentiable) at a vector point x is a column vector

$$\nabla f = \left[\begin{array}{cccc} \frac{\partial f}{\partial x_1}, & \frac{\partial f}{\partial x_2}, & \cdots & \frac{\partial f}{\partial x_n} \end{array} \right]^T,$$

and the Hessian is a $n \times n$ square matrix bellow:

$$\nabla^2 f = \left[\begin{array}{cccc} \frac{\partial^2 f}{\partial x_1^2}, & \frac{\partial^2 f}{\partial x_1 \partial x_2}, & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}, & \frac{\partial^2 f}{\partial x_2^2}, & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots, & \vdots, & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}, & \frac{\partial^2 f}{\partial x_n \partial x_2}, & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{array} \right].$$

In this text we use notations $f_k = f(x_k)$, $g_k = \nabla f_k = \nabla f(x_k)$ and $H_k = \nabla^2 f(x_k)$ to indicate the values of the function, gradient and the Hessian of the function f at the point x_k, respectively.

At step k of the optimization procedure we use the terms p_k which stands for the search directions and α_k for the linesearch step. Using mentioned terms any

update in the sequence of optimization solution can be presented by:

$$x_{k+1} = x_k + \alpha_k p_k.$$

The necessary conditions of three linesearch updates that will be used in the implementations are as following. Interested reader is referred to [2] for more details of the algorithms.

- **Backtracking Armijo.** Given a point x_k and search direction p_k, we say that α_k satisfies the Armijo condition,

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \eta \alpha_k g_k^T p_k, \tag{2}$$

  for some $\eta \in (0, 1)$, e.g. $\eta = 0.1$ or even $\eta = 0.0001$. Performing backtracking Armijo linesearch includes starting with an initial α and substituting α by $\tau\alpha$ where $\tau \in (0, 1)$ until finding first α which satisfying the Armijo condition (2).

- **Wolfe conditions.** Given the current iterate x_k, search direction p_k, and constants $0 < c_1 < c_2 < 1$ we say that the step length α_k satisfies the Wolfe conditions if

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T p_k, \tag{3}$$
$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k. \tag{4}$$

- **Strong Wolfe conditions.**Given the current iterate x_k, search direction p_k, and constants $0 < c_1 < c_2 < 1$ we say that the step length α_k satisfies the Wolfe conditions if

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T p_k, \tag{5}$$
$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq c_2 |\nabla f(x_k)^T p_k|. \tag{6}$$

We use $||x||_2$ which stands for norm two of the n dimensional vector x by following formula to define the termination criterion for the implemented algorithms.

$$||x||_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

Termination criterion that has been used in the MATLAB implementations is as:

$$||\nabla f(x_k)||_2 \leq \text{tol} \ .\max(1, ||\nabla f(x_0)||_2), \tag{7}$$

where constant tol $\in (0, 1)$ is considered as optimization tolerance, e.g. 10^{-8}.

The rest of the manuscript is organized in the following way. Section 2 describes the linesearch methods and search direction algorithms implemented in

the ADMB. Implemented BFGS, modified Newton and steepest descent methods combined with different linesearchs have been presented in section 3. Numerical comparison of the ADMB, matlab toolbox and my implementation algorithms is given in section 4 and finally we end the report by a discussion of the results in section 5.

2 ADMB

Termination criterion that has been used in the ADMB is as following:

- **Successful termination**: The condition for successful termination in ADMB has been set as:

$$||\nabla f(x_k)||_2 \leq \text{tolerance}. \tag{8}$$

- **Unsuccessful termination**: Two events show the unsuccessful termination. First if there is no improvement in the function value during last 10 iterations and second if the number of function evaluations go larger than 100000.

2.1 Linesearches

Strong Wolfe condition has been used as linesearch criteria in the ADMB. Here we describe the algorithmic definition of the linesearch method used in *fmin* function of the ADMB. First we present the algorithm being used as backtracking procedure in the Algorithm 1 and then we give the Algorithm 2 for linesearch method.

2.2 Linesearch directions

BFGS updating linesearch direction has been used in *fmin* function of ADMB. Newton method with Hessian being an identity matrix is implemented for the initial approximation.

Before we discuss the details of the search direction algorithm being used we mention some notations and formula of the BFGS method following [2]. We

Algorithm 1 Backtracking

input α, x_i, α_i, p_i for $i = k-1, k$
Set $gs = p_{k-1}^T \nabla f(x_{k-1} + \alpha_{k-1} p_{k-1})$ and $gys = p_k^T \nabla f(x_k + \alpha_k p_k)$
Calculate $z = 3\frac{f(x_{k-1}) - f(x_k)}{\alpha} + gys + gs$
If $z^2 - gs \times gys > 0$
  Set $zz = \sqrt{z^2 - gs \times gys}$
else
  Set $zz = 0$
end
Compute $z = 1 - \frac{gys + zz - z}{2zz + gys - gs}$
If $|f(x_k) - 1.e + 95| < 1.e - 66$
  Set $\alpha = \alpha \times 0.001$
else
  If $z > 10$ print out 'large z' and $z = 10$
  Set $\alpha = \alpha \times z$
end
if $\alpha = 0$ print out 'step size is too small'
Return α

Algorithm 2 Linesearch step

Input x_k, p_k and initial constant fringe (default equals 0)
Start with Newton step size $\alpha_0 = 1$ and $l = 0$
Compute new step length $0 < \alpha_{l+1} <= 1$ by $\alpha_{l+1} = -2\frac{f(x_k)}{p_k^T \nabla f(x_k + \alpha_l p_k)}$
if $\alpha_{l+1} > 1$ then $\alpha_{l+1} = 1$
If $f(x_k + \alpha_{l+1} p_k) > f(x_k) +$ fringe
  Jump to backtracking step 13
else
If condition (6) with $c_2 = 0.9$,
  Jump to step 12
else if number of verification > 4 and condition (6) with $c_2 = 0.95$
  Jump to step 12
else
  backtracking step 13
end
end
Return α_l
Backtracking defined in Algorithm 1 and $l = l + 1$ and jump to step 5

define two vectors s_k (displacement) and y_k (the change of gradients) as bellow:

$$s_k = x_{k+1} - x_k = \alpha_k p_k, \tag{9}$$
$$y_k = \nabla f_{k+1} - \nabla f_k. \tag{10}$$

Using defined vectors and an approximate hessian matrix B_k, the BFGS updating formula is written as:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T. \tag{11}$$

where $\rho_k = \frac{1}{y_k^T s_k}$. The algorithmic definition of the ADMB search direction is presented in the Algorithm 3.

Algorithm 3 Search direction

Input starting point x_0, convergence tolerance ϵ
Initialize H_0 to the unit matrix
while $||\nabla f_k||_2 > \epsilon$
  $p_k = -H_k^{-1} \nabla f_k$
  Calculate step length α_k from Algorithm 2
  Set $x_{k+1} = x_k + \alpha_k p_k$
  Calculate s_k and y_k by equations (9) and (10)
  If $k = 0$
    If $p_k^T y_k + s_k^T \nabla f_k > 0$
      Set $z = \frac{s_k^T y_k}{p_k^T y_k + s_k^T \nabla f_k} - 1$
      Set $p_k = z \nabla f_k + \nabla f_{k+1}$
      Set $\rho_k = \frac{1}{s_k^T y_k} - \frac{s_k^T \nabla f_k}{(p_k^T y_k)^2}$
      Jump to step 22
    end
    Set $\rho_k = \frac{1}{s_k^T y_k}$
  end
  If $p_k^T y_k + s_k^T \nabla f_k > 0$
    Set $\rho_k = \frac{-\alpha_k}{p_k^T y_k - s_k \nabla f_k}$
    Jump to step 22
  end
  Set $\rho_k = \frac{1}{p_k^T y_k}$
  Compute H_{k+1} using equation (11)
  k=k+1
end
Return x_k

3 Implemented algorithms

In this section we implemented three methods BFGS (used in ADMB), modified Newton and Steepest descent in Matlab with different combinations of Backtracking Armijo, Wolfe condition and Strong wolfe conditions as linesearch updating. For each of the methods we first specify the algorithm being used.

First we give the algorithm of implemented Steepest descent method as:

Algorithm 4 Steepest descent

1: **Input** initial x_0
2: $k = 0$
3: **Loop**
4: Set $p_k = -g_k$
5: Calculate α_k by one of the linesearch method 1) Backtracking Armijo, 2) Wolfe condition, 3) Strong Wolfe conditions.
6: $x_{k+1} = x_k + \alpha_k p_k$
7: **End loop**
8: **Return** approximate solution x_k

The Matlab code for Steepest descent algorithm is written as bellow:

```
function [xstar, fval,z,a,f_cont,i] = ...
    Steepest_descent(f,x,tol,iter,method)
% Steepest descent algorithm
fprintf('Minimization using Steepest Descent algorithm: \n');
x0 = x;
z = x';
alpha0 = 0.002;          % Steepest decsent
c1 = 0.0001;
c2 = 0.1;                % Steepest decsent
a = alpha0;
f_cont = 0;
fval = [];
fprintf('%12s %12s %8s ...
    %16s\n','Iteration','Func-cont','f(x)','Step-size');
for i = 1:iter
    g = grad(x);
    if method == 1
        [alpha, f_iter] = Backtracking_Armijo(f,g,x,-g,alpha0);
    elseif method == 2
        [alpha, f_iter] = Wolfe_condition(f,x,-g,alpha0,c1,c2);
    else
        [alpha, f_iter] = Strong_Wolfe(f,x,-g,alpha0,c1,c2);
    end
    f_cont = f_cont+f_iter;
```

```
23      a = [a,alpha];
24      xstar = x - alpha*g;
25      z = [z;xstar'];
26      fval = [fval,f(xstar)];
27      gnorm = norm(grad(xstar));
28      if ¬isfinite(xstar)
29          fprintf(1,'Number of iterations: %d\n', i);
30          error('x is inf or NaN')
31      end
32      if gnorm < tol*max(1,norm(grad(x0)))
33          fprintf(1,'Convergance criterion: %d\n', tol);
34          fprintf(1,'Maximum gradient component: %d\n', gnorm);
35          fprintf(1,'Number of iterations: %d\n', i-1);
36          break;
37      end
38      x = xstar;
39      fprintf('%10d %10d %14.2e %14.2e\n',i,f_iter,fval(i),alpha);
40  end
41  if i == iter
42      fprintf(1,'Number of iterations is: %d\n', iter);
43      fprintf(1,'Convergance criterion: %d\n', tol);
44      fprintf(1,'Maximum gradient component: %d\n', gnorm);
45  end
```

Next we give the algorithm of implemented Modified Newton method as bellow:

Algorithm 5 Modified-Newton

1: **Input** x_0 and H and $\alpha_0 = 1.0$
2: Select $\beta > 1$
3: $k = 0$
4: **Loop**
5: Compute spectral decomposition $H = VDV^T$
6: Set $\epsilon = 1$ (if $H = 0$) or $\epsilon = ||H||_2/\beta$ (o.w)
7: Modify eigenvalues in matrix D as matrix $\bar{D}$ with diagonal eateries $> \epsilon$
8: Compute $\bar{H} = V\bar{D}V^T$
9: Solve $\bar{H}p_k = -g_k$
10: Calculate α_k by one of the linesearch method 1) Backtracking Armijo, 2) Wolfe condition, 3) Strong Wolfe conditions.
11: $x_{k+1} = x_k + \alpha_k p_k$
12: **End loop**
13: **Return** approximate solution x_k

Following code is a function that implements Modified Newton algorithm:

```
1  function [xstar, fval,z,a,f_cont,i] = ...
       Modified_Newton(f,x,tol,iter,method)
2  % Modified Newton algorithm
```

```
fprintf('Minimization using Modified Newton algorithm: \n');
x0 = x;
nvar = length(x);
beta = 1e4;
z = x';
alpha0 = 1.0;           % Newton
c1 = 0.0001;
c2 = 0.9;               % Newton, BFGS
a = alpha0;
f_cont = 0;
fval = [];
fprintf('%12s %12s %8s ...
    %16s\n','Iteration','Func-cont','f(x)','Step-size');
for i = 1:iter
    B = Hessian(x);
    [V,D] = eig(B);
    if nnz(B)== 0
        epsilon = 1;
    else
        epsilon = norm(B)/beta;
    end
    for j=1:nvar
        if D(j,j) ≥ epsilon
            D(j,j) = D(j,j);
        elseif D(j,j) ≤ -epsilon
            D(j,j) = -D(j,j);
        else
            D(j,j) = epsilon;
        end
    end
    B = V*D*V';
    g = grad(x);
    p = -inv(B)*g;
    if method == 1
        [alpha, f_iter] = Backtracking_Armijo(f,g,x,p,alpha0);
    elseif method == 2
        [alpha, f_iter] = Wolfe_condition(f,x,p,alpha0,c1,c2);
    else
        [alpha, f_iter] = Strong_Wolfe(f,x,p,alpha0,c1,c2);
    end
    f_cont = f_cont+f_iter;
    a = [a,alpha];
    xstar = x + alpha*p;
    z = [z;xstar'];
    g = grad(xstar);
    gnorm = norm(g);
    fval = [fval,f(xstar)];
    fprintf('%10d %10d %14.2e %14.2e\n',i,f_iter,fval(i),alpha);
    if ¬isfinite(xstar)
        fprintf(1,'Number of iterations: %d\n', i);
        error('x is inf or NaN')
    end
    if gnorm < tol*max(1,norm(grad(x0)))
        fprintf(1,'Convergance criterion: %d\n', tol);
        fprintf(1,'Maximum gradient component: %d\n', gnorm);
        fprintf(1,'Number of iterations: %d\n', i);
        break;
```

```
    end
    x = xstar;
end
if i == iter
    fprintf(1,'Number of iterations is: %d\n', iter);
    fprintf(1,'Convergance criterion: %d\n', tol);
    fprintf(1,'Maximum gradient component: %d\n', gnorm);
end
```

Finally we give the algorithm for BFGS method as:

