Optimization Practical Exercise
Sommersemester 2023

Donnermair Maximilian @students.jku.at
Fromherz Jakob @students.jku.at
Haslhofer Eva-Maria k12007773@students.jku.at
Scharnreitner Franz @students.jku.at
Weiss Hannah k12021111@students.jku.at

12 May 2023

1 Exercise 1

1.1 Linesearch Algorithm with Wolfe-Powell Condition

```
function [x_new,f_new,g_new, exit_flag, alpha, eval] = LineSearch(f, x_old, f_old, g_old, p, phi_min, alpha_st)
    %Linesearch algorithm with Wolfe-Powell Condition (Algorithm 4.2)

    %Setting of parameters as described in the scriptum
    tau =  0.1;
    tau1 = 0.1;
    tau2 = 0.6;
    xi1 = 1;
    xi2 = 10;
    mu1 = 1/4;
    mu2 = 0.9;
    sigma = 0.91; %sigma greater than mu1

    %if only 6 arguments are given
    if nargin == 6
      alpha_st = 1;
    end

    alpha_l = 0;
    phi_l = f_old;  %phi_l=phi(0)
    x_old=x_old(:); % x_old always considered as column vector
    dphi_l = dot(g_old,p);
    exit_flag = 0;
    flag = true;  % means that alpha_r is infinity
    alpha_r = 10^30; %alpha_r should be very large

    alpha_tilde = 0; %initialization of alpha_tilde
    eval = 0; %number of evaluations needed

    while abs(alpha_r - alpha_l) > 10^(-15)
        %evaluation of function, gradient and exit_flag
        x_temp = x_old + alpha_st * p;
        [f_temp, g_temp, exit_flag] = f(x_temp);
        eval = eval + 1;

        %function could not be evaluated (alpha_hat not in omega_prime)
        if exit_flag ~= 0
            alpha_r = alpha_st;
            alpha_st = alpha_l + tau1 * (alpha_r - alpha_l);

        %function could be evaluated (alpha_hat in omega_prime)
        else
            phi_hat = f_temp;

            %if function smaller that phi_min, function is declared as
            %unbounded
            if phi_hat < phi_min
                exit_flag = 2; % output is not minimum
                fprintf("Error, unbounded function")
```

```
                alpha = alpha_st;
                x_new = x_temp;
                f_new = f_temp;
                return
            end
            %setting alpha_r and calculating alpha_st
            if phi_hat > (f_old + mu1 * alpha_st * dphi_l)
                flag = false; %alpha_r is not infinity
                alpha_r = alpha_st;
                length = alpha_r - alpha_l;
                c = (phi_hat - phi_l  - dphi_l*length)/ (length^2);
                alpha_tilde = alpha_l - dphi_l/(2*c);
                alpha_st = min(max(alpha_l + tau * length , alpha_tilde),alpha_r - tau * length);

            else
                dphi_hat = dot(g_temp,p); %derivative of phi_hat
                %calculating alpha_tilde
                if dphi_hat < sigma * dphi_l
                    if flag
                        if dphi_l/dphi_hat > (1 + xi2)/xi2
                            alpha_tilde = alpha_st + (alpha_st - alpha_l) * max(dphi_hat/(dphi_l - dphi_hat),xi1);
                        else
                            alpha_tilde = alpha_st + xi2 * (alpha_st - alpha_l);
                        end
                    else
                        if dphi_l/dphi_hat>1+(alpha_st-alpha_l)/(tau2*(alpha_r-alpha_st))
                            alpha_tilde = alpha_st+max((alpha_st-alpha_l)*dphi_hat/(dphi_l-dphi_hat),tau1*(alpha_r-alpha_st));
                        else
                            alpha_tilde=alpha_st+tau2*(alpha_r-alpha_st);
                        end
                    end
                % setting alpha_l, phi_l, derivative of phi_l and alpha_st
                alpha_l = alpha_st;
                phi_l=phi_hat;
                dphi_l = dphi_hat;
                alpha_st=alpha_tilde;

                %returning from algorithm
                else
                    alpha = alpha_st;
                    x_new = x_old+alpha*p;
                    [f_new,g_new,exit_flag] = f(x_new);
                    eval = eval+1;
                    return
                end
            end
        end
    end
    % final return (if there was no return before)
    alpha = alpha_st;
```

```
    x_new = x_old+alpha*p;
    [f_new,g_new,exit_flag] = f(x_new);
    eval = eval+1;
    end
```

