Non Linear Programming: Homework 9

vishvAs vAsuki

April 14, 2010

1 Bisection implementation

[**Incomplete**]

2 Newton's method

2.1 Results

See 2.1.

2.2 Code

2.2.1 Experiment code

```
function newtonMethodExperiment()
    [secantScaler, shrinkageFactor, A] = getData();
    [m, n] = size(A);
    cutoffNewton = 10^-5;
    cutoffGradientDescent = 10^-1;

    objFnHandle = @(x)objFn(x, A);
    gradientFnHandle = @(x)gradientFn(x, A);
    hessianFnHandle = @(x)hessianFn(x, A);
    domainMembershipFnHandle = @(x)domainMembershipFn(x,
       A);
    stepSizeFinderFnHandle = @(x, searchDirection)
       optimization.LineSearch.backtrackingSearchWrapper(
       x, searchDirection, objFnHandle, gradientFnHandle,
        secantScaler, shrinkageFactor,
       domainMembershipFnHandle);

    x_init = zeros(n,1);
    x_init(1) = 0.0001;
```

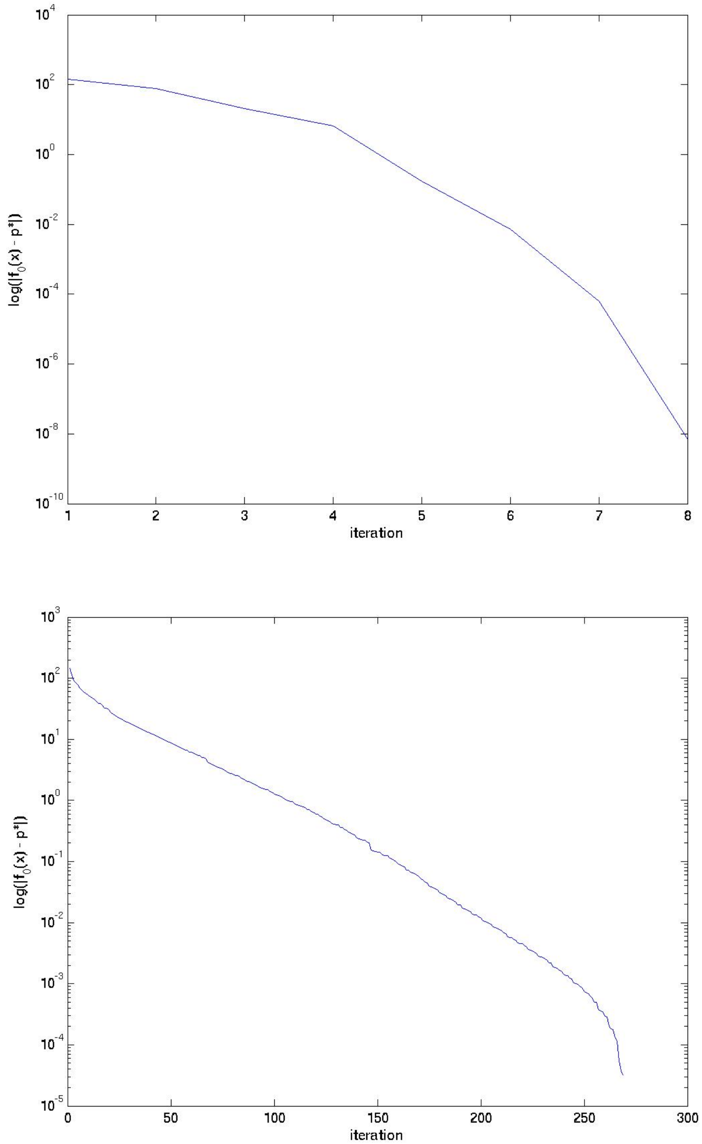

Figure 1: Residual graphs: newton method and gradient descent. Note that lower accuracy was attempted with gradient descent.

```
    [x_opt, x_iterates] = optimization.DescentMethods.
        steepestDescentHessian(x_init, objFnHandle,
        gradientFnHandle, hessianFnHandle,
        stepSizeFinderFnHandle, cutoffNewton);

    fig = optimization.DescentMethods.plotError(x_opt,
        x_iterates, objFnHandle);
    saveas(fig, '/v/filer4b/v20q001/vvasuki/vishvas/work/
        optimization/hw/hw9/code/residualNewton.jpg');
    close all;

    [x_opt, x_iterates] = optimization.DescentMethods.
        gradientDescent(x_init, objFnHandle,
        gradientFnHandle, stepSizeFinderFnHandle,
        cutoffGradientDescent);
    fig = optimization.DescentMethods.plotError(x_opt,
        x_iterates, objFnHandle);
    saveas(fig, '/v/filer4b/v20q001/vvasuki/vishvas/work/
        optimization/hw/hw9/code/residualGradientDescent.
        jpg');
    close all;



end

function [secantScaler, shrinkageFactor, A] = getData()
    n = 100;
    m = 200;
    secantScaler = 0.01;
    shrinkageFactor = 0.5;
    randn('state',1);
    A = randn(m,n);
end

function value = fn1_Ax(x, A)
    [m, n] = size(A);
    value = ones(m, 1) - A*x;
end

function value = fn1_xx(x)
    n = numel(x);
    value = ones(n, 1) - x.^2;
end
```

```
function objValue = objFn(x, A)
    objValue = -sum(log(fn1_Ax(x, A))) - sum(log(fn1_xx(x
        )));
end


function gradient = gradientFn(x, A)
    v1 = fn1_Ax(x, A);
    v2 = fn1_xx(x);
    gradient = A'*(v1.^-1) + 2*diag(x)*(v2.^-1);
end

function Hessian = hessianFn(x, A)
    [m, n] = size(A);
    v1 = fn1_Ax(x, A);
    v2 = fn1_xx(x);
    D1 = diag(v1.^-2);
%       keyboard
    D2 = diag(2*diag(x.^2)*(v2.^-2) + (v2.^-1));
    Hessian = A'*D1*A + 2*D2;
end

function bInDomain = domainMembershipFn(x, A)
    bInDomain = all(A*x < 1) &&  all(abs(x)<1);
end
```