Algorithm 6 BFGS

1: **Input** initial guess x_0 and $\alpha_0 = 0.007$ and $\alpha_1 = 1.0$
2: Initial Hessian (B) as identity matrix
3: $k = 0$
4: **Loop**
5: Solve $Bp_k = -g_k$
6: Calculate α_k by one of the linesearch method 1) Backtracking Armijo, 2) Wolfe condition, 3) Strong Wolfe conditions.
7: $x_{k+1} = x_k + \alpha_k p_k$
8: calculate $s = x_{k+1} - x_k$ and $y = g_{k+1} - g_k$
9: Update $B = B - \frac{1}{s^T Bs}(Bs)(Bs)^T + \frac{1}{y^T s}(yy^T)$
10: **End loop**
11: **Return** approximate solution x_k

BFGS codes is given in the following matlab function:

```
function [xstar, fval,z,a,f_cont,i] = BFGS(f,x,tol,iter,method)
% BFGS Quasi-Newton algorithm
fprintf('Minimization using BFGS Quasi-Newton algorithm: \n');
m = 1;
x0 = x;
z = x';
alpha = 0.007;
a = alpha;
f_cont = 0;
fval = f(x);
for i = 1:m
    g = grad(x);
    gnorm = norm(g);
    xstar = x - alpha*g;
    z = [z;xstar'];
    %fval = [fval, f(xstar)];
    if ¬isfinite(xstar)
        fprintf(1,'Number of iterations: %d\n', i);
        error('x is inf or NaN')
    end
    if gnorm < tol*max(1,norm(grad(x0)))
        fprintf(1,'Gradient is less than: %d\n', tol);
```

```
        fprintf(1,'Number of iterations: %d\n', i);
        break;
    end
    xold = x;
    x = xstar;
end
fprintf('%12s %12s %8s ...
    %16s\n','Iteration','Func-cont','f(x)','Step-size');
fprintf('%10d %10d %14.2e %14.2e\n',1,1,fval(i),alpha);
alpha0 = 0.9;           % BFGS
c1 = 0.0001;
c2 = 0.9;               % Newton, BFGS
xnew = xstar;
nvar = length(x);
B = eye(nvar);
g = grad(xnew);
for i = 1:iter
    y = grad(xnew) - grad(xold);
    s = xnew - xold;
    B = B - (1/(s'*B*s))*(B*s)*(B*s)' + (1/(y'*s))*(y*y');
    p = -inv(B)*g;
    if method == 1
        [alpha, f_iter] = Backtracking_Armijo(f,g,xnew,p,alpha0);
    elseif method == 2
        [alpha, f_iter] = Wolfe_condition(f,xnew,p,alpha0,c1,c2);
    else
        [alpha, f_iter] = Strong_Wolfe(f,xnew,p,alpha0,c1,c2);
    end
    f_cont = f_cont+f_iter;
    a = [a,alpha];
    xstar = xnew + alpha*p;
    z = [z;xstar'];
    g = grad(xstar);
    gnorm = norm(g);
    fval = [fval,f(xstar)];
    fprintf('%10d %10d %14.2e %14.2e\n',i+m,f_iter,f(xstar),alpha);
    if ~isfinite(xstar)
        fprintf(1,'Number of iterations: %d\n', i);
        error('x is inf or NaN')
    end
    if gnorm < tol*max(1,norm(grad(x0)))
        fprintf(1,'Convergance criterion: %d\n', tol);
        fprintf(1,'Maximum gradient component: %d\n', gnorm);
        fprintf(1,'Number of iterations: %d\n', i);
        break;
    end
    xold = xnew;
    xnew = xstar;
end
if i == iter
    fprintf(1,'Number of iterations is: %d\n', iter);
    fprintf(1,'Convergance criterion: %d\n', tol);
    fprintf(1,'Maximum gradient component: %d\n', gnorm);
end
```

We Algorithm and matlab code of the Backtracking Armijo linesearch method as following:

Algorithm 7 Backtracking Armijo

1: **Input** x_k and p_k
2: Set iter $= 1e2$, $\eta = 0.1$, $\tau = 0.5$ and $l = 1$
3: **until** condition (2)
4: **if** $l >$ iter **then** $\alpha_l = 0.002$ and break
5: $\alpha_{l+1} = \tau\alpha_l$
6: l=l+1
7: **End until**
8: **Return** α_l

```
function [alpha, l] = Backtracking_Armijo(f,g,x,p,alpha)
maxiter = 1e2;
tau = 0.5;
eta = 0.1;
l = 1;
while f(x+alpha*p) > f(x)+eta*alpha*g'*p
    alpha = tau*alpha;
    l = l+1;
    if l>maxiter
        alpha = 0.002;
        break;
    end
end
if l≠1
    l = l-1;
   alpha = alpha/tau;
end
```

The Wolfe condition linesearch method is given in the following Algorith and matlab function:

Algorithm 8 Wolfe conditions

1: **Input** x_k, p_k, c_1 and c_2
2: Set iter $= 1e2$, $\tau = 0.5$ and $l = 1$
3: **until** conditions (3) and (4)
4: **if** $l >$ iter **then** $\alpha_l = 0.0005$ and break
5: $\alpha_{l+1} = \tau\alpha_l$
6: l=l+1
7: **End until**
8: **Return** α_l

```
function [alpha, l] = Wolfe_condition(f,x,p,alpha,c1,c2)
maxiter = 1e2;
```

```
tau = 0.5;
l = 1;
while 1
    if f(x+alpha*p) ≤ f(x)+c1*alpha*grad(x)'*p && ...
        grad(x+alpha*p)'*p ≥ c2*grad(x)'*p
        break;
    end
    alpha = tau*alpha;
    l = l+1;
    if l>maxiter
        alpha = 0.0005;
        break;
    end
end
if l≠1
    l = l−1;
   alpha = alpha/tau;
end
```

Bellow Algorithm and matlab function present Strong Wolfe conditions:

Algorithm 9 Strong Wolfe conditions

1: **Input** x_k, p_k, c_1 and c_2
2: Set iter $= 1e2$, $\tau = 0.5$ and $l = 1$
3: **until** conditions (5) and (6)
4: **if** $l >$ iter **then** $\alpha_l = 0.0005$ and break
5: $\alpha_{l+1} = \tau\alpha_l$
6: l=l+1
7: **End until**
8: **Return** α_l

```
function [alpha, l] = Strong_Wolfe(f,x,p,alpha,c1,c2)
maxiter = 1e2;
tau = 0.5;
l = 1;
while 1
    if f(x+alpha*p) ≤ f(x)+c1*alpha*grad(x)'*p && ...
        abs(grad(x+alpha*p)'*p) ≤ c2*abs(grad(x)'*p)
        break;
    end
    alpha = tau*alpha;
    l = l+1;
    if l>maxiter
        alpha = 0.0005;
        break;
    end
end
if l≠1
    l = l−1;
   alpha = alpha/tau;
end
```

4 Numerical comparisons

In this section we compare the ADMB, matlab toolbox function *fminunc* and my matlab implementation of BFGS, steepest descent and modified Newton on the following problems [3]:

- Rosenbrock function,
- Osborne 1 function,
- Discrete boundary value problem with $n = 100$.

4.1 Rosenbrock function

Mathematical discretion of the Rosenbrock function $f : \mathcal{R}^2 \to \mathcal{R}$ is given [3] as:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{12}$$

The standard starting point for the optimization procedure is $(-1.2, 1)$ and the minimum value $f = 0$ is achieved at point $(1, 1)$.

ADMB Implementation. The ADMB code for minimization of the Rosenbrock function which has been implemented in the AD studio platform is presented as bellow:

```
DATA_SECTION
PARAMETER_SECTION
  init_number x1;
  init_number x2;
  objective_function_value f
INITIALIZATION_SECTION
  x1 -1.2
  x2 1
PROCEDURE_SECTION
  f = 100*pow((x2-pow(x1,2)),2)+pow((1 - x1),2);
RUNTIME_SECTION
  maximum_function_evaluations 20000
  convergence_criteria 1.e-8
```

Result of running the ADMB code above is given as:

```
 - final statistics:
2 variables; iteration 38; function evaluation 50
Function value  5.5346e-021; maximum gradient component mag  2.7524e-009
Exit code = 1;  converg criter  1.0000e-008
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  1.00000  2.7524e-009 |  2  1.00000 -1.3443e-009 |
Estimating row 1 out of 2 for hessian
Estimating row 2 out of 2 for hessian

Process Rosenbrock finished
```

Matlab Implementation. In order to write a matlab tool to minimize the Rosenbrock function we present the matlab functions of the Rosenbrock and it's gradient vector and also Hessian matrix:

```
function f = Rosenbrock(x)
%  Rosenbrock function
f = 100*(x(2) - x(1)^2 )^2 + (1 - x(1))^2;
```

```
function g = grad(x)
%  Rosenbrock gradient
g = [2*x(1) - 400*x(1)*(- x(1)^2 + x(2)) - 2; - 200*x(1)^2 + ...
     200*x(2)];
```

```
function h = Hessian(x)
%  Rosenbrock hessian
h = 200*[6*x(1)^2 - 2*x(2) + 0.01, - 2*x(1) ; - 2*x(1), 1];
```

The main function to implement any combination of the search directions (BFGS, Modified-Newton, steepest descent) and linesearch criteria (Wolfe, strong-wolfe and Armijo conditions) algorithms for optimization of the Rosenbrock function is as follow:

```
clc; clear;
%%  Initial variables
x = [-1.2; 1];

% select a linesearch methods:
method = 1; %      1: Backtracking_Armijo
            %      2: Wolfe condition
            %      3: Strong Wolfe conditions

%% optimization
tol = 10^(-8);
iter = 20000;
```

```
13
14 % Active one of the following linesearch direction algorithms:
15 %%%%%% Steepest descent
16 % [Epar,fval,z,alpha,f_cont,iter] = ...
       Steepest_descent(@Rosenbrock,x,tol,iter,method);
17 %%%%%% Modified Newton
18 [Epar,fval,z,alpha,f_cont,iter] = ...
       Modified_Newton(@Rosenbrock,x,tol,iter,method);
19 % %%%%% BFGS
20 % [Epar,fval,z,alpha,f_cont,iter] = ...
       BFGS(@Rosenbrock,x,tol,iter,method);
21
22 %% result
23 fprintf(1,'Function evaluation: %d\n', f_cont);
24 fprintf(1,'fval: %e\n', fval(end));
25 fprintf(1,'Estimated x1 is: %d\n', Epar(1));
26 fprintf(1,'Estimated x2 is: %d\n', Epar(2));
27
28 %% Illustration
29 syms x1 x2
30 f = 100*(x2 - x1^2 )^2 + (1 - x1)^2;
31 fcontour(f,[-1.2 1.2 -1.5 1.5])
32 hold on
33 figure(1), plot(z(:,1),z(:,2),'r*-')
34 xlabel('x1'); ylabel('x2'); box off
35 figure(2), plot(1:length(alpha),alpha,'LineWidth',1.5)
36 xlabel('Iteration');  ylabel('alpha'); ylim([0,1.2]); box off
37 figure(3), plot(1:length(fval),log(abs(fval)),'LineWidth',1.5)
38 xlabel('Iteration');  ylabel('log(f(x))'); box off
39
40 %% fminunc
41 fprintf('Minimization MATLAB fminunc with Quasi-Newton ...
       algorithm: \n');
42 history= histclass;
43 outf= @(x,optimValues,state)outfun(x,optimValues,state,history);
44 options = optimoptions(@fminunc,'Display','iter','Algorithm',...
45           'quasi-newton','HessUpdate','bfgs','OutputFcn',outf,...
46           'TolFun',1e-8,'MaxFunEvals',2e4,'MaxIter',2e4);
47 xfmin = fminunc(@Rosenbrock,x,options);
48 fprintf('Number of function evaluations: %d\n', history.count);
49 hold on
50 plot(1:length(history.fval),log(history.fval),'LineWidth',1.5)
51
52 %% ADMB
53 dat = 'ADMB.txt';  % fun values
54 z = load(dat);
55 plot(1:length(z),log(z),'LineWidth',1.5)
56 legend('My result','fminunc','ADMB')
57
58 %%
59 %close all;
```

BFGS algorithm. The results of BFGS method using strong wolfe condition linesearch updating algorithm are presented as bellow. It should be pointed out that the BFGS algorithm with other linesearches like backtracking Armijo and wolfe conditin produces same results. Considered parameters for this method are as:

- $\alpha = 0.9$
- $c_1 = 0.0001$
- $c_2 = 0.9$
- $\tau = 0.5$
- Convergence tol. : 10^{-8}
- Maximum iteration : 20000
- Initial values : $(-1.2, 1)$

Result of running the Matlab codes:

```
Minimization using BFGS Quasi-Newton algorithm:
Convergance criterion: 1.000000e-08
Maximum gradient component: 8.734266e-07
Number of iterations: 26
Function evaluation: 35
fval: 4.943083e-16
Estimated x1 is: 1.000000e+00
Estimated x2 is: 1.000000e+00
```

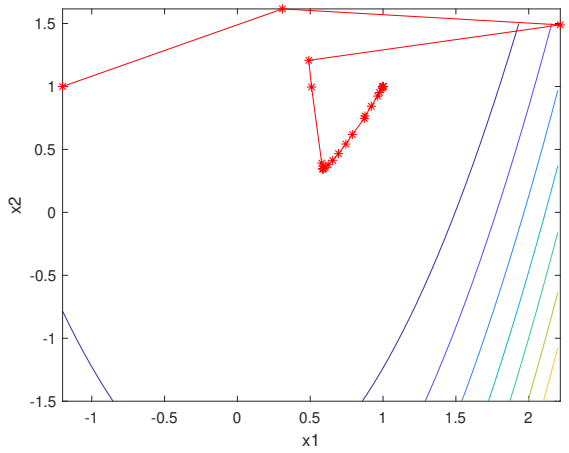

Figure 1: BFGS algorithm with Strong wolfe condition linesearch method.