src/LineSearch.m

1.2 Method of steepest descent

```
function [x,f_val,g,exit_flag, iter, evals] = SteepestDescent (f,
    x0, phi_min,eps,itmax, typ_f,typ_x)
  %Steepest Descent Algorithm for testing LineSearch

  %for varible input arguments
  if nargin < 7
    typ_x(1:length(x0)) = 10^-4;
  end

  if nargin < 6
    typ_f = 10^-4;
  end

  if nargin < 5
    itmax = 1000;
  end

  if nargin < 4
    eps = 10^-6;
  end

  if nargin < 3
    phi_min = -10^30;
  end

  %starting value for iteration, function value, gradient,
    exit_flag
  xk = x0;
  [fk,gk,exit_flag] = f(x0);
  evals = 1;

  %iterations of steepest descent
  for iter = 1:itmax

    %termination condition fullfilled (relative gradient less than
    tolerance)
    if max(abs(gk) .* typ_x / typ_f) <= eps
      x = xk;
      f_val = fk;
      g = gk;
      return
    end

    %calling LineSearch
    [xk,fk,gk,exit_flag,~,eval_temp] = LineSearch(f,xk,fk,gk,-gk,
    phi_min);
```

```
    typ_f = max(typ_f,abs(fk));
    typ_x = max(typ_x,abs(xk));
    evals = evals + eval_temp; %updating number of evaluations
  end

  %Setting output, if termination condition is not fullfilled and
    maximal
  %number of iterations reached
  x = xk;
  f_val = fk;
  g = gk;
  exit_flag = 1;

end
```

src/SteepestDescent.m

1.3 Testfunctions a) to d)

```
function [f_val, g, exit_flag] = f_a(x)
    exit_flag=0;
    x1=x(1);
    x2=x(2);
    f_val = 0.5*(x1+x2)^2 + 0.05*(x1-x2)^2;
    g = [1.1*x1 + 0.9*x2; 0.9*x1 + 1.1*x2];
end
```

src/f_a.m

```
function [f_val, g, exit_flag] = f_b(x)
    exit_flag=0;
    x1=x(1);
    x2=x(2);
    f_val = 100*(x2-x1^2)^2 + (1-x1)^2;
    g = [400*x1*(x1^2-x2) + 2*(x1-1); 200*(x2-x1^2)];
end
```

src/f_b.m

```
function [f_val, g, exit_flag] = f_c(x)
    exit_flag=0;
    n = length(x);
    if min(x)<=0
        f_val = 10^30; %dummy values
        g = ones(n,1); %dummy values
        exit_flag = 1;
        return
    end

    x=x(:); %transform to column vector
    %(1:n)', x, log(x) column vectors
    f_val = sum((1:n)'.*x.*log(x)) + 1/sum(x);
    g = zeros(n,1); %column vector
    for i=1:n
```

```
        g(i)=i*(log(x(i))+1) - 1/sum(x)^2;
    end
end
```

src/f_c.m

```
function [f_val, g, exit_flag] = f_d(x)
    exit_flag=0;
    n = length(x);

    x=x(:);
    v=(1:n)';
    A=(1/(100*n))*eye(n)+hilb(n);
    xAx = x'*A*x;
    f_val = (xAx)^2-(x'*A*v)^2;
    g = 4*xAx*(A*x)-2*(x'*A*v)*A*v;
end
```

