Non Linear Programming: Homework 10

vishvAs vAsuki

April 21, 2010

1 Equality constrained entropy maximization

Answers to theoretical questions submitted handwritten.

1.1 Results

$f_0(x_opt) : -3.364994e + 01$.

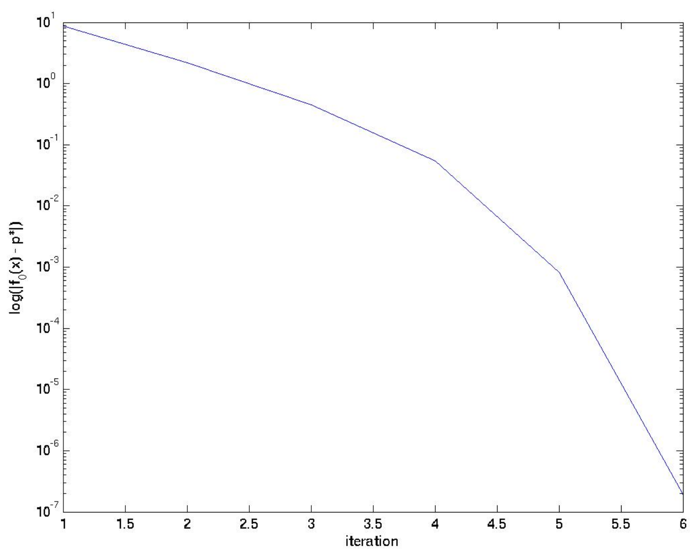

Figure 1: Residual graphs: newton method.

1.2 Code

1.2.1 Implementation comments

I have taken advantage of the structure in the Hessian to ensure that only p*p matrices are inverted. I have also verified the Newton method implementations by observing quadratic convergence.

Completed coding infeasible start newton method, but have not tried it or debugged it.

1.2.2 Experiment code

```
function eqConstrainedEntropyMinimization()
    [A, b, x_init] = getData();
    secantScaler = 0.01;
    shrinkageFactor = 0.5;
    cutoff = 10^-5;
    LOG_PATH = '/v/filer4b/v20q001/vvasuki/vishvas/work/
        optimization/hw/hw10/code/';

    objFnHandle = @objFn;
    gradientFnHandle = @gradientFn;
    hessianFnHandle = @hessianFn;
    invHessianFnHandle = @invHessianFn;

%       Find the search direction without taking advantage
     of structure in H.
    searchDirectionFinderFn = @(x)optimization.
        DescentMethods.searchDirection_2ndOrderApproxMinEq
        (x, gradientFnHandle, invHessianFnHandle, A, b);
    domainMembershipFnHandle = @domainMembershipFn;

%       Find the search direction WHILE taking advantage
    of structure in H.
    searchDirectionFinderFnSmart = @(x)optimization.
        DescentMethods.
        searchDirection_2ndOrderApproxMinEq_invH(x,
        gradientFnHandle, invHessianFnHandle, A, b);

    stepSizeFinderFnHandle = @(x, searchDirection)
        optimization.LineSearch.backtrackingSearchWrapper(
        x, searchDirection, objFnHandle, gradientFnHandle,
         secantScaler, shrinkageFactor,
        domainMembershipFnHandle);
```

```
    [x_opt, x_iterates] = optimization.DescentMethods.
        steepestDescentHessianEq(x_init, objFnHandle,
        gradientFnHandle, searchDirectionFinderFnSmart,
        stepSizeFinderFnHandle, cutoff);

    fprintf(1, 'f_0(x_opt): %d \n', objFn(x_opt));
    fig = optimization.DescentMethods.plotError(x_opt,
        x_iterates, objFnHandle);
    saveas(fig, [LOG_PATH 'residualNewtonEq.jpg']);
    close all;

    stepSizeFinderFnHandle = @(x, searchDirection,
        lagrangeMultiplier)
        backtrackingSearchWrapperEq(x, lagrangeMultiplier,
         searchDirection,
        lagrangeMultiplierSearchDirection,
        gradientFnHandle, secantScaler, shrinkageFactor,
        domainMembershipFnHandle);



    display 'svAgataM! Ready for inspection!';
    keyboard
end

function [A, b, x_init] = getData()
    n=100; p=30;
    rand('state',0);
    randn('state',0);
    A = randn(p,n);
    x_init = rand(n,1);
    b = A * x_init;
end

function objValue = objFn(x)
%       Want to define x_i log x_i = 0
    y = log(x);
    y(y == -Inf) = 0;
    objValue = sum(diag(x)*y);
end

function gradient = gradientFn(x)
    n = length(x);
    gradient = log(x) + ones(n, 1);
end
```

```
function Hessian = hessianFn(x)
    Hessian = diag(1./x);
end

function Hessian = invHessianFn(x)
    Hessian = diag(x);
end


function bInDomain = domainMembershipFn(x)
    bInDomain = all(x>0);
end
```