Modified Newton. The result of Modified Newton method using strong wolfe linesearch updating algorithms are presented as bellow. Combination of the Modified Newton algorithm with other linesearches like backtracking Armijo and wolfe conditions produces same results. Considered parameters for this method are as:

- $\alpha = 1.0$
- $c_1 = 0.0001$
- $c_2 = 0.9$
- $\tau = 0.5$
- Convergence tol. : 10^{-8}
- Maximum iteration : 20000
- Initial values : $(-1.2, 1)$

Result of running the Matlab codes:

```
Minimization using Modified Newton algorithm:
Convergance criterion: 1.000000e-08
Maximum gradient component: 8.349152e-10
Number of iterations: 16
Function evaluation: 23
fval: 3.486832e-22
Estimated x1 is: 1.000000e+00
Estimated x2 is: 1.000000e+00
```

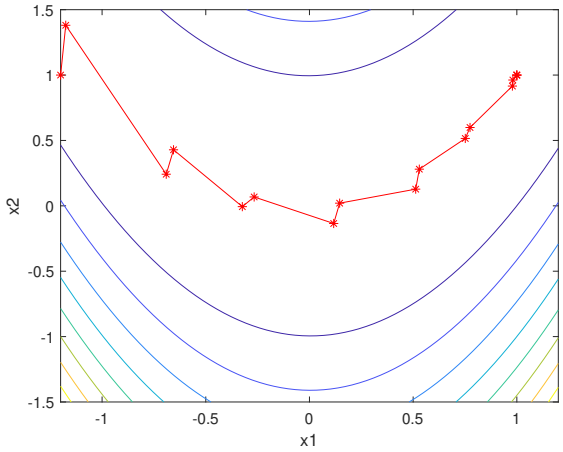

Figure 2: Modified Newton algorithm with strong wolfe condition linesearch method.

Steepest descent. The result of steepest descent method using three Backtracking Armijo linesearch updating algorithm are presented as bellow. Combination of the steepest descent algorithm with other linesearches like backtracking wolfe condition and strong wolfe conditions produces very similar results. Considered parameters for this method are as:

- $\eta = 0.1$
- $\tau = 0.5$
- Convergence tol. : 10^{-8}
- Maximum iteration : 20000
- Initial values : $(-1.2, 1)$

Result of running the Matlab codes:

```
Minimization using Steepest Descent algorithm:
Number of iterations is: 20000
Convergence criterion: 1.000000e-08
Maximum gradient component: 1.211342e+00
Function evaluation: 20000
fval: 7.325490e-04
Estimated x1 is: 1.000380e+00
Estimated x2 is: 9.980546e-01
```

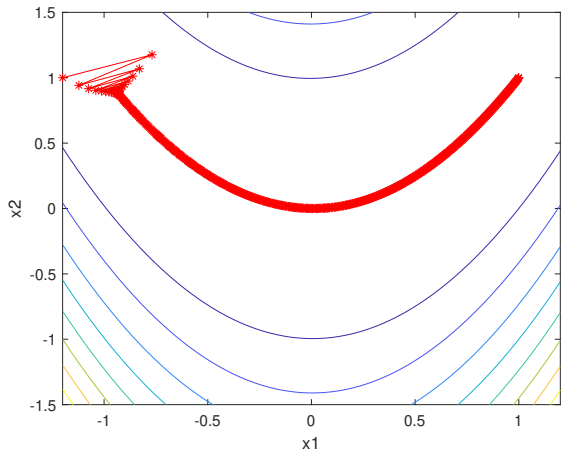

Figure 3: Steepest descent algorithm with backtracking Armijo linesearch method.

***fminunc* optimization tool.** Here, we present the result of the *fmininc* function of matlab optimization tool. BFGS method has been selected as linesearch in the command options in order to have a similar method with ADMB.

```
Minimization MATLAB fminunc with Quasi-Newton algorithm:

                                                         First-order
 Iteration  Func-count       f(x)        Step-size       optimality
     0           3             24.2                          216
     1           9          4.28049    0.000861873           15.2
     2          12          4.12869              1              3
     3          15          4.12069              1            1.5
     4          18           4.1173              1           1.62
     5          21          4.08429              1           5.72
     6          24          4.02491              1           10.4
     7          27           3.9034              1           17.4
     8          30           3.7588              1           20.1
     9          33          3.41694              1           19.9
    10          36          2.88624              1           11.9
    11          39           2.4428              1           9.78
    12          42          1.93707              1           3.01
    13          51          1.64358       0.141993           5.54
    14          54          1.52561              1           7.57
    15          57          1.17013              1           4.53
    16          60         0.940886              1           3.17
    17          63         0.719811              1           5.15
    18          66         0.409467              1           5.74
    19          75         0.259847      0.0842987           5.02
    20          78         0.239037              1           1.07
    21          81         0.210255              1           1.23
    22          87         0.182742       0.587153           3.33
    23          90         0.158905              1           2.91
    24          93        0.0895318              1          0.763
    25          99        0.0727862       0.440669           3.19
    26         102        0.0413467              1            2.3
    27         105        0.0222054              1          0.497
    28         111        0.0126069       0.406355           1.98
    29         114       0.00702839              1           1.35
    30         117       0.00203069              1          0.191
    31         123       0.00108975            0.5          0.877
    32         126      9.14432e-05              1          0.143
    33         129      7.27182e-06              1         0.0807
    34         132      4.44847e-07              1          0.022
    35         135      1.47502e-08              1        0.00272
    36         138      2.83357e-11              1       1.91e-05
    37         141      2.00651e-11              1       3.56e-08

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>
Number of function evaluations: 141
```

Table 1: Result of different optimization tools for the Rosenbrock function

method	iteration	fun evaluation	efficiency	fun value
ADMB	38	50	0.76	5.53e-21
fminunc	37	141	0.26	2.01e-11
my result(BFGS)	26	35	0.74	4.94e-16

We compare the efficiency of the employed methods for optimizing Rosenbrock function by calculating the fraction of number of iteration over number of function evaluations. Table 1 present the results. Figure 4 gives the comparison of ADMB, fminunc and my implemented codes. Step size α in the implemented BFGS algorithm using strong wolfe condition is plotted in the figure 5.

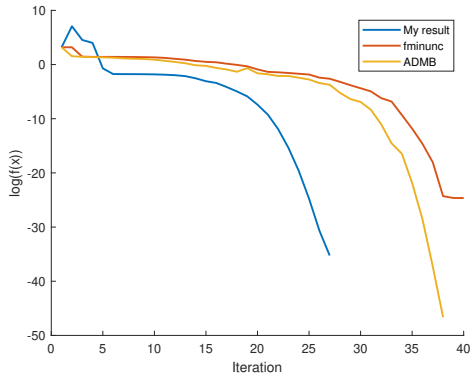

Figure 4: Logarithm of the function value at each iteration of the different optimization tools

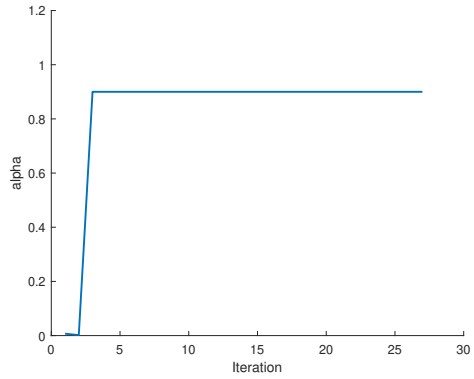

Figure 5: Step size α at each iteration for the implemented BFGS method

4.2 Osborne 1 function

The Osborne 1 is function $g : \mathcal{R}^5 \to \mathcal{R}$ where the description is as follow:

$$g(x_1, ..., x_5) = \sum_{i=1}^{33} [y_i - (x_1 + x_2 e^{-t_i x_4} + x_3 e^{-t_i x_5})]^2, \tag{13}$$

where $t_i = 10(i-1)$ and the values of the y_i is given in [3]. The standard starting point for the optimization procedure is $(0.5, 1.5, -1, 0.01, 0.02)$ and the function minimum value is $g = 5.46489 \times 10^{-5}$.

ADMB Implementation. The ADMB [3, 1, 2] code for minimization of the Osborne 1 function which has been implemented in the AD studio platform is presented as bellow:

```
DATA_SECTION
  init_number n
  init_vector y(1,n)
PARAMETER_SECTION
  init_number x1;
  init_number x2;
  init_number x3;
  init_number x4;
  init_number x5;
  objective_function_value f
  //f = 5.46489...e-5
INITIALIZATION_SECTION
  x1 0.5
  x2 1.5
  x3 -1
  x4 0.01
  x5 0.02
PROCEDURE_SECTION
  f = 0;
  for(int i=1;i<=n;i++){
    double t = 10*(i-1);
    f = f+pow((y[i]-(x1+x2*exp(-t*x4)+x3*exp(-t*x5))),2);
  }
PRELIMINARY_CALCS_SECTION
  cout << "y  = " <<y<< endl;
RUNTIME_SECTION
  maximum_function_evaluations 20000
  convergence_criteria 1.e-6
```

Results of the above ADMB code is given as figure 6. Code 1 in the final line of the ADMB result indicate that ADMB code with general minimizer was not able to converge. Therefore we try another minimization option *-lmn N* which stands for limited memory quasi Newton minimization. N is the number of pass steps of information kept for the quasi-Newton update. We give the result for $N = 5$ and $N = 20$ in the figures 7 and 8, respectively.

```
 - final statistics:
5 variables; iteration 2865; function evaluation 3728
Function value  4.6697e-002; maximum gradient component mag  1.9867e+006
Exit code = 1;  converg criter  1.0000e-006
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  16664045.04  3.3702e-004 |  221431527.16  3.3694e-004 |  3-28095571.22  3.3696e-004
  4  0.00000 -1.5154e+006 |  5  0.00000  1.9867e+006 |
Estimating row 1 out of 5 for hessian
Estimating row 2 out of 5 for hessian
Estimating row 3 out of 5 for hessian
Estimating row 4 out of 5 for hessian
Estimating row 5 out of 5 for hessian
Warning -- Hessian does not appear to be positive definite
Hessian does not appear to be positive definite

Process Osborne exited abnormally with code 1
```

Figure 6: Unsuccessful termination of the ADMB

```
final statistics: 5 variables; iteration 112; function evaluation 149
Function value  5.4649e-005; maximum gradient component mag  5.0152e-006
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  0.37541  4.2946e-007 |  2  1.93583 -8.6623e-008 |  3 -1.46467 -1.3211e-007
  4  0.01287 -1.5348e-005 |  5  0.02212  5.0152e-006 |
Estimating row 1 out of 5 for hessian
Estimating row 2 out of 5 for hessian
Estimating row 3 out of 5 for hessian
Estimating row 4 out of 5 for hessian
Estimating row 5 out of 5 for hessian

Process Osborne -lmn 5 finished
```

Figure 7: ADMB successful termination with N=5

```
final statistics: 5 variables; iteration 98; function evaluation 116
Function value  5.4649e-005; maximum gradient component mag  1.7716e-006
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  0.37541  1.1793e-007 |  2  1.93585  1.0805e-007 |  3 -1.46469  1.0080e-007
  4  0.01287 -1.6675e-007 |  5  0.02212  1.7716e-006 |
Estimating row 1 out of 5 for hessian
Estimating row 2 out of 5 for hessian
Estimating row 3 out of 5 for hessian
Estimating row 4 out of 5 for hessian
Estimating row 5 out of 5 for hessian

Process Osborne -lmn 20 finished
```

