Non Linear Programming: Exam 2

vishvAs vAsuki

May 13, 2010

Remark. If I missed including any piece of code I should have included, please email and get it from me.

1 Portfolio optimization

1.1 a

Theoretical part submitted handwritten.

Figures are in 1.1.

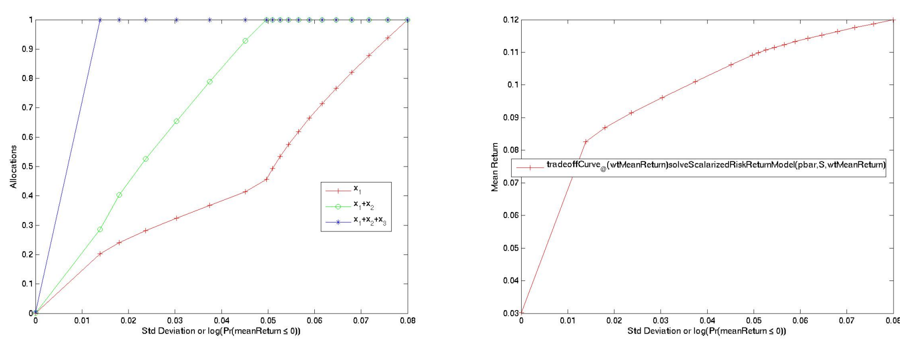

Figure 1: Area plot and Tradeoff curve

1.2 b

Theoretical part submitted handwritten.

Figures are in 1.2.

1.3 c

Theoretical part submitted handwritten.

Figures are in 1.3.

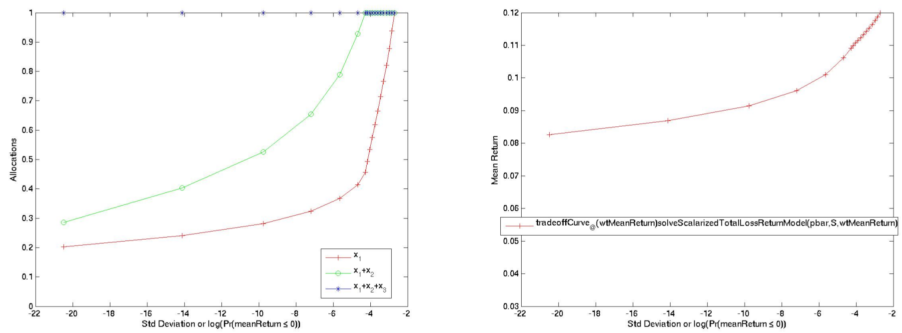

Figure 2: Area plot and Tradeoff curve. Observe that the tradeoff curve does not show the point (-Inf, 0.03).

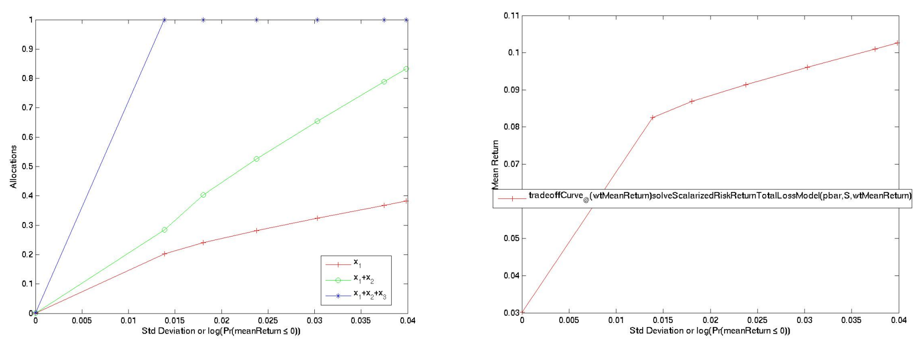

Figure 3: Area plot and Tradeoff curve

2 Enclosing Ellipses

Theoretical part submitted handwritten.

Figures are in 2.

2.1 Code

```
function final_ellipse()
    import topology.*;
    X = getData();
    weights = [0.1:0.1:1];
    numWeights = length(weights);
    ellipsesX1 = {};
```