src/f_d.m

1.4 Testscript

```
diary test1.txt
diary on
disp("Funktion f_a test")

[x_val,f_val,g_val,~,iter,evals] = SteepestDescent(@f_a,[10,-3]);
displayVals(x_val,f_val,g_val,iter,evals);

disp("--------------------------------------")

disp("Funktion f_b test")

[x_val,f_val,g_val,~,iter,evals]= SteepestDescent(@f_b
    ,[-1,2],-10^30,10^-6,5000);
    displayVals(x_val,f_val,g_val,iter,evals);

disp("--------------------------------------")

disp("Funktion f_c test");
for j = [10 100 1000]
    fprintf("For n = %d \n",j);
    x0 = 5 * ones(1,j);
    [x_val,f_val,g_val,~,iter,evals] = SteepestDescent(@f_c,x0
    ,-10^30,10^-9,10000);
    displayVals(x_val,f_val,g_val,iter,evals);

end


disp("--------------------------------------")

disp("Funktion f_d test");

for j = [10 100 1000]
```

```
    fprintf("For n = %d \n",j);
    x0 = zeros(j,1);
    x0(j) = 10*j;
    [x_val,f_val,g_val,~,iter,evals] = SteepestDescent(@f_d,x0
    ,-10^30,10^-6,10000);
    displayVals(x_val,f_val,g_val,iter,evals);

end

disp("——————————————————————————")


diary off
```

src/testex1.m

With a quick printing function.

```
function retval = displayVals (x_val, f_val,g_val,iter,evals)
    if length(x_val) < 11
        fprintf("Minimum computed at: ");
        x_val


 end

    disp("Minimum at function f value: ");
    f_val

    disp("norm of gradient: ");
    no = norm(g_val);
    no

    fprintf("with %d iterations and %d function evals \n",iter,
    evals);


end
```

src/displayVals.m

We get the output

```

Funktion f_a test
Minimum computed at:
x_val =

   1.0e-05 *

    0.2079
   -0.1852

Minimum at function f value:

f_val =

   7.9817e-13
```

```
norm of gradient:

no =

    6.4204e-07

with 73 iterations and 157 function evals
----------------------------------------
Funktion f_b test
Minimum computed at:
x_val =

     1.0000
     1.0000

Minimum at function f value:

f_val =

    4.3037e-11

norm of gradient:

no =

    6.1783e-06

with 4319 iterations and 19493 function evals
----------------------------------------
Funktion f_c test
For n = 10
Minimum computed at:
x_val =

     0.3949
     0.3811
     0.3767
     0.3745
     0.3731
     0.3722
     0.3716
     0.3712
     0.3708
     0.3705

Minimum at function f value:

f_val =

   -19.9644

norm of gradient:

no =

    7.1634e-08
```

```

with 95 iterations and 328 function evals
For n = 100
Minimum at function f value:

f_val =

  -1.8578e+03

norm of gradient:

no =

   2.8699e-06

with 797 iterations and 3582 function evals
For n = 1000
Minimum at function f value:

f_val =

  -1.8412e+05

norm of gradient:

no =

   3.9574e-04

with 5535 iterations and 30423 function evals
-----------------------------------
Funktion f_d test
For n = 10
Minimum computed at:
x_val =

    0.7074
    1.4125
    2.1185
    2.8354
    3.5448
    4.2462
    4.9424
    5.6360
    6.3286
    7.1201

Minimum at function f value:

f_val =

  -1.6770e+04

norm of gradient:

no =

```

```
    0.0436

with 10000 iterations and 50003 function evals
For n = 100
Minimum at function f value:

f_val =

  -1.0983e+10

norm of gradient:

no =

   3.6520e+04

with 10000 iterations and 80003 function evals
For n = 1000
Minimum at function f value:

f_val =

  -1.0513e+16

norm of gradient:

no =

   7.1346e+07

with 10000 iterations and 110001 function evals
-----------------------------------
```

src/test1.txt

1.5 Interpretation

We can see, that the SteepestDescent works for f_a relatively well, while to get a relatively good result for f_b we already need > 4000 iterations. f_c works quite well, altough the number of iterations needed grows fast with increase in dimension size. The algorithm fails to work for f_d however for big dimensions.