Figure 8: ADMB successful termination with N=20

Matlab Implementation. In order to write a matlab tool to minimize the Osborne 1 function we present the matlab functions of the Osborne 1 and it's gradient vector and also Hessian matrix:

```
function f = Osborne(x)
%  Osborne function
data = 'Osborne.txt';  % y_i values
y = load(data);
n = length(y);
f = 0;
for i=1:n
    t = 10*(i-1);
    f = f +(y(i)-x(1)-x(2)*exp(-t*x(4))-x(3)*exp(-t*x(5))).^2;
end
```

```
function g = grad(x)
%  Osborne gradient
g = zeros(5,1);
data = 'Osborne.txt';
y = load(data);
n = length(y);
%
f = 0;
for i=1:n
    t = 10*(i-1);
    f = f-2*(y(i)-x(1)-x(2)*exp(-t*x(4))-x(3)*exp(-t*x(5)));
end
g(1) = f;
%
f = 0;
for i=1:n
    t = 10*(i-1);
    f = f-2*exp(-t*x(4))*(y(i)-x(1)-x(2)*exp(-t*x(4))...
        -x(3)*exp(-t*x(5)));
end
g(2) = f;
%
f = 0;
for i=1:n
    t = 10*(i-1);
    f = f-2*exp(-t*x(5))*(y(i)-x(1)-x(2)*exp(-t*x(4))...
        -x(3)*exp(-t*x(5)));
end
g(3) = f;
%
f = 0;
for i=1:n
    t = 10*(i-1);
    f = f+2*(t*x(2)*exp(-t*x(4)))*(y(i)-x(1)-x(2)...
        *exp(-t*x(4))-x(3)*exp(-t*x(5)));
end
g(4) = f;
%
f = 0;
```

```
for i=1:n
    t = 10*(i-1);
    f = f+2*(t*x(3)*exp(-t*x(5)))*(y(i)-x(1)-x(2)...
        *exp(-t*x(4))-x(3)*exp(-t*x(5)));
end
g(5) = f;
```

```
function h = Hessian(x)
%  Osborne hessian
data = 'Osborne.txt';
y = load(data);
n = length(y);

t = zeros(1,n);
for i= 1:n
    t(i) = 10*(i-1);
end
h = zeros(5,5);

%% First row
h(1,1) = 66;
s = 0;
for i= 1:n
    s = s+exp(-t(i)*x(4));
end
h(1,2) = 2*s;
s = 0;
for i= 1:n
    s = s+exp(-t(i)*x(5));
end
h(1,3) = 2*s;
s = 0;
for i= 1:n
    s = s+(t(i)*x(2))*exp(-t(i)*x(4));
end
h(1,4) = -2*s;
s = 0;
for i= 1:n
    s = s+(t(i)*x(3))*exp(-t(i)*x(5));
end
h(1,5) = -2*s;

%% Second row
s = 0;
for i= 1:n
    s = s+exp(-t(i)*x(4));
end
h(2,1) = 2*s;
s = 0;
for i= 1:n
    s = s+exp(-2*t(i)*x(4));
end
h(2,2) = 2*s;
s = 0;
for i= 1:n
    s = s+exp(-t(i)*(x(4)+x(5)));
```

```
50  end
51  h(2,3) = 2*s;
52  s = 0;
53  for i= 1:n
54      s = s+t(i)*exp(-t(i)*x(4))*(y(i)-x(1)-x(2)...
55          *exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
56          -t(i)*x(2)*exp(-2*t(i)*x(4));
57  end
58  h(2,4) = 2*s;
59  s = 0;
60  for i= 1:n
61      s = s+t(i)*x(3)*exp(-t(i)*(x(4)+x(5)));
62  end
63  h(2,5) = -2*s;
64
65  %% Third row
66  s = 0;
67  for i= 1:n
68      s = s+exp(-t(i)*x(5));
69  end
70  h(3,1) = 2*s;
71  s = 0;
72  for i= 1:n
73      s = s+exp(-t(i)*(x(4)+x(5)));
74  end
75  h(3,2) = 2*s;
76  s = 0;
77  for i= 1:n
78      s = s+exp(-2*t(i)*x(5));
79  end
80  h(3,3) = 2*s;
81  s = 0;
82  for i= 1:n
83      s = s+t(i)*x(2)*exp(-t(i)*(x(4)+x(5)));
84  end
85  h(3,4) = -2*s;
86  s = 0;
87  for i= 1:n
88      s = s+t(i)*exp(-t(i)*x(5))*(y(i)-x(1)-x(2)...
89          *exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
90          -t(i)*x(3)*exp(-2*t(i)*x(5));
91  end
92  h(3,5) = 2*s;
93
94  %% Forth row
95  s = 0;
96  for i= 1:n
97      s = s+(t(i)*x(2))*exp(-t(i)*x(4));
98  end
99  h(4,1) = -2*s;
100 s = 0;
101 for i= 1:n
102     s = s+t(i)*exp(-t(i)*x(4))*(y(i)-x(1)-x(2)...
103         *exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
104         -t(i)*x(2)*exp(-2*t(i)*x(4));
105 end
106 h(4,2) = 2*s;
```

```
107  s = 0;
108  for i= 1:n
109      s = s+t(i)*x(2)*exp(-t(i)*(x(4)+x(5)));
110  end
111  h(4,3) = -2*s;
112  s = 0;
113  for i= 1:n
114      s = s+(-t(i)^2)*x(2)*exp(-t(i)*x(4))*(y(i)...
115          -x(1)-x(2)*exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
116          +t(i)^2*x(2)^2*exp(-2*t(i)*x(4));
117  end
118  h(4,4) = 2*s;
119  s = 0;
120  for i= 1:n
121      s = s+t(i)^2*x(2)*x(3)*exp(-t(i)*(x(4)+x(5)));
122  end
123  h(4,5) = 2*s;
124
125  %% Fifth row
126  s = 0;
127  for i= 1:n
128      s = s+(t(i)*x(3))*exp(-t(i)*x(5));
129  end
130  h(5,1) = -2*s;
131  s = 0;
132  for i= 1:n
133      s = s+t(i)*x(3)*exp(-t(i)*(x(4)+x(5)));
134  end
135  h(5,2) = -2*s;
136  s = 0;
137  for i= 1:n
138      s = s+t(i)*exp(-t(i)*x(5))*(y(i)-x(1)-x(2)...
139          *exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
140          -t(i)*x(3)*exp(-2*t(i)*x(5));
141  end
142  h(5,3) = 2*s;
143  s = 0;
144  for i= 1:n
145      s = s+t(i)^2*x(2)*x(3)*exp(-t(i)*(x(4)+x(5)));
146  end
147  h(5,4) = 2*s;
148  s = 0;
149  for i= 1:n
150      s = s+(-t(i)^2)*x(3)*exp(-t(i)*x(5))*(y(i)...
151          -x(1)-x(2)*exp(-t(i)*x(4))-x(3)*exp(-t(i)*x(5)))...
152          +t(i)^2*x(3)^2*exp(-2*t(i)*x(5));
153  end
154  h(5,5) = 2*s;
```

The main function to implement any combination of the search directions (BFGS, Modified-Newton, steepest descent) and linesearch criteria Armijo conditions for optimization of the Osborne 1 function is presented. The algorithm didn't give satisfactory results for Wolfe and strong-Wolfe conditions linesearch methods.

```
clc; clear;
%%  Initial variables
x = [0.5; 1.5; -1; 0.01; 0.02];
% select a linesearch methods:
method = 1; %      1: Backtracking_Armijo
            %      2: Wolfe condition
            %      3: Strong Wolfe conditions

%% optimization
tol = 10^(-6);
iter = 20000;

% Active one of the following linesearch direction algorithms:
%%%%%% Steepest descent
% [Epar,fval,alpha,f_cont,iter] = ...
    Steepest_descent(@Osborne,x,tol,iter,method);
%%%%%% Modified Newton
[Epar,fval,alpha,f_cont,iter] = ...
    Modified_Newton(@Osborne,x,tol,iter,method);
% %%%%% BFGS
% [Epar,fval,alpha,f_cont,iter] = BFGS(@Osborne,x,tol,iter,method);

%% result
fprintf(1,'Function evaluation: %d\n', f_cont);
fprintf(1,'fval: %e\n', fval(end));
fprintf(1,'Estimated x1 is: %d\n', Epar(1));
fprintf(1,'Estimated x2 is: %d\n', Epar(2));
fprintf(1,'Estimated x3 is: %d\n', Epar(3));
fprintf(1,'Estimated x4 is: %d\n', Epar(4));
fprintf(1,'Estimated x5 is: %d\n', Epar(5));
figure(1), plot(1:length(alpha),alpha,'LineWidth',1.5)
xlabel('Iteration');  ylabel('alpha'); ylim([0,1.2]); box off
figure(2), plot(1:length(fval),log(abs(fval)),'LineWidth',1.5)
xlabel('Iteration');  ylabel('log(f(x))'); box off

%% fminunc
fprintf('Minimization MATLAB fminunc with Quasi-Newton ...
    algorithm: \n');
history= histclass;
outf= @(x,optimValues,state)outfun(x,optimValues,state,history);
options = optimoptions(@fminunc,'Display','iter','Algorithm',...
          'quasi-newton','HessUpdate','bfgs','OutputFcn',outf,...
          'TolFun',1e-6,'MaxFunEvals',2e4,'MaxIter',2e4);
xfmin = fminunc(@Osborne,x,options);
fprintf('Number of function evaluations: %d\n', history.count);
hold on
plot(1:length(history.fval),log(history.fval),'LineWidth',1.5)

%% ADMB
dat = 'ADMB.txt';  % fun values
z = load(dat);
plot(1:length(z),log(z),'LineWidth',1.5)
legend('My result','fminunc','ADMB(lmn 20)')

%%
%close all;
```

1. BFGS algorithm.

   Backtracking Armijo: Considered parameters for this method are as:

   - $\eta = 0.1$
   - $\tau = 0.5$
   - Convergence tol. : 10^{-6}
   - Maximum iteration : 20000
   - Initial values : $(0.5, 1.5, -1, 0.01, 0.02)$

   Result of running Matlab codes:

```
Minimization using BFGS Quasi-Newton algorithm:
Convergance criterion: 1.000000e-06
Maximum gradient component: 3.051841e-04
Number of iterations: 99
Function evaluation: 125
fval: 5.464897e-05
Estimated x1 is: 3.754031e-01
Estimated x2 is: 1.935031e+00
Estimated x3 is: -1.463866e+00
Estimated x4 is: 1.286588e-02
Estimated x5 is: 2.212600e-02
```

2. Modified Newton.

   Backtracking Armijo: Considered parameters for this method are as:

   - $\eta = 0.1$
   - $\tau = 0.5$
   - Convergence tol. : 10^{-6}
   - Maximum iteration : 20000
   - Initial values : $(0.5, 1.5, -1, 0.01, 0.02)$

   Result of running Matlab codes:

```
Minimization using Modified Newton algorithm:
Gradient is less than: 1.000000e-06
Number of iterations: 1613
Function evaluation: 1613
fval: 7.695402e-05
Estimated x1 is: 3.689997e-01
Estimated x2 is: 1.488999e+00
Estimated x3 is: -1.013016e+00
Estimated x4 is: 1.166378e-02
Estimated x5 is: 2.488665e-02
```

3. Steepest descent.

   Backtracking Armijo: Considered parameters for this method are as:

- $\eta = 0.1$
- $\tau = 0.5$
- Convergence tol. : 10^{-6}
- Maximum iteration : 20000
- Initial values : $(0.5, 1.5, -1, 0.01, 0.02)$

Result of running Matlab codes:

```
Minimization using Steepest Descent algorithm:
Convergance critrrion: 1.000000e-06
Maximum gradient component: 1.016705e-01
Number of iterations is larger than: 20000
fval: 1.074591e-03
Estimated x1 is: 3.874790e-01
Estimated x2 is: 1.475414e+00
Estimated x3 is: -1.025686e+00
Estimated x4 is: 1.234308e-02
Estimated x5 is: 2.647213e-02
```

***fminunc* optimization tool.** Here, we present the result of the *fmininc* function of matlab optimization tool. BFGS algorithm has been selected as line-search method.

```
Minimization MATLAB fminunc with Quasi-Newton algorithm:

                                                        First-order
 Iteration  Func-count       f(x)        Step-size       optimality
     0           6         0.879026                          412
     1          48         0.174969    1.24061e-05           32.6
     2          54         0.162181              1           16.2
     3          60         0.156862              1           4.89
     4          72         0.151856             10           7.08
     5          78         0.134945              1           13.9
     6          84         0.127579              1           14.2
     7          90         0.123165              1           9.57
     8          96         0.120699              1           2.31
     9         102          0.12049              1           2.02
    10         108         0.120401              1           2.04
    11         114             0.12              1           2.09
    12         120         0.119182              1           3.77
    13         126         0.116993              1           6.43
    14         132         0.112047              1           9.64
    15         138         0.101927              1           12.9
    16         144        0.0857742              1           15.8
    17         150        0.0636763              1           19.2
    18         156        0.0311518              1           19.9
    19         162       0.00158473              1          0.553
    20         168       0.00114871              1           2.77
    21         174       0.00104951              1           3.38
    22         180      0.000914135              1          0.424
```

```
27      23          186       0.000907364              1          0.0471
28      24          192       0.000906359              1          0.0443
29      25          204       0.000905917             10          0.0663
30      26          210       0.000903596              1           0.252
31      27          216       0.000896476              1           0.606
32      28          222        0.00087928              1            1.13
33      29          228        0.00083559              1            1.93
34      30          234       0.000736516              1            2.96
35      31          240       0.000547254              1            3.82
36      32          246       0.000296125              1            3.47
37      33          252       0.000121466              1            1.64
38      34          258       7.97332e-05              1           0.325
39      35          264       7.71077e-05              1          0.0187
40      36          270       7.70217e-05              1         0.00396
41      37          276       7.70195e-05              1        0.000373
42
43  Local minimum found.
44
45  Optimization completed because the size of the gradient is less than
46  the value of the optimality tolerance.
47
48  <stopping criteria details>
49  Number of function evaluations: 276
```

The efficiency of the employed methods for optimizing Osborne 1 function and number of iteration and final function values are presented in the Table 2. It worth mentioning that the implemented modified Newton method which uses backtracking Armijo as linesearch achieved the efficiency value one which is the best efficiency among all the methods and not mentioned in the table. Figure 9 gives the comparison of ADMB, fminunc and my implemented codes. As it can be seen in the figure my implemented algorithms declined the function values very fast. ADMB (*lmn 20*) and fminunc have very similar behavior and their pattern for declining the function values are very similar. Although general minimizer of the ADMB experienced unsuccessful termination on Osborne 1 problem another algorithms ADMB with *lmn N*, fminunc, my implemented BFGS and modified Newton finished their job with successful termination. Step size α in the implemented BFGS algorithm using strong wolfe condition is plotted in the figure 10.

Table 2: Result of different optimization tools for the Osborne 1 function

method	iteration	fun evaluation	efficiency	fun value
ADMB (lmn 20)	98	116	0.84	5.4649e-05
fminunc	37	276	0.13	7.7019e-05
my result (BFGS)	99	125	0.79	5.4648e-05

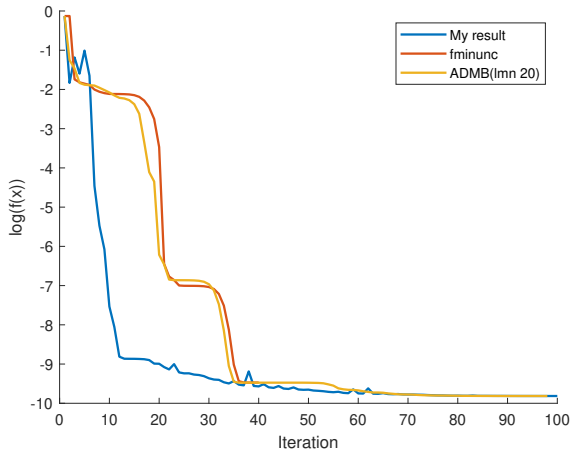

Figure 9: Logarithm of the function value at each iteration of the different optimization tools

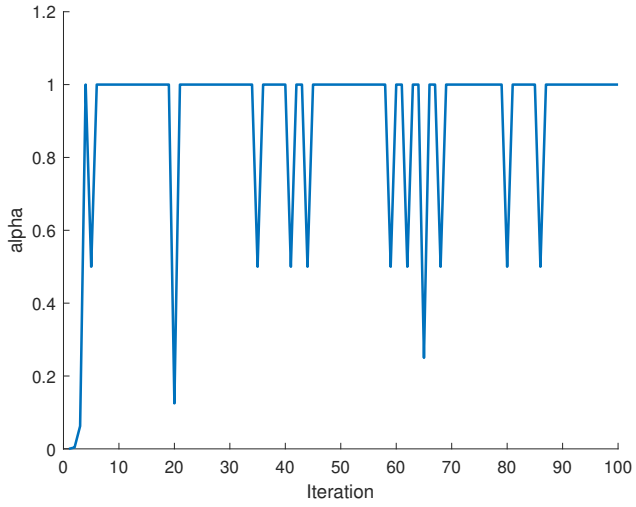

Figure 10: Step size α at each iteration for the implemented BFGS method

4.3 Discrete boundary value problem.

The discrete boundary value function $u : \mathcal{R}^n \to \mathcal{R}$ is given in [3] where in this report n is considered as 100. We describe the function u as:

$$u(x_1, ..., x_{100}) = \sum_{i=1}^{100} [2x_i - x_{i-1} - x_{i+1} + h^2(x_i + t_i + 1)^3/2]^2, \qquad (14)$$

where $h = \frac{1}{n+1}$ and $t_i = ih$ and $x_0 = x_{100} = 0$. The standard starting point for the optimization procedure is $(\zeta_i)_{i=1,...,100}$ where $\zeta_i = t_i(t_i - 1)$ and the function minimum value is $u = 0$.