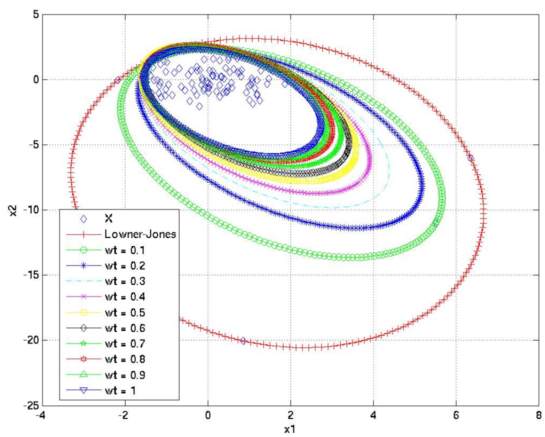

Figure 4: The ellipses

```
ellipsesX2 = {};
figureTitle = '';
legendNames = {'X'};
for i = 0:numWeights
    if i==0
        [A, b, area] = minVolumeEllipsoid(X);
        legendNames{end+1} = ['Lowner-Jones'];
        fprintf('Area: %d\n', area);
    else
        wtArea = weights(i);
        [A, b, area] =
            solveScalarizedAreaDistanceModel(X, wtArea
            );
        fprintf('Weight: %d, Area: %d\n', wtArea,
            area);
        legendNames{end+1} = ['wt = ' num2str(wtArea,
             2)];
    end
    ellipse = R2Geometry.ellipseLoci_Ab(A, b);
    ellipsesX1{end+1} = ellipse(1,:);
    ellipsesX2{end+1} = ellipse(2,:);
end
filePrefix = '/u/vvasuki/vishvas/work/optimization/hw
    /exam2/log/ellipses/';
figureName = 'ellipses';
figureHandle = plot(X(1,:), X(2, :), 'd');
hold on;
```

```
    figureHandle = IO.plotAndSave(ellipsesX1, ellipsesX2,
        'x1', 'x2', filePrefix, figureName, figureTitle,
        legendNames, figureHandle);

    display 'All done, ready for inspection';
    keyboard
end

function [A, b, area] = solveScalarizedAreaDistanceModel(
    X, wtArea)
    import topology.*;
    n = size(X, 2);
    % keyboard
    cvx_begin
    cvx_quiet(true);
    variable A(2,2);
    variable b(2, 1);
    variable t(n, 1);
    minimize(wtArea*det_inv(A) + sum(max(t - ones(n, 1),
        zeros(n, 1))));
    subject to
        A == semidefinite(2);
        for i = 1:n
            norm(A*X(:,i) + b) <= t(i);
        end
    cvx_end
    area = R2Geometry.ellipseArea_Ab(A);
end

function [A, b, area] = minVolumeEllipsoid(X)
    import topology.*;
    n = size(X, 2);
    % keyboard
    cvx_begin
    cvx_quiet(true);
    variable A(2,2);
    variable b(2, 1);
    variable t(n, 1);
    minimize(det_inv(A));
    subject to
        A == semidefinite(2);
        for i = 1:n
            norm(A*X(:,i) + b) <= 1;
        end
    cvx_end
    area = R2Geometry.ellipseArea_Ab(A);
```

```
end

function X = getData()
    % Data for the Mahalanobis tradeoff ellipsoid
        covering problem
    randn('state',0);
    X = randn(2,100);
    % add a few outliers
    X(:,50) = 10*randn(2,1);
    X(:,80) = 10*randn(2,1);
    X(:,30) = 10*randn(2,1);
end
```

3 Job Scheduling

Theoretical part submitted handwritten.

Figures are in 3.

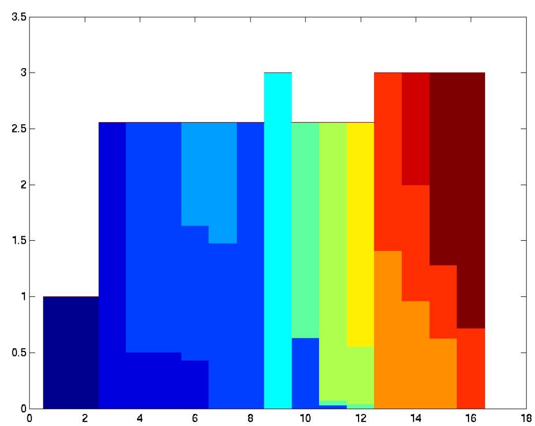

Figure 5: Speed allocation

4 Barrier method implimentation

Theoretical part submitted handwritten.

The barrier method was implemented and tested successfully, with cvx being used as the solver for the centering problems. But, my attempt to implement the newton method to solve the centering problem failed: the values of Z seem to plateau after a certain point, despite the code finding various search directions. Figures are in 4.