2 Exercise 2

2.1 Testfunctions a) to d) with Hessian

```
function [f_val, g, H] = f_aH(x)
    x1=x(1);
    x2=x(2);
    f_val = 0.5*(x1+x2)^2 + 0.05*(x1-x2)^2;
    g = [1.1*x1+0.9*x2;0.9*x1+1.1*x2];
    if nargout > 2
        H = [1.1,0.9;0.9,1.1];
```

```
    end
end
```

src/f_aH.m

```
function [f_val, g, H] = f_bH(x)
    x1=x(1);
    x2=x(2);
    f_val = 100*(x2-x1^2)^2 + (1-x1)^2;
    g = [400*x1*(x1^2-x2) + 2*(x1-1); 200*(x2-x1^2)];
    if nargout > 2
        H = [1200*x1^2 - 400*x2 + 2, -400*x1; -400*x1, 200];
    end
end
```

src/f_bH.m

```
function [f_val, g, H] = f_cH(x)
    n = length(x);
    if min(x)<=0
        f_val = 10^30; %dummy values
        g = ones(n,1); %dummy values
        return
    end

    x=x(:); %transform to column vector
    %(1:n)', x, log(x) column vectors
    f_val = sum((1:n)'.*x.*log(x)) + 1/sum(x);
    g = zeros(n,1); %column vector
    for i=1:n
        g(i) = i*(reallog(x(i))+1) - 1/sum(x)^2;
    end
    if nargout > 2
        h = 2/sum(x)^3;
        H = h*ones(n,n) + diag((1:n)'./x,0);
    end
end
```

src/f_cH.m

```
function [f_val, g, H] = f_dH(x)
    n = length(x);

    x=x(:);
    v=(1:n)';
    A=(1/(100*n))*eye(n)+hilb(n);
    xAx = x'*A*x;
    f_val = (xAx)^2-(x'*A*v)^2;

    if nargout > 1
        g = 4*xAx*(A*x)-2*(x'*A*v)*A*v;
        if nargout >2
            H = 4*(2*A*x+(A*x)'+x'*A*x*A)-2*A*v*(A*v)';
        end
    end

```

```
end
```

src/f_dH.m

2.2 Tests in Exercise 2

```
dfile1 = 'Test1.txt';
if exist(dfile1, 'file') ; delete(dfile1); end
diary(dfile1)
diary on

options1= optimset('LargeScale','off','GradObj','on');
options2= optimset('LargeScale','on','GradObj','on','Hessian','off'
    );
options3= optimset('Algorithm','trust-region','LargeScale','on','
    GradObj','on','Hessian','on');
options = [options1,options2,options3];

%(a)
disp
    ("==========================================================")
disp("(A)")
disp
    ("==========================================================")
for i=1:length(options)
    disp("Testing objective function f_a with options: ")
    disp(options(i));

    s_a = fminunc(@f_aH,[10,-3],options(i));
    disp("Calculated minimizer: ");
    s_a
    [fm,g] = f_aH(s_a);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));
end


disp
    ("==========================================================")
disp("(B)")
disp
    ("==========================================================")
for i=1:length(options)
    disp("Testing objective function f_b with options: ")
    disp(options(i));

    s_b = fminunc(@f_bH,[-1,2],options(i));
    disp("Calculated minimizer: ");
    s_b

    [fm,g] = f_bH(s_b);
```

```
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));
end


disp
    ("====================================================================================================")
disp("C")
disp
    ("====================================================================================================")

for i=1:length(options)
    disp("Testing objective function f_c with options: ")
    disp(options(i));

    disp
    ("====================================================================================================")

    s_c1 = fminunc(@f_cH,5*ones(1,10),options(i));
    disp("Calculated minimizer: ");
    s_c1
    [fm,g] = f_cH(s_c1);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));

    s_c2 = fminunc(@f_cH,5*ones(1,100),options(i));
    [fm,g] = f_cH(s_c2);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));

    s_c3 = fminunc(@f_cH,5*ones(1,1000),options(i));
    [fm,g] = f_cH(s_c3);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));
end



disp
    ("====================================================================================================")
disp("(D)")
disp
    ("====================================================================================================")

for i=1:length(options)
    disp("Testing objective function f_d with options: ")
    disp(options(i));

    disp
    ("====================================================================================================")
```

```
    s_d1 = fminunc(@f_dH,[zeros(1,9),100],options(i));
    disp("Calculated minimizer: ")
    s_d1

    [fm,g] = f_dH(s_d1);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));

    disp
   ("==================================================================")


    s_d2 = fminunc(@f_dH,[zeros(1,99),1000],options(i));
    [fm,g] = f_dH(s_d2);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));

    disp
   ("==================================================================")


    s_d3 = fminunc(@f_dH,[zeros(1,999),10000],options(i));
    [fm,g] = f_dH(s_d3);
    disp("Calculated Minimum value: " + fm);
    disp("Norm of gradient at calculated Minimum: " + norm(g));

    disp
   ("==================================================================")

end


diary off
```