ADMB Implementation. The ADMB code for minimization of the discrete boundary value problem with 100 parameters which has been implemented in the AD studio platform is presented as bellow:

```
DATA_SECTION
  init_vector y(1,100);
PARAMETER_SECTION
  init_vector x(1,100);
  !! x=y;
  objective_function_value f
PROCEDURE_SECTION
  int n = 100;
  double m = 100;
  double h = 1/(m+1);
  f = pow(2*x[1]-x[2]+pow(h,2)*pow(x[1]+h+1,3)/2,2);
  for (int i=2;i<n;i++){
    double t = i*h;
    f = f+pow(2*x[i]-x[i-1]-x[i+1]+pow(h,2)*pow(x[i]+t+1,3)/2,2);
  }
  f = f+pow(2*x[n]-x[n-1]+pow(h,2)*pow(x[n]+(n*h)+1,3)/2,2);
RUNTIME_SECTION
  maximum_function_evaluations 20000
  convergence_criteria 1.e-8
```

Results of running the above ADMB codes is given as:

```
 - final statistics:
100 variables; iteration 220; function evaluation 222
Function value  1.2195e-011; maximum gradient component mag  2.8439e-009
Exit code = 1;  converg criter  1.0000e-008
```

Var	Value	Gradient	Var	Value	Gradient	Var	Value	Gradient
1	-0.00491	2.8536e-010	2	-0.00978	-8.9763e-010	3	-0.01459	6.5602e-010
4	-0.01936	-9.0764e-010	5	-0.02407	5.5798e-010	6	-0.02872	-4.2091e-010
7	-0.03333	-4.3532e-010	8	-0.03788	-1.2381e-010	9	-0.04237	-1.2349e-009
10	-0.04681	-4.2042e-010	11	-0.05119	-1.2890e-009	12	-0.05552	-1.2921e-009
13	-0.05978	-4.0765e-010	14	-0.06399	-1.2305e-009	15	-0.06813	-1.8989e-010
16	-0.07221	-1.2261e-009	17	-0.07623	-1.4289e-009	18	-0.08019	-4.6317e-010
19	-0.08408	-9.9638e-010	20	-0.08791	-1.3549e-009	21	-0.09167	-1.0446e-009
22	-0.09536	-6.2966e-011	23	-0.09898	-8.5920e-010	24	-0.10253	1.1306e-009
25	-0.10601	-5.1344e-010	26	-0.10942	-2.7239e-010	27	-0.11275	-8.8424e-011
28	-0.11601	1.2322e-010	29	-0.11919	1.0380e-009	30	-0.12230	-2.1977e-010
31	-0.12532	1.2256e-009	32	-0.12827	1.4968e-009	33	-0.13113	8.5441e-010
34	-0.13391	2.6847e-009	35	-0.13660	1.1969e-009	36	-0.13921	5.1176e-010
37	-0.14173	1.0790e-009	38	-0.14416	7.7164e-010	39	-0.14650	1.5793e-009
40	-0.14875	1.1785e-009	41	-0.15090	2.1253e-009	42	-0.15296	1.2653e-009
43	-0.15491	1.7964e-009	44	-0.15677	1.7541e-009	45	-0.15852	2.1723e-009
46	-0.16018	1.3993e-009	47	-0.16172	7.5494e-010	48	-0.16316	1.9593e-009
49	-0.16448	1.6646e-009	50	-0.16570	1.9844e-009	51	-0.16680	2.1024e-009
52	-0.16778	1.8536e-009	53	-0.16864	2.3886e-009	54	-0.16939	8.3935e-010
55	-0.17000	1.6965e-009	56	-0.17049	1.8960e-009	57	-0.17086	1.8771e-009
58	-0.17108	1.7471e-009	59	-0.17118	1.5930e-009	60	-0.17114	2.5893e-009
61	-0.17095	1.7211e-009	62	-0.17062	1.8001e-009	63	-0.17015	1.8514e-009
64	-0.16952	1.2172e-009	65	-0.16875	1.3768e-009	66	-0.16781	1.4902e-009
67	-0.16672	2.8439e-009	68	-0.16546	9.8832e-010	69	-0.16403	2.1058e-009
70	-0.16243	1.7074e-009	71	-0.16066	6.1807e-010	72	-0.15871	1.9224e-009
73	-0.15657	4.2037e-010	74	-0.15425	1.0371e-009	75	-0.15173	3.9691e-010
76	-0.14902	6.2020e-010	77	-0.14610	1.6651e-009	78	-0.14298	4.7868e-010
79	-0.13965	4.4941e-010	80	-0.13610	4.7514e-010	81	-0.13233	-1.1465e-010
82	-0.12832	-2.4806e-010	83	-0.12409	5.0798e-010	84	-0.11962	-1.3340e-010
85	-0.11490	-5.4480e-010	86	-0.10993	9.7405e-010	87	-0.10470	-6.1888e-010
88	-0.09920	2.1030e-010	89	-0.09343	7.4697e-011	90	-0.08738	-1.0896e-009
91	-0.08105	6.6008e-010	92	-0.07442	-5.4321e-010	93	-0.06748	5.6871e-011
94	-0.06024	-1.3238e-010	95	-0.05267	5.9544e-010	96	-0.04477	1.9987e-010
97	-0.03654	-4.0639e-010	98	-0.02795	6.7329e-010	99	-0.01901	-1.1317e-010
100	-0.00969	-7.7917e-011						

Matlab Implementation. In order to write a matlab tool to minimize the discrete boundary value problem we present the matlab functions of the discrete boundary value function and it's gradient vector as following:

```
function f = dbvf(x)
%  DBVF function
n = length(x);
t = zeros(1,n);
h = 1/(n+1);
for i = 1:n
    t(i) = i*h;
end
f = (2*x(1)-x(2)+h^2*(x(1)+t(1)+1)^.3/2).^2;
for i = 2:n-1
    f = f+(2*x(i)-x(i-1)-x(i+1)+h^2*(x(i)+t(i)+1).^3/2).^2;
end
  f = f+(2*x(n)-x(n-1)+h^2*(x(n)+t(n)+1).^3/2).^2;
```

```
function g = grad(x)
%  DBVF gradient
n = length(x);
g = zeros(n,1);
t = zeros(n,1);
h = 1/(n+1);
for i = 1:n
    t(i) = i*h;
end
% first element
g(1) = 2*((2+3*h^2*(x(1)-t(1)+1).^2/2)*(2*x(1)...
    -x(2)+h^2*(x(1)+t(1)+1).^3/2)-(2*x(2)-x(1)...
    -x(3)+h^2*(x(2)+t(2)+1).^3/2));
% second element
g(2) = 2*(-(2*x(1)-x(2)+h^2*(x(1)+t(1)+1).^3/2)...
    +(2+3*h^2*(x(2)-t(2)+1).^2/2)*(2*x(2)-x(1)...
    -x(3)+h^2*(x(2)+t(2)+1).^3/2)-(2*x(3)-x(2)...
    -x(4)+h^2*(x(3)+t(3)+1).^3/2));
% 3 to n-1 element
for i = 3:n-2
    g(i) = 2*(-(2*x(i-1)-x(i-2)-x(i)+h^2*(x(i-1)...
        +t(i-1)+1).^3/2)+(2+3*h^2*(x(i)-t(i)+1).^2/2)...
        *(2*x(i)-x(i-1)-x(i+1)+h^2*(x(i)+t(i)+1).^3/2)...
         -(2*x(i+1)-x(i)-x(i+2)+h^2*(x(i+1)+t(i+1)+1).^3/2));
end
% n-1 element
g(n-1) = 2*(-(2*x(n-2)-x(n-3)-x(n-1)+h^2*(x(n-2)...
    +t(n-2)+1).^3/2)+(2+3*h^2*(x(n-1)-t(n-1)+1).^2/2)...
    *(2*x(n-1)-x(n-2)-x(n)+h^2*(x(n-1)+t(n-1)+1).^3/2)...
            -(2*x(n)-x(n-1)+h^2*(x(n)+t(n)+1).^3/2));
% last element
g(n) = 2*(-(2*x(n-1)-x(n-2)-x(n)+h^2*(x(n-1)...
    +t(n-1)+1).^3/2)+(2+3*h^2*(x(n)-t(n)+1).^2/2)...
    *(2*x(n)-x(n-1)+h^2*(x(n)+t(n)+1).^3/2));
```

The main function to implement any combination of the search directions (BFGS, Modified-Newton, steepest descent) and linesearch criteria Armijo conditions for optimization of the discrete boundary value problem with 100 parameters is presented. The algorithm didn't give satisfactory results for Wolfe and strong-Wolfe conditions linesearch methods.

```
clc; clear;
%%  Initial variables
n = 100;
x = zeros(n,1);
h = 1/(n+1);
for i = 1:n
    t = i*h;
    x(i) = t*(t-1);
end
% select a linesearch methods:
method = 1; %      1: Backtracking_Armijo
            %      2: Wolfe condition
            %      3: Strong Wolfe conditions

%% optimization
tol = 10^(-8);
iter = 20000;
% %%%%% BFGS
[Epar,fval,alpha,f_cont,iter] = BFGS(@dbvf,x,tol,iter,method);

%% result
fprintf(1,'Function evaluation: %d\n', f_cont);
fprintf(1,'fval: %e\n', fval(end));
for i=1:n
    fprintf(1,'Estimated x%d is: %d\n', i,Epar(i));
end
figure(1), plot(1:length(alpha),alpha,'LineWidth',1.5)
xlabel('Iteration');  ylabel('alpha'); ylim([0,1.2]); box off
figure(2), plot(1:length(fval),log(abs(fval)),'LineWidth',1.5)
xlabel('Iteration');  ylabel('log(f(x))'); box off

%% fminunc
fprintf('Minimization MATLAB fminunc with Quasi-Newton ...
    algorithm: \n');
history= histclass;
outf= @(x,optimValues,state)outfun(x,optimValues,state,history);
options = optimoptions(@fminunc,'Display','iter','Algorithm',...
          'quasi-newton','HessUpdate','bfgs','OutputFcn',outf,...
          'TolFun',1e-8,'MaxFunEvals',2e4,'MaxIter',2e4);
xfmin = fminunc(@dbvf,x,options);
fprintf('Number of function evaluations: %d\n', history.count);
hold on
plot(1:length(history.fval),log(history.fval),'LineWidth',1.5)

%% ADMB
dat = 'ADMB.txt';  % fun values
z = load(dat);
plot(1:length(z),log(z),'LineWidth',1.5)
legend('My result','fminunc','ADMB')
```

1. BFGS algorithm.

   Backtracking Armijo: Considered parameters for this method are as:

   - $\eta = 0.1$
   - $\tau = 0.5$
   - Convergence tol. : 10^{-8}
   - Maximum iteration : 20000
   - Initial values : $x_i = t_i(t_i - 1), \quad t_i = \frac{i}{n+1}, \quad i = 1, 2, \cdots, n.$