1.2.3 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 [searchDirection lagrangeMultiplier] =
    searchDirection_2ndOrderApproxMinEq(x, gradientFn,
    hessianFn, A, b)
%       Finds the search direction used in the newton
    method for equality constrained (Ax = b) convex
    optimization problems.
%       Also works with infeasible x.
     [m, n] = size(A);
    M = [hessianFn(x) A'; A zeros(m, m)];
```

```
    b = [-gradientFn(x); Ax-b];
    searchDirection_with_l = M\b;
    searchDirection = searchDirection_with_l(1:n);
    lagrangeMultiplier = searchDirection_with_l(n+1:end);
end

function searchDirection =
    searchDirection_2ndOrderApproxMinEq_invH(x, gradientFn
    , invHessianFn, A, b)
%       Finds the search direction used in the newton
    method for equality constrained (Ax = b) convex
    optimization problems. Special for easy to invert
    hessians.
%       Also works with infeasible x.
    [m, n] = size(A);
%       Solve [H A; A' 0] [searchDir; w] = [-gradientFn(x)
    ; 0]
    invH = invHessianFn(x);
    gradient = gradientFn(x);
    lagrangeMultiplier = A*invH*A'\(Ax-b -A*invH*gradient
        );
    searchDirection = -invH*(gradient + A' *
        lagrangeMultiplier);
end


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

function bStop = stoppingCriterionNewtonInf(x,
    lagrangeMultiplier, searchDirection, cutoff,
    gradientFn)
    residualFn = @(x, lagrangeMultiplier)[gradientFn(x) +
         A'*lagrangeMultiplier; A*x - b];
    bStop = (norm(residualFn(x, lagrangeMultiplier)) <
        cutoff);
end

function [x_opt, x_iterates] = steepestDescentHessianEq(
    x_init, objFn, gradientFn, searchDirectionFinderFn,
    stepSizeFinderFn, cutoff)
%  The newton method for equality constrained (Ax = b)
```

```
    convex optimization problems.
    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] =
    steepestDescentHessianEqInf(x_init, objFn, gradientFn,
     searchDirectionFinderFn, stepSizeFinderFn, cutoff)
%  The newton method for equality constrained (Ax = b)
    convex optimization problems.
%       Also works with infeasible x.
    stoppingCriterionFn = @(x, searchDirection)
        optimization.DescentMethods.
        stoppingCriterionNewton(x, searchDirection, cutoff
        , gradientFn);
    [x_opt, x_iterates] = optimization.DescentMethods.
        descentAlg(x_init, searchDirectionFinderFn,
        stepSizeFinderFn, stoppingCriterionFnInf);
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
```

1.2.4 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 = backtrackingSearchEq(x,
    lagrangeMultiplier, gradientFn, searchDirection,
    lagrangeMultiplierSearchDirection, secantScaler,
    shrinkageFactor, domainMembershipFnSlice)
%     Backtracking search for equality constrained
    optimization problems.
    stepSize = 1;
    residualFn = @(x, lagrangeMultiplier)[gradientFn(x) +
        A'*lagrangeMultiplier; A*x - b];
```

```
    while(true)
        is_tInDomain = domainMembershipFnSlice(stepSize);
        if(is_tInDomain && residualFn(x + t*
            searchDirection, lagrangeMultiplier + t*
            lagrangeMultiplierSearchDirection) < (1 -
            secantScaler*t) * residualFn(x,
            lagrangeMultiplier))
             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 stepSize = backtrackingSearchWrapperEq(x,
    lagrangeMultiplier, searchDirection,
    lagrangeMultiplierSearchDirection, gradientFn,
    secantScaler, shrinkageFactor, domainMembershipFn)
    domainMembershipFnSlice = @(stepSize)
        domainMembershipFn(x + stepSize*searchDirection);
    stepSize = optimization.LineSearch.
        backtrackingSearchEq(x, lagrangeMultiplier,
        gradientFn, searchDirection,
        lagrangeMultiplierSearchDirection, secantScaler,
        shrinkageFactor, domainMembershipFnSlice);
end

function testClass
    display 'Class definition is ok';
end

end
```

end