2.2.2 Optimization code

```
classdef DescentMethods
methods(Static=true)
function [x_opt, x_iterates] = descentAlg(x_init,
   searchDirectionFinderFn, stepSizeFinderFn,
   stoppingCriterionFn)
%       Input:
%           x_init
%           searchDirectionFinderFn
%           setpSizeFinderFn
%           stoppingCriterionFn
%       Output:
%           x_opt.
%           objectiveGaps : A vector of gaps from the
   optimum at each iteration.
    x_opt = x_init;
    n = numel(x_opt);
    x_iterates = zeros(1,n);
    iteration = 1;
```

```
    while(true)
        x_iterates(iteration, :) = x_opt';
        searchDirection = searchDirectionFinderFn(x_opt);
        stepSize = stepSizeFinderFn(x_opt,
            searchDirection);
        x_opt = x_opt + stepSize*searchDirection;
        [bStop] = stoppingCriterionFn(x_opt,
            searchDirection);
        if(bStop)
            break;
        end
        iteration = iteration+1;
    end
end

function [x_opt, x_iterates] = steepestDescentHessian(
    x_init, objFn, gradientFn, hessianFn, stepSizeFinderFn
    , cutoff)
%   The newton method
    searchDirectionFinderFn = @(x)optimization.
        DescentMethods.searchDirection_2ndOrderApproxMin(x
        , gradientFn, hessianFn);
    stoppingCriterionFn = @(x, searchDirection)
        optimization.DescentMethods.
        stoppingCriterionNewton(x, searchDirection, cutoff
        , gradientFn);
    [x_opt, x_iterates] = optimization.DescentMethods.
        descentAlg(x_init, searchDirectionFinderFn,
        stepSizeFinderFn, stoppingCriterionFn);
end

function [x_opt, x_iterates] = gradientDescent(x_init,
    objFn, gradientFn, stepSizeFinderFn, cutoff)
    searchDirectionFinderFn = @(x)(-gradientFn(x));
    stoppingCriterionFn = @(x, searchDirection)(norm(
        gradientFn(x)) < cutoff);
    [x_opt, x_iterates] = optimization.DescentMethods.
        descentAlg(x_init, searchDirectionFinderFn,
        stepSizeFinderFn, stoppingCriterionFn);

end

function searchDirection =
    searchDirection_2ndOrderApproxMin(x, gradientFn,
    hessianFn)
```

```
%       Finds the search direction used in the newton
    method
     searchDirection = - hessianFn(x)\gradientFn(x);
end

function bStop = stoppingCriterionNewton(x,
    searchDirection, cutoff, gradientFn)
     newtonDecrement = sqrt(-gradientFn(x)'*
         searchDirection);
     bStop = (abs(newtonDecrement) < cutoff);
end

function fig = plotError(x_opt, x_iterates, objFn)
     fig = figure();
     numIterations = size(x_iterates, 1);
     iterations = 1:numIterations;
     y = [];
     for iteration = iterations
          y(iteration, 1) = objFn(x_iterates(iteration,:)')
              ;
     end
     y = abs(objFn(x_opt)-y);
     fig = semilogy(iterations, y);
     ylabel('log(|f_0(x) - p*|)');
     xlabel('iteration');
%       keyboard
end

function testClass
     display 'Class definition is ok';
end


end
end
```

2.2.3 Line search code

```
classdef LineSearch
methods(Static=true)
function stepSize = backtrackingSearch(objFnSlice,
    gradient, searchDirection, secantScaler,
    shrinkageFactor, domainMembershipFnSlice)
%       Input:
%           objFnSlice: Function handle. objFnSlice(
    stepSize) = f_0(x + stepSize \change x), where f_0 is
```

```
    the objective of the optimization problem, \change x
    is the search direction.
%           gradient: \gradient f_0(x), a vector.
%           searchDirection: a vector.
%           secantScaler: used to specify the secant used
    in the stopping criterion.
%           shrinkageFactor: used to shrink stepSize
    repeatedly until stopping criterion is satisfied.
%           domainMembershipFnSlice: function handle.
    Checks if, for a given stepSize, x + stepSize \change
    x \in dom(f_0).
%       Output: stepSize, a scalar.
    stepSize = 1;
    while(true)
        is_tInDomain = domainMembershipFnSlice(stepSize);
        if(is_tInDomain && objFnSlice(stepSize) <
            objFnSlice(0) + secantScaler*stepSize*gradient
            '*searchDirection)
            break;
        end
        stepSize = shrinkageFactor*stepSize;
    end
end

function stepSize = backtrackingSearchWrapper(x,
    searchDirection, objFn, gradientFn, secantScaler,
    shrinkageFactor, domainMembershipFn)
    objFnSlice = @(stepSize)objFn(x + stepSize*
        searchDirection);
    gradient = gradientFn(x);
    domainMembershipFnSlice = @(stepSize)
        domainMembershipFn(x + stepSize*searchDirection);
    stepSize = optimization.LineSearch.backtrackingSearch
        (objFnSlice, gradient, searchDirection,
        secantScaler, shrinkageFactor,
        domainMembershipFnSlice);
end

function testClass
    display 'Class definition is ok';
end

end
end
```