Result of running Matlab codes:

```
Minimization using BFGS Quasi-Newton algorithm:
Convergance criteion: 1.000000e-08
Maximum gradient component: 6.329249e-09
Number of iterations: 331
Function evaluation: 3119
fval: 4.384541e-13
Estimated x1 is: -4.925696e-03
Estimated x2 is: -9.801642e-03
Estimated x3 is: -1.462709e-02
Estimated x4 is: -1.940127e-02
Estimated x5 is: -2.412340e-02
Estimated x6 is: -2.879270e-02
Estimated x7 is: -3.340833e-02
Estimated x8 is: -3.796948e-02
Estimated x9 is: -4.247530e-02
Estimated x10 is: -4.692493e-02
Estimated x11 is: -5.131747e-02
Estimated x12 is: -5.565203e-02
Estimated x13 is: -5.992769e-02
Estimated x14 is: -6.414351e-02
Estimated x15 is: -6.829853e-02
Estimated x16 is: -7.239177e-02
Estimated x17 is: -7.642223e-02
Estimated x18 is: -8.038887e-02
Estimated x19 is: -8.429067e-02
Estimated x20 is: -8.812654e-02
Estimated x21 is: -9.189539e-02
Estimated x22 is: -9.559612e-02
Estimated x23 is: -9.922757e-02
Estimated x24 is: -1.027886e-01
Estimated x25 is: -1.062780e-01
Estimated x26 is: -1.096945e-01
Estimated x27 is: -1.130369e-01
Estimated x28 is: -1.163039e-01
Estimated x29 is: -1.194943e-01
Estimated x30 is: -1.226066e-01
Estimated x31 is: -1.256395e-01
Estimated x32 is: -1.285916e-01
Estimated x33 is: -1.314615e-01
Estimated x34 is: -1.342477e-01
```

```
 Estimated x35 is: -1.369487e-01
 Estimated x36 is: -1.395629e-01
 Estimated x37 is: -1.420888e-01
 Estimated x38 is: -1.445248e-01
 Estimated x39 is: -1.468692e-01
 Estimated x40 is: -1.491203e-01
 Estimated x41 is: -1.512764e-01
 Estimated x42 is: -1.533357e-01
 Estimated x43 is: -1.552963e-01
 Estimated x44 is: -1.571564e-01
 Estimated x45 is: -1.589141e-01
 Estimated x46 is: -1.605675e-01
 Estimated x47 is: -1.621143e-01
 Estimated x48 is: -1.635527e-01
 Estimated x49 is: -1.648805e-01
 Estimated x50 is: -1.660955e-01
 Estimated x51 is: -1.671954e-01
 Estimated x52 is: -1.681780e-01
 Estimated x53 is: -1.690409e-01
 Estimated x54 is: -1.697817e-01
 Estimated x55 is: -1.703978e-01
 Estimated x56 is: -1.708867e-01
 Estimated x57 is: -1.712459e-01
 Estimated x58 is: -1.714725e-01
 Estimated x59 is: -1.715638e-01
 Estimated x60 is: -1.715170e-01
 Estimated x61 is: -1.713290e-01
 Estimated x62 is: -1.709969e-01
 Estimated x63 is: -1.705176e-01
 Estimated x64 is: -1.698879e-01
 Estimated x65 is: -1.691044e-01
 Estimated x66 is: -1.681638e-01
 Estimated x67 is: -1.670627e-01
 Estimated x68 is: -1.657973e-01
 Estimated x69 is: -1.643640e-01
 Estimated x70 is: -1.627589e-01
 Estimated x71 is: -1.609782e-01
 Estimated x72 is: -1.590179e-01
 Estimated x73 is: -1.568736e-01
 Estimated x74 is: -1.545411e-01
 Estimated x75 is: -1.520160e-01
 Estimated x76 is: -1.492936e-01
 Estimated x77 is: -1.463693e-01
 Estimated x78 is: -1.432381e-01
 Estimated x79 is: -1.398950e-01
 Estimated x80 is: -1.363348e-01
 Estimated x81 is: -1.325522e-01
 Estimated x82 is: -1.285415e-01
 Estimated x83 is: -1.242970e-01
 Estimated x84 is: -1.198127e-01
 Estimated x85 is: -1.150826e-01
 Estimated x86 is: -1.101002e-01
 Estimated x87 is: -1.048589e-01
 Estimated x88 is: -9.935209e-02
 Estimated x89 is: -9.357254e-02
 Estimated x90 is: -8.751299e-02
 Estimated x91 is: -8.116589e-02
```

```
Estimated x92 is: −7.452338e−02
Estimated x93 is: −6.757734e−02
Estimated x94 is: −6.031933e−02
Estimated x95 is: −5.274062e−02
Estimated x96 is: −4.483211e−02
Estimated x97 is: −3.658440e−02
Estimated x98 is: −2.798770e−02
Estimated x99 is: −1.903185e−02
Estimated x100 is: −9.706273e−03
```

***fminunc* optimization tool.** Here, we present the result of the *fmininc* function of matlab optimization tool. BFGS algorithm has been selected as line-search method.

```
Minimization MATLAB fminunc with Quasi−Newton algorithm:

                                                        First−order
 Iteration  Func−count       f(x)        Step−size       optimality
     0         101      1.23293e−06                      0.000392
     1         303      1.22094e−06            0.1       0.000313
     2         404      1.21384e−06              1       0.000179
     3         505      1.20605e−06              1       0.000171
     4         606      1.20088e−06              1       0.000182
     5         707      1.19511e−06              1       0.000122
     6         808      1.19033e−06              1       0.000123
     7         909      1.18573e−06              1       0.000104
     8        1010      1.18147e−06              1       0.000115
     9        1111      1.17737e−06              1       9.26e−05
    10        1212       1.1735e−06              1       0.000109
    11        1313      1.16982e−06              1       9.12e−05
    12        1414      1.16628e−06              1       8.29e−05
    13        1515      1.16286e−06              1       8.19e−05
    14        1616      1.15957e−06              1       7.48e−05
    15        1717       1.1564e−06              1       6.89e−05
    16        1818      1.15332e−06              1       8.14e−05
    17        1919      1.15032e−06              1       6.57e−05
    18        2020      1.14742e−06              1       7.45e−05
    19        2121       1.1446e−06              1       6.84e−05
    20        2222      1.14184e−06              1       7.18e−05
    21        2323      1.13915e−06              1       6.88e−05
    22        2424      1.13653e−06              1       6.57e−05
    23        2525      1.13396e−06              1       6.31e−05
    24        2626      1.13145e−06              1        5.7e−05
    25        2727      1.12899e−06              1       6.15e−05
    26        2828      1.12658e−06              1       5.74e−05
    27        2929      1.12421e−06              1       5.28e−05
    28        3030      1.12189e−06              1       6.01e−05
    29        3131      1.11961e−06              1       5.18e−05
    30        3232      1.11737e−06              1       5.31e−05
    31        3333      1.11518e−06              1        5.6e−05
    32        3434      1.11301e−06              1       5.15e−05
    33        3535      1.11088e−06              1       5.34e−05
    34        3636      1.10878e−06              1       5.07e−05
    35        3737      1.10672e−06              1       4.91e−05
    36        3838      1.10468e−06              1       4.84e−05
```

```
    37        3939      1.10268e-06             1        4.63e-05
    38        4040       1.1007e-06             1         4.9e-05
    39        4141      1.09875e-06             1        4.84e-05
     .           .             .                .             .
     .           .             .                .             .

                                                        First-order
 Iteration  Func-count       f(x)         Step-size       optimality
   170        17372      9.13998e-07             1        9.74e-06
   171        17473      9.13711e-07             1        1.42e-05
   172        17574      9.13079e-07             1        1.96e-05
   173        17675      9.11813e-07             1        2.48e-05
   174        17776      9.09892e-07             1        2.72e-05
   175        17877      9.08215e-07             1        2.03e-05
   176        17978      9.07576e-07             1         8.5e-06
   177        18079       9.0746e-07             1         5.3e-06
   178        18180      9.07412e-07             1        6.12e-06
   179        18281      9.07292e-07             1        7.59e-06
   180        18382      9.07024e-07             1        1.12e-05
   181        18483      9.06401e-07             1        1.63e-05
   182        18584      9.05244e-07             1        2.17e-05
   183        18685      9.03767e-07             1        2.16e-05
   184        18786      9.02818e-07             1        1.28e-05
   185        18887      9.02555e-07             1        4.97e-06
   186        18988      9.02492e-07             1        4.63e-06
   187        19089      9.02418e-07             1        6.31e-06
   188        19190      9.02219e-07             1        9.17e-06
   189        19291      9.01743e-07             1        1.35e-05
   190        19392      9.00574e-07             1        1.99e-05
   191        19493       8.9801e-07             1        2.76e-05
   192        19594      8.93468e-07             1        3.47e-05
   193        19695      8.88393e-07             1        2.94e-05
   194        19796      8.85703e-07             1        1.51e-05
   195        19897      8.85091e-07             1        7.53e-06
   196        19998      8.84935e-07             1        7.44e-06
   197        20099      8.84694e-07             1        8.98e-06

Solver stopped prematurely.

fminunc stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 2.000000e+04.

Number of function evaluations: 20099
```

Comparison of efficiency of the employed methods for optimizing discrete boundary value problem as well as number of iterations and function evaluations are presented in the Table 3. As it can be seen in the table and result from matlab optimization too, fminunc function finished the minimization with unsuccessful termination since number of function evaluations exceeds the maximum criterion. fminunc has also the weakest efficiency value among the methods. Figure 11 gives the comparison of ADMB, fminunc and my implemented codes. Step size α in the implemented BFGS algorithm using strong wolfe condition is plotted in the figure 12.

Table 3: Result of different optimization tools for the discrete boundary value problem

method	iteration	fun evaluation	efficiency	fun-value
ADMB	220	222	0.99	1.2195e-11
fminunc	197	20099	0.01	8.8469e-07
my result	331	3119	0.11	4.3845e-13

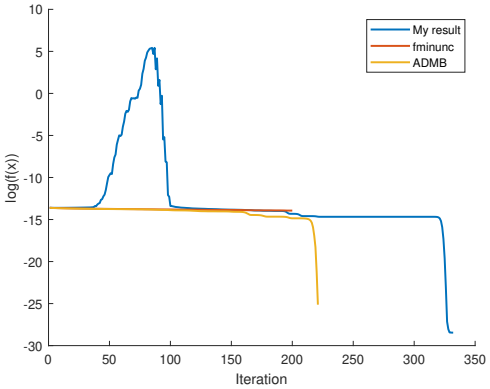

Figure 11: Logarithm of the function value at each iteration of the different optimization tools

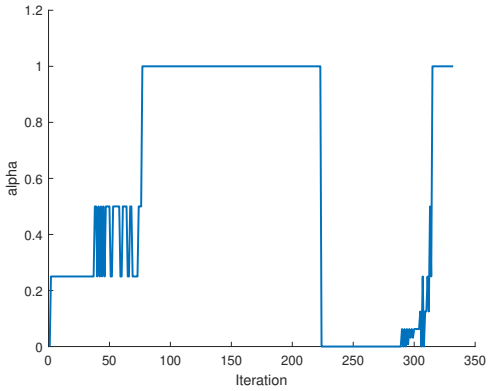

Figure 12: Step size α at each iteration for the implemented BFGS method

5 Discussion

Results from comparison of the employed optimization tools indicate that efficiency of the ADMB is much more than *fminunc* and my implemented algorithms. This holds particularly for discrete boundary value function where the dimension of the problem is very big. It is because ADMB has been designed to deal with large scale problems with many parameters. Optimization tool in ADMB can work more effectively if the procedure of optimization be divided into more than one phase where in each phase only part of parameters be estimated. However in this report ADMB optimization is done in one phase to be comparable with two other tools.

Our observations show that efficiency of the optimization tool *fminunc* is the lowest in all of the test problems. One of the important reasons is that *fminunc* uses approximation of the gradient while ADMB and implemented matlab codes are using accurate information of the gradient and Hessian matrix. It is another important benefit of using ADMB because this platform is able to calculate exact gradient vector and Hessian matrix which can extremely help the minimizer.

1) Rosenbrock function.

The general minimizer of ADMB uses BFGS search direction with Strong-wolfe condition to minimize the Rosenbrock function. We have implemented similar search direction with similar linesearch method in Matlab. The termination condition in both ADMB and the implemented matlab codes is set by falling the gradient value in current point less than convergence criterion multiplied by the value of gradient vector in the initial values of parameters. So we can claim that we are using same method with same maximum function evaluations (20000) and convergence ceriteria (10^{-8}) as in ADMB. Termination condition in *fminunc* is different than two other tools where it controls if the gradient vector falls bellow a threshold. Successful terminations for the ADMB, *fminunc* and my implemented algorithms (except steepest descent) are observed.

Comparison of the ABMB results in section 4.1 with *fminunc* and matlab implementation shows that all of the optimization tools successfully finished the process and reached the exact parameter value $(1, 1)$ of the minimized function. The main difference between the tools is that implemented matlab codes with BFGS search direction used only 26 iteration to complete the process but in ADMB and *fminunc* iteration numbers are 38 and 37, respectively. The final function value in ADMB is $5.23e-21$ which is the smallest among the tools.

I have also implemented other algorithms than ADMB in matlab. I observed same mentioned result in above paragraph by employing BFGS method with other linesearch algorithms such as wofle conditions and backtracking Armijo. Perfect results have been obtained by using Modified Newton algorithm with different linesearch criterion. Modified Newton with any of the linesearch meth-

ods reaches the exact minimization parameter values with 16 iterations. We also used steepest descent method with different linesearches. Steepest descent combined with any of the linesearch methods uses whole 20000 iteration but didn't satisfy the termination criterion. Best result from steepest descent obtained by combination with strong-wolfe condition where the maximum gradient component is $1.21e-4$.

2) Osborne 1 function.

The BFGS method and strong-wolfe condition in implemented matlab codes didn't give satisfactory results so we can not compare it with the results obtained with ADMB. I only reported the results obtained from different search directions and backtracking Armijo implemented in matlab and results from ADMB.

The general minimizer of the ADMB was not able to converge to the solution of problem. The final error expressed that Hessian matrix is not positive definite which means that ADMB optimizer got stuck in the Newton algorithm and couldn't reach the BFGS updating section. I used limited memory BFGS with two different number of keeping pass step information. For both of the cases ($N = 5$ and $N = 20$) ADMB converged and finished the process. In the case of $N = 5$ number of iteration was 112 and for $N = 20$ it was 98.

The result of matlab implementation for BFGS with backtracking Armijo is very similar to ADMB (*-lmn 20*) where the implemented minimizer took 99 iteration to finish the process. As it can be seen in the figure 9 the behavior of ADMB (*-lmn 20*) and *fminunc* for converging the minimum value is very similar since both of them following same pattern. The matlab implementation of Modified Newton method with backtracking Armijo linesearch algorithm also converged to the solution after 1613 iteration where achieved efficiency value one. Steepest descent algorithm with same above linesearch criterion didn't converge to the solution after 20000 iteration but acceptable result is obtained.

3) Discrete boundary value problem.

It is not possible to compare the result for ADMB and matlab implementations since the implemented matlab BFGS method with strong-wolfe conditions as line search steps didn't give acceptable results. Therefor we discuss the results of ADMB and implemented algorithms separately.

The ADMB converged to the solution after 220 iteration with efficiency value 0.99 which can be considered as best result among the tools. Implemented matlab code uses BFGS method with backtracking Armijo linesearch algorithm to minimize the problem. The observed number of iteration for matlab tool is 330 and efficiency value is 0.11. The smallest efficiency belongs to the *fminunc* where experienced unsuccessful termination for the optimization procedure.

Conclusion. In conclusion ADMB optimization tool obtain better results than other two minimization tools. In the case of large scale problems *fminunc* has very poor results and ADMB work properly. Our results suggest to use the

optimization tools which are able to take benefits of exact information of the gradient and Hessian matrix.

References

[1] D. A. Fournier, H. J. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. N. Maunder, A. Nielsen, J. Sibert, AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models, Optimization Methods and Software 72 (2) (2012) 233–249.

[2] J. Nocedal, S. J. Wright, Numerical Optimization, Springer, USA, 2006.

[3] J. J. Moré, B. S. Garbow, K. E. Hillstrom, Testing Unconstrained Optimization Software, ACM Transactions on Mathematical Software 7 (1) (1981) 17–41.