src/Prog1Ex2.m

2.3 Interpretation

3 Exercise 3

3.1 Interpretation

4 Exercise 4

4.1 Source Code

```
options1 = optimoptions('fminunc','GradObj','off');
options2 = optimset('GradObj','off');

f = @(x) x(1)+10*max(x(1)^2+2*x(2)^2-1,0);
x0=[1,1];
s1 = fminunc(f,x0,options1)
s2 = fminsearch(f,x0,options2)
```

```
e1 = norm(s1-[-1,0])
e2 = norm(s2-[-1,0])
```

src/Prog1Ex4.m

```
>> Prog1Ex4

Local minimum possible.

fminunc stopped because it cannot decrease the objective function
along the current search direction.

<stopping criteria details>

s1 =

   -0.9997   -0.0168


s2 =

   -1.0000    0.0000


e1 =

    0.0168


e2 =

   4.3007e-05
```

src/test4_1.txt

4.2 Solution of `fminsearch`

When applying `fminsearch` to the problem considering the given non-continuously differentiable function

$$\min_{x\in\mathbb{R}^2} x_1 + 10\max\{x_1^2 + 2x_2^2 - 1, 0\}$$

we get that $x = (x_1, x_2) \approx (-0.9997, -0.0168)$ solves the above equation where the exact solution should correspond to the vector $\overline{x} = (x_1, x_2) = (-1, 0)$. This implies a numerical error of $\|x - \overline{x}\| \approx 0.0168$ as can be obtained by the `MATLAB` source code in section 4.1.

4.3 Solution of `fminunc`

If instead of `fminsearch` the `MATLAB` command `fminunc` is applied to the same problem as in section 4.2 we get the result of $x = (x_1, x_2) \approx (-1.0000, 0.0000)$.

Consequently, we also get a numerical error much smaller in size and given by $\|x - \overline{x}\| \approx 4.3007 \cdot 10^{-5}$.

4.4 Interpretation

In the `MATLAB` output it states that `fminsearch` stopped because it cannot decrease the objective function along the current search direction any further. Though, if stopping criteria details are displayed we get the following additional information.

```
Optimization stopped because the objective function cannot be decreased in the
current search direction. Either the predicted change in the objective function,
or the line search interval is less than eps.
```

src/test4_2.txt

Consequently, it must be the case that the simplex search method of `fminsearch` which does not make use of numerical or analytic gradients as the line search algorithm in `fminunc` is not appropriate for the considered problem. On the opposite, `fminunc` estimates according gradients using finite differences and therefore provides an adequate line search interval and ultimately a more reliable result.