Appendix

Numerical function minimization written in C++ which presented in the ADMB documentation is as bellow:

```
/*
 * $Id$
 *
 * Author: Unknown
 * Copyright (c) 2009-2012 ADMB Foundation
 *
 * This file was originally written in FORTRAN II by an unknown author.
 * In the 1980s, it was ported to C and C++ and extensively modified by
 * David Fournier.
 *
 */
#include <fvar.hpp>
#if defined(_WIN32)
 #include <windows.h>
 #include <admodel.h>
#endif
#if defined(__BORLANDC__)
 #include <signal.h>
#endif
#ifdef __ZTC__
 #include <conio.h>
#endif
#include <fvar.hpp>
extern int ctlc_flag;

#if defined (__WAT32__) || defined (_MSC_VER)
 #include <conio.h>
#endif
#if defined(__CYGWIN__)
 int kbhit(void)
 {
 return 0;
 }
#endif
#ifdef __ZTC__
 #include <iostream.hpp>
```

```
 #include <disp.h>
 #define endl "\n"
#endif
 #include <signal.h>
#ifdef __NDPX__
 #include <iostream.hxx>
#endif

#if defined (_MSC_VER)
 void __cdecl clrscr();
#else
 extern "C" void clrscr();
 #define getch getchar
#endif

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#ifdef _MSC_VER
BOOL CtrlHandler(DWORD fdwCtrlType)
{
 if (fdwCtrlType == CTRL_C_EVENT)
 {
 //Should exit if CTRL_C_EVENT occurs again.
 if (ctlc_flag) ad_exit(1);

 ctlc_flag = 1;
 if (ad_printf)
 (*ad_printf)("\npress q to quit or c to invoke derivative checker: ");
 return true;
 }
 return false;
}
#else
extern "C" void onintr(int k)
{
 signal(SIGINT, exit_handler);
 ctlc_flag = 1;
 if (ad_printf)
 (*ad_printf)("\npress q to quit or c to invoke derivative checker"
 " or s to stop optimizing: ");
}
```

```
 86 #endif
 87
 88 int* pointer_to_phase = 0;
 89
 90 double dafsqrt( double x );
 91
 96  void tracing_message(int _traceflag,const char *s)
 97  {
 98  if (_traceflag)
 99  {
100  ofstream ofs;
101  ofs.open("adtrace",ios::app);
102  ofs << s << endl;
103  ofs.close();
104  }
105  }
106
111  void tracing_message(int _traceflag,const char *s,int *pn)
112  {
113  if (_traceflag)
114  {
115  ofstream ofs;
116  ofs.open("adtrace",ios::app);
117  ofs << s << " " << *pn << endl;
118  ofs.close();
119  }
120  }
121
126  void tracing_message(int _traceflag,const char *s,double *pd)
127  {
128  if (_traceflag)
129  {
130  ofstream ofs;
131  ofs.open("adtrace",ios::app);
132  ofs << s << " " << *pd << endl;
133  ofs.close();
134  }
135  }
136
141  void tracing_message(int _traceflag,const char *s,double d)
142  {
143  if (_traceflag)
144  {
145  ofstream ofs;
146  ofs.open("adtrace",ios::app);
147  ofs << s << " " << d << endl;
```

```
148  ofs.close();
149  }
150  }
151 int log_values_switch=0;
152 ofstream logstream("fmin.log");
153 
158 void print_values(const double& f, const dvector & x,const dvector& g)
159 {
160  logstream << setprecision(13) << f << endl;
161  logstream << setprecision(13) << x << endl;
162  logstream << setprecision(13) << g << endl;
163 }
164 extern int traceflag;
165 //#pragma warn -sig
166 
167 
214 void fmm::fmin(const double& _f, const dvector &_x, const dvector& _g)
215 {
216  if (log_values_switch)
217  {
218  print_values(_f,_x,_g);
219  }
220 
221 #ifdef DEBUG
222  adtimer fmintime;
223 #endif
224 
225  tracing_message(traceflag,"A3");
226 
227  /* Remember gradient and function values
228  resulted from previous function evaluation */
229  dvector& g=(dvector&) _g;
230  double& f=(double&) _f;
231 
232  /* Create local vector x as a pointer to the argument vector _x */
233  independent_variables& x= (independent_variables&) _x;
234  #ifdef DIAG
235  cout << "On entry to fmin: " << *this << endl;
236  #endif
237  tracing_message(traceflag,"A4");
238 
239 #ifdef _MSC_VER
240  SetConsoleCtrlHandler((PHANDLER_ROUTINE)CtrlHandler, true);
241 #else
242  /* Check the value of control variable ireturn:
243  -1 (exit status)
```

```
244  0 (initialization of function minimizer)
245  1 (call1 - x update and convergence check)
246  2 (call2 - line search and Hessian update)
247  >=3 (derivative check)
248  */
249  if (ireturn <= 0 )
250  {
251  signal(SIGINT, &onintr);
252  }
253 #endif
254
255 #ifdef __ZTC__
256  if (ireturn <= 0 )
257  {
258  if (disp_inited == 0)
259  {
260  disp_open();
261  disp_usebios();
262  }
263  }
264 #endif
265  tracing_message(traceflag,"A5");
266  if (ireturn ==1 && dcheck_flag ==0)
267  {
268  ireturn = 3;
269  }
270  if (ireturn >= 3)
271  {
272  /* Entering derivative check */
273  derch(f, x, g, n, ireturn);
274  return;
275  }
276  if (ireturn == 1) goto call1;
277  if (ireturn == 2) goto call2;
278
279  /* we are here because ireturn=0
280  Start initializing function minimizer variables */
281  fbest=1.e+100;
282  tracing_message(traceflag,"A6");
283
284  /* allocate Hessian
285  h - object of class dfsdmat, the memory is allocated
286  only for elements of lower triagonal matrix */
287  if (!h) h.allocate(n);
288
289  /* initialize w, the dvector of size 4*n:
```

```
290  w(1,n) contains gradient vector in the point x_new = x_old + alpha*p_k
291  w(n+1,2n) contains direction of linear search, i.e. transpose(p_k)
292  w(2n+1,4n) are used for Hessian updating terms */
293  w.initialize();
294
295  /* alpha - the Newton step size */
296  alpha=1.0; /* full Newton step at the beginning */
297  ihflag=0;
298
299  /* validity check for dimensions and indexing of
300  independent vector and gradient vector
301  Note, this function will work correctly only if
302  indices start at 1 */
303  if (n <= 0)
304  {
305  cerr << "Error -- the number of active parameters"
306  " fmin must be > 0\n";
307  ad_exit(1);
308  }
309  tracing_message(traceflag,"A7");
310  if (x.indexmin() !=1)
311  {
312  cerr << "Error -- minimum valid index"
313  " for independent_variables in fmin must be 1\n"
314  << " it is " << x.indexmin() << "\n";
315  ad_exit(1);
316  }
317  if (x.size() < static_cast<unsigned int>(n))
318  {
319  cerr << "Error -- the size of the independent_variables"
320  " which is " << x.size() << " must be >= " << n << "\n"
321  << " the number of independent variables in fmin\n";
322  ad_exit(1);
323  }
324  tracing_message(traceflag,"A8");
325  if (g.indexmin() !=1)
326  {
327  cerr << "Error -- minimum valid index"
328  " for the gradient vector in fmin must be 1\n"
329  << " it is " << g.indexmin() << "\n";
330  ad_exit(1);
331  }
332  if (g.size() < static_cast<unsigned int>(n))
333  {
334  cerr << "Error -- the size of the gradient vector"
335  " which is " << g.size() << " must be >=\n"
```

```
<< " the number of independent variables in fmin\n";
ad_exit(1);
}
/* end of validity check */
tracing_message(traceflag,"A9");

/* xx is reserved for the updated values of independent variables,
at the moment put the current values */
for (i=1; i<=n; i++)
xx.elem(i)=x.elem(i);
tracing_message(traceflag,"A10");

/* itn - iteration counter */
itn=0;
/* icc - obsolete calls counter? */
icc=0;

/* initialize funval vector,
it will contain last 10 function values (10-th is the most recent)
needed to stop minimization in case if f(1)-f(10)<min_improve */
for (i=1; i< 11; i++)
funval.elem(i)=0.;
tracing_message(traceflag,"A11");
ihang = 0; /* flag, =1 when function minimizer is not making progress */
llog=1;
np=n+1;
n1=n-1;
nn=n*np/2;
is=n;
iu=n;
iv=n+n;
ib=iv+n;
iexit=0; /* exit code */
tracing_message(traceflag,"A12");

/* Initialize hessian to the unit matrix */
h.elem(1,1) = 1;
for (i=2; i<=n; i++)
{
for ( j=1; j<i; j++)
{
h.elem(i,j)=0;
}
h.elem(i,i)=1;
}
tracing_message(traceflag,"A13");
```

```
382
383 /* dmin - minimal element of hessian used to
384 verify its positive definiteness */
385 dmin=h.elem(1,1);
386 for ( i=2; i<=n; i++)
387 {
388 if(h.elem(i,i)<dmin)
389 dmin=h.elem(i,i);
390 }
391 if (dmin <= 0.) /* hessian is not positive definite? */
392 goto label7020; /* exit */
393 if(dfn == 0.)
394 z = 0.0;
395 tracing_message(traceflag,"A14");
396 for (i=1; i<=n; i++)
397 {
398 xsave.elem(i)=x.elem(i);
399 x.elem(i)=xx.elem(i);
400 }
401 ireturn=1; /* upon next entry will go to call1 */
402 tracing_message(traceflag,"A15");
403 if (h.disk_save())
404 {
405 cout << "starting hessian save" << endl;
406 h.save();
407 cout << "finished hessian save" << endl;
408 }
409 tracing_message(traceflag,"A16");
410 return;
411 /* End of initialization */
412 call1: /* first line search step and x update */
413 tracing_message(traceflag,"A17");
414 if (h.disk_save())
415 {
416 cout << "starting hessian restore" << endl;
417 h.restore();
418 cout << "finished hessian restore" << endl;
419 }
420 tracing_message(traceflag,"A18");
421 for (i=1; i<=n; i++)
422 {
423 x.elem(i)=xsave.elem(i);
424 }
425 ireturn=3;
426 tracing_message(traceflag,"A19");
427 {
```

```
 }
 for ( i=1; i<=n; i++)
 gbest.elem(i)=g.elem(i);
 tracing_message(traceflag,"A20");
 funval.elem(10) = f;
 df=dfn;
 if(dfn == 0.0)
 df = f - z;
 if(dfn < 0.0)
 df=fabs(df * f);
 if(df <= 0.0)
 df=1.;
label20: /* check for convergence */
 ic=0; /* counter for number of calls */
 iconv = 1; /* convergence flag: 1 - convergence, 2 - not yet */
 for ( i=1; i<=9; i++)
 funval.elem(i)= funval.elem(i+1);
 funval.elem(10) = f;
 /* check if function value is improving */
 if ( itn>15 && fabs( funval.elem(1)-funval.elem(10))< min_improve )
 ihang = 1;
 gmax = 0;
 /* satisfy convergence criterion? */
 for ( i=1; i<=n; i++)
 {
 if(fabs(g.elem(i)) > crit) iconv = 2;
 if(fabs(g.elem(i)) > fabs(gmax) ) gmax = g.elem(i);
 }
 /* exit if either convergence or no improvement has been achieved
 during last 10 iterations */
 if( iconv == 1 || ihang == 1)
 {
 ireturn=-1;
 goto label92;
 }
 /* otherwise proceed to the Newton step (label21) */
 if(iprint == 0)
 goto label21; /* without printing */
 if(itn == 0)
 goto label7000; /* printing Initial statistics first */
#if defined(USE_DDOUBLE)
#undef double
 if(fmod(double(itn),double(iprint)) != 0)
 goto label21;
#define double dd_real
```

```
#else
 if(fmod(double(itn),double(iprint)) != 0)
 goto label21;
#endif
 if (llog) goto label7010;
#if defined (_MSC_VER) && !defined (__WAT32__)
 if (!scroll_flag) clrscr();
#endif
label7003: /* Printing table header */
 if (iprint>0)
 {
 if (ad_printf)
 {
 (*ad_printf)("%d variables; iteration %ld; function evaluation %ld",
 n, itn, ifn);
 if (pointer_to_phase)
 {
 (*ad_printf)("; phase %d", *pointer_to_phase);
 }
 (*ad_printf)(
 "\nFunction value %15.7le; maximum gradient component mag %12.4le\n",
#if defined(USE_DDOUBLE)
 #undef double
 double(f), double(gmax));
 #define double dd_real
#else
 f, gmax);
#endif
 }
 }
/*label7002:*/
 /* Printing Statistics table */
 if(iprint>0)
 {
 fmmdisp(x, g, n, this->scroll_flag,noprintx);
 }
label21 : /* Calculating Newton step */
 /* found good search direction, increment iteration number */
 itn=itn+1;
 for (i=1; i<=n; i++)
 x.elem(i)=xx.elem(i);
 w.elem(1)=-g.elem(1);

#ifdef DEBUG
 cout << __FILE__ << ':' << __LINE__ << ' '
```

```
518  << fmintime.get_elapsed_time_and_reset() << endl;
519 #endif
520
521  /* solving system of linear equations H_(k+1) * (x_(k+1)-x(k)) = -g_k
522  to get next search direction
523  p_k = (x_(k+1)-x(k)) = - inv(H_(k+1)) * g_k */
524  for (i=2; i<=n; i++)
525  {
526  i1=i-1;
527  z=-g.elem(i);
528  double * pd=&(h.elem(i,1));
529  double * pw=&(w.elem(1));
530  for (j=1; j<=i1; j++)
531  {
532  z-=*pd++ * *pw++;
533  }
534  w.elem(i)=z;
535  }
536  w.elem(is+n)=w.elem(n)/h.elem(n,n);
537  {
538  dvector tmp(1,n);
539  tmp.initialize();
540  for (i=1; i<=n1; i++)
541  {
542  j=i;
543  double * pd=&(h.elem(n-j+1,n-1));
544  double qd=w.elem(is+np-j);
545  double * pt=&(tmp(1));
546  for (int ii=1; ii<=n1; ii++)
547  {
548  *pt++ +=*pd-- * qd;
549  }
550  w.elem(is+n-i)=w.elem(n-i)/h.elem(n-i,n-i)-tmp(i);
551  }
552  }/* w(n+1,2n) now contains search direction
553  with current Hessian approximation */
554  gs=0.0;
555  for (i=1; i<=n; i++)
556  gs+=w.elem(is+i)*g.elem(i);/* gs = -inv(H_k)*g_k*df(x_k+alpha_k*p_k) */
557  iexit=2;
558  if(gs >= 0.0)
559  goto label92; /* exit with error */
560  gso=gs;
561  /* compute new step length 0 < alpha <= 1 */
562  alpha=-2.0*df/gs;
563  if(alpha > 1.0)
```

```
564  alpha=1.0;
565  df=f;
566  tot=0.0;
567 label30: /* Taking a step, updating x */
568  iexit=3;
569  if (ialph) { goto label92; } /* exit if step size too small */
570  /* exit if number of function evaluations exceeded maxfn */
571  if( ifn >= maxfn)
572  {
573  maxfn_flag=1;
574  goto label92;
575  }
576  else
577  {
578  maxfn_flag=0;
579  iexit=1;
580  }
581  if(quit_flag) goto label92;
582  for (i=1; i<=n; i++)
583  {
584  /* w(n+1,2n) has the next direction to go */
585  z=alpha*w.elem(is+i);
586  /* new independent vector values */
587  xx.elem(i)+=z;
588  }
589  for (i=1; i<=n; i++)
590  { /* save previous values and update x return value */
591  xsave.elem(i)=x.elem(i);
592  gsave.elem(i)=g.elem(i);
593  x.elem(i)=xx.elem(i);
594  fsave = f;
595  }
596  fsave = f;
597  ireturn=2;
598  if (h.disk_save())
599  {
600  cout << "starting hessian save" << endl;
601  h.save();
602  cout << "finished hessian save" << endl;
603  }
604  return; // end of call1
605  call2: /* Line search and Hessian update */
606  if (h.disk_save())
607  {
608  cout << "starting hessian restore" << endl;
609  h.restore();
```

```
610  cout << "finished hessian restore" << endl;
611  }
612  /* restore x_k, g(x_k) and g(x_k+alpha*p_k) */
613  for (i=1; i<=n; i++)
614  {
615  x.elem(i)=xsave.elem(i); //x_k
616  w.elem(i)=g.elem(i); //g(x_k+alpha*p_k)
617  g.elem(i)=gsave.elem(i); //g(x_k)
618  }
619  fy = f;
620  f = fsave; /* now fy is a new function value, f is the old one */
621  ireturn=-1;
622  /* remember current best function value, gradient and parameter values */
623  if (fy <= fbest)
624  {
625  fbest=fy;
626  for (i=1; i<=n; i++)
627  {
628  x.elem(i)=xx.elem(i);
629  gbest.elem(i)=w.elem(i);
630  }
631  }
632  /* what to do if CTRL-C keys were pressed */
633  if (use_control_c==1)
634  {
635 #if defined(__BORLANDC__)
636  if ( kbhit() || ctlc_flag|| ifn == dcheck_flag )
637 #elif defined(_MSC_VER)
638  //if ( kbhit() || ifn == dcheck_flag )
639  if ( _kbhit() || ctlc_flag || ifn == dcheck_flag )
640 #else
641  if(ctlc_flag || ifn == dcheck_flag )
642 #endif
643  {
644  int c=0;
645  if (ifn != dcheck_flag)
646  {
647  #if defined(__DJGPP__)
648  c = toupper(getxkey());
649  #else
650  c = toupper(getch());
651  #endif
652  }
653  else
654  c='C';
655  if ( c == 'C')
```

```
{
for (i=1; i<=n; i++)
{
x.elem(i)=xx.elem(i);
}
ireturn = 3;
derch(f, x , w, n, ireturn);
return;
}
else if(c=='S')
{
//set convergence criteria to something high to stop now
crit=100000.0;
return;
}
else
{
if ( c == 'Q'|| c == 'N')
{
quit_flag=c;
goto label92;
}
else
{
quit_flag=0;
}
}
}
}
if (quit_flag)
{
if (quit_flag==1) quit_flag='Q';
if (quit_flag==2) quit_flag='N';
goto label92;
}
/* Otherwise, continue */
/* Note, icc seems to be unused */
icc+=1;
if( icc >= 5)
icc=0;
/* ic - counter of calls without x update */
ic++;
if( ic >imax)
{
if (iprint>0)
{
```

```
 if (ad_printf)
 (*ad_printf)(" ic > imax in fminim is answer attained ?\n");
 fmmdisp(x, g, n, this->scroll_flag,noprintx);
 }
 ihflag=1;
 ihang=1;
 goto label92;
 }
 /* new function value was passed to fmin, will increment ifn */
 ifn++;

 gys=0.0;

 /* gys = transpose(p_k) * df(x_k+alpha_k*p_k) */
 for (i=1; i<= n; i++)
 gys+=w.elem(i)*w.elem(is+i);

 /* bad step; unless modified by the user, fringe default = 0 */
 if(fy>f+fringe)
 {
 goto label40; /* backtrack */
 }
 /* Otherwise verify if the search step length satisfies
 strong Wolfe condition */
 if(fabs(gys/gso)<=.9)
 goto label50; /* proceed to Hessian update */
/* or slightly modified constant in Wolfe condition for the number of
 calls > 4 */
 if(fabs(gys/gso)<=.95 && ic > 4)
 goto label50; /* proceed to Hessian update */
 if(gys>0.0) /* not a descent direction */
 goto label40; /* backtrack */
 tot+=alpha;
 z=10.0;
 if(gs<gys)
 z=gys/(gs-gys);
 if(z>10.0)
 z=10.0;
 /* increase step length */
 alpha=alpha*z;
 if (alpha == 0.)
 {
 ialph=1;
 #ifdef __ZTC__
 if (ireturn <= 0)
 {
```

```
748  disp_close();
749  }
750  #endif
751  return;
752  }
753  f=fy;
754  gs=gys;
755  goto label30; /* update x with new alpha */
756 label40: /* new step is not acceptable, stepping back and
757  start backtracking along the Newton direction
758  trying a smaller value of alpha */
759  for (i=1;i<=n;i++)
760  xx.elem(i)-=alpha*w.elem(is+i);
761  if (alpha == 0.)
762  {
763  ialph=1;
764  return;
765  }
766  /* compute new alpha */
767  z=3.0*(f-fy)/alpha+gys+gs;
768  zz=dafsqrt(z*z-gs*gys);
769  z=1.0-(gys+zz-z)/(2.0*zz+gys-gs);
770  if (fabs(fy-1.e+95) < 1.e-66)
771  {
772  alpha*=.001;
773  }
774  else
775  {
776  if (z>10.0)
777  {
778  cout << "large z" << z << endl;
779  z=10.0;
780  }
781  alpha=alpha*z;
782  }
783  if (alpha == 0.)
784  {
785  ialph=1;
786  if (ialph)
787  {
788  if (ad_printf) (*ad_printf)("\nFunction minimizer: Step size"
789  " too small -- ialph=1");
790  }
791  return;
792  }
793  goto label30; /* update x with new alpha */
```

```
794 label50: /* compute Hessian updating terms */
795  alpha+=tot;
796  f=fy;
797  df-=f;
798  dgs=gys-gso;
799  xxlink=1;
800  if(dgs+alpha*gso>0.0)
801  goto label52;
802  for (i=1;i<=n;i++)
803  w.elem(iu+i)=w.elem(i)-g.elem(i);
804  /* now w(n+1,2n) = df(x_k+alpha_k*p_k)-df(x_k) */
805  sig=1.0/(alpha*dgs);
806  goto label70;
807 label52: /* compute Hessian updating terms */
808  zz=alpha/(dgs-alpha*gso);
809  z=dgs*zz-1.0;
810  for (i=1;i<=n;i++)
811  w.elem(iu+i)=z*g.elem(i)+w.elem(i);
812  sig=1.0/(zz*dgs*dgs);
813  goto label70;
814 label60: /* compute Hessian updating terms */
815  xxlink=2;
816  for (i=1;i<=n;i++)
817  w.elem(iu+i)=g.elem(i);
818  if(dgs+alpha*gso>0.0)
819  goto label62;
820  sig=1.0/gso;
821  goto label70;
822 label62: /* compute Hessian updating terms */
823  sig=-zz;
824  goto label70;
825 label65: /* save in g the gradient df(x_k+alpha*p_k) */
826  for (i=1;i<=n;i++)
827  g.elem(i)=w.elem(i);
828  goto label20; //convergence check
829 label70: // Hessian update
830  w.elem(iv+1)=w.elem(iu+1);
831
832 #ifdef DEBUG
833  cout << __FILE__ << ':' << __LINE__ << ' '
834  << fmintime.get_elapsed_time_and_reset() << endl;
835 #endif
836
837  for (i=2;i<=n;i++)
838  {
839  i1=i-1;
```

```
840  z=w.elem(iu+i);
841  double * pd=&(h.elem(i,1));
842  double * pw=&(w.elem(iv+1));
843  for (j=1;j<=i1;j++)
844  {
845  z-=*pd++ * *pw++;
846  }
847  w.elem(iv+i)=z;
848  }
849
850 #ifdef DEBUG
851  cout << __FILE__ << ':' << __LINE__ << ' '
852  << fmintime.get_elapsed_time_and_reset() << endl;
853 #endif
854
855  for (i=1;i<=n;i++)
856  { /* BFGS updating formula */
857  z=h.elem(i,i)+sig*w.elem(iv+i)*w.elem(iv+i);
858  if(z <= 0.0)
859  z=dmin;
860  if(z<dmin)
861  dmin=z;
862  h.elem(i,i)=z;
863  w.elem(ib+i)=w.elem(iv+i)*sig/z;
864  sig-=w.elem(ib+i)*w.elem(ib+i)*z;
865  }
866  for (j=2;j<=n;j++)
867  {
868  double * pd=&(h.elem(j,1));
869  double * qd=&(w.elem(iu+j));
870  double * rd=&(w.elem(iv+1));
871  for (i=1;i<j;i++)
872  {
873  *qd-=*pd * *rd++;
874  *pd++ +=w.elem(ib+i)* *qd;
875  }
876  }
877  if (xxlink == 1) goto label60;
878  if (xxlink == 2) goto label65;
879 /*label90:*/
880  for (i=1;i<=n;i++)
881  g.elem(i)=w.elem(i);
882 label92: /* Exit with error */
883  if (iprint>0)
884  {
885  if (ialph)
```

```
 {
 if (ad_printf)
 (*ad_printf)("\nFunction minimizer: Step size too small -- ialph=1");
 }
 if (ihang == 1)
 {
 if (ad_printf)
 (*ad_printf)(
 "Function minimizer not making progress ... is minimum attained?\n");
#if defined(USE_DDOUBLE)
#undef double
 if (ad_printf)
 (*ad_printf)("Minimprove criterion = %12.4le\n",double(min_improve));
#define double dd_real
#else
 if (ad_printf)
 (*ad_printf)("Minimprove criterion = %12.4le\n",min_improve);
#endif
 }
 }
 if(iexit == 2)
 {
 if (iprint>0)
 {
 if (ad_printf)
 (*ad_printf)("*** grad transpose times delta x greater >= 0\n"
 " --- convergence critera may be too strict\n");
 ireturn=-1;
 }
 }
#if defined (_MSC_VER) && !defined (__WAT32__)
 if (scroll_flag == 0) clrscr();
#endif
 if (maxfn_flag == 1)
 {
 if (iprint>0)
 {
 if (ad_printf)
 (*ad_printf)("Maximum number of function evaluations exceeded");
 }
 }
 if (iprint>0)
 {
 if (quit_flag == 'Q')
 if (ad_printf) (*ad_printf)("User initiated interrupt");
```

```
 }
 if(iprint == 0) goto label777;
 if (ad_printf) (*ad_printf)("\n - final statistics:\n");
 if (ad_printf)
 (*ad_printf)("%d variables; iteration %ld; function evaluation %ld\n",
 n, itn, ifn);
#if defined(USE_DDOUBLE)
#undef double
 if (ad_printf)
 (*ad_printf)(
 "Function value %12.4le; maximum gradient component mag %12.4le\n",
 double(f), double(gmax));
 if (ad_printf)
 (*ad_printf)(
 "Exit code = %ld; converg criter %12.4le\n",iexit,double(crit));
#define double dd_real
#else
 if (ad_printf)
 (*ad_printf)(
 "Function value %12.4le; maximum gradient component mag %12.4le\n",
 f, gmax);
 if (ad_printf)
 (*ad_printf)(
 "Exit code = %ld; converg criter %12.4le\n",iexit,crit);
#endif
 fmmdisp(x, g, n, this->scroll_flag,noprintx);
label777: /* Printing final Hessian approximation */
 if (ireturn <= 0)
 #ifdef DIAG
 if (ad_printf) (*ad_printf)("Final values of h in fmin:\n");
 cout << h << "\n";
 #endif
 #ifdef __ZTC__
 {
 disp_close();
 }
 #endif
 return;
label7000:/* Printing Initial statistics */
 if (iprint>0)
 {
#if defined (_MSC_VER) && !defined (__WAT32__)
 if (!scroll_flag) clrscr();
#endif
 if (ad_printf) (*ad_printf)("\nInitial statistics: ");
```

```
 }
 goto label7003;
label7010:/* Printing Intermediate statistics */
 if (iprint>0)
 {
#if defined (_MSC_VER) && !defined (__WAT32__)
 if (!scroll_flag) clrscr();
#endif
 if (ad_printf) (*ad_printf)("\nIntermediate statistics: ");
 }
 llog=0;
 goto label7003;
label7020:/* Exis because Hessian is not positive definite */
 if (iprint > 0)
 {
 if (ad_printf) (*ad_printf)("*** hessian not positive definite\n");
 }
#ifdef __ZTC__
 if (ireturn <= 0)
 {
 disp_close();
 }
#endif
}
double dafsqrt(double x)
{
 return x > 0 ? sqrt(x) : 0.0;
}
```