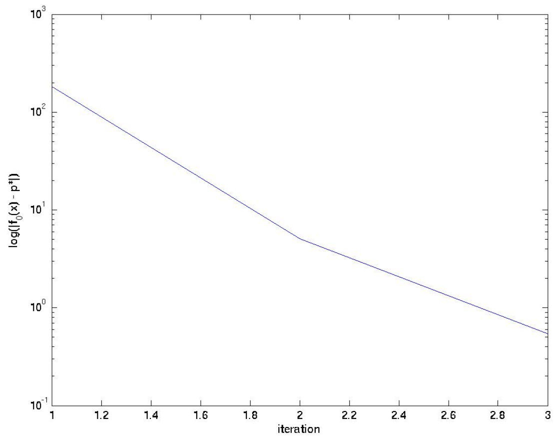

Figure 6: Error magnitude: cvx used to solve centering problem

4.1 Experiment code

```
function final_etp(centeringSolverType)
    import optimization.*;
    logPath = '/u/vvasuki/vishvas/work/optimization/hw/
       exam2/log/etp/';
    [Sigma, objOpt] = getData();
    n = size(Sigma, 1);
    ZInit = eye(n);
    xInit = zeros(n, 1);

    stoppingCriterion = @(Z, x, wt)
       stoppingCriterionBarrierMethod(Z, x, wt, Sigma);

    if(centeringSolverType == 'cvx')
        centeringProblemSolver = @(wt, Z_init)
            centeringSolver_cvx(Sigma, wt, Z_init);
    else
        centeringProblemSolver = @(wt, Z_init)
            centeringSolver(Sigma, wt, Z_init);
    end
    wtBoostFactor = 25;
    [ZValues, xValues] = DescentMethods.barrierSolver([],
        wtBoostFactor, ZInit, xInit,
       centeringProblemSolver, stoppingCriterion);
    figureHandle = DescentMethods.plotError(objOpt,
       xValues, @sum);
```

```
    figureName = func2str(centeringProblemSolver);
    IO.saveFigure(figureHandle, logPath, figureName);

    fprintf('objOpt: %d xValues goodness: ', objOpt);
    cellfun(@sum, xValues)
    fprintf('objOpt: %d ZValues goodness: ', objOpt);
    cellfun(@(Z)trace(Sigma*Z),ZValues)
%       cellfun(@(Z)objFunction(1, Sigma, Z),ZValues)

    display 'All done, ready for inspection';
    keyboard
end

function bStop = stoppingCriterionBarrierMethod(Z, x, wt,
    Sigma)
    bStop = (trace(Sigma*Z) - sum(x) <= 0.01*trace(Sigma*
        Z));
end

function SearchDirection = searchDirectionFinder(wt,
   Sigma, Z)
    n = size(Sigma, 1);
    m = [];
    m = (diag(wt*Z*Sigma*Z) - diag(Z))./(diag(Z*Z));
    SearchDirection = wt*Z*diag(m)*Z + Z - wt*Z*Sigma*Z;
    SearchDirection = 0.5*(SearchDirection +
        SearchDirection');
    SearchDirection = SearchDirection/(norm(
        SearchDirection));
%       SearchDirection = SearchDirection.*~eye(n);
    fprintf('sum(diag(SearchDirection)): %d norm: %d \n',
        sum(diag(SearchDirection)), norm(SearchDirection)
        );
    fprintf('sum(x) = %d\n', sum(m)/wt);
%       fprintf('trace(Sigma*Z): %d\n',trace(Sigma*Z));
%       fprintf('trace(Sigma*(Z + SearchDirection)): %d\n
   ',trace(Sigma*(Z + SearchDirection)));
end

function objValue = objFunction(wt, Sigma, Z)
    Z_c = chol(Z);
    lgdet = 2*sum(log(diag(Z_c)));
    objValue = wt*trace(Sigma*Z) - lgdet;
end

function [Z_iterates, x_iterates] = centeringSolver(Sigma
```

```
    , wt, Z_init)
    fprintf('wt: %d\n', wt);
    import optimization.*;
    objFn = @(Z)objFunction(wt, Sigma, Z);
    gradientFn = @(Z)wt*Sigma - pinv(Z);
    secantScaler = [];
    shrinkageFactor = [];
    cutoff = 10^-5;
    domainMembershipFn = @(Z) MatrixFunctions.
        positiveDefinitenessChecker(Z);
    searchDirectionFinderFn = @(Z)searchDirectionFinder(
        wt, Sigma, Z);
%       searchDirectionFinderFn = @(Z)-gradientFn(Z);
    stepSizeFinderFn = @(x, searchDirection)LineSearch.
        backtrackingSearchWrapper(x, searchDirection,
        objFn, gradientFn, secantScaler, shrinkageFactor,
        domainMembershipFn);
    stoppingCriterionFn = @(x, searchDirection)
        DescentMethods.stoppingCriterionNewton(x,
        searchDirection, cutoff, gradientFn);

    [Z_iterates] = DescentMethods.descentAlg(Z_init,
        searchDirectionFinderFn, stepSizeFinderFn,
        stoppingCriterionFn, [], objFn);
    x_iterates = {};
    for i = 1:length(Z_iterates)
         x_iterates{i} = getXFromOptZ(Sigma, Z_iterates{i
             }, wt);
    end
    Z_iterates = {Z_iterates{end}};
    x_iterates = {x_iterates{end}};
end

function [Z_iterates, x_iterates] = centeringSolver_cvx(
    Sigma, wt, Z_init)
    fprintf('wt: %d\n', wt);
    n = size(Sigma, 1);
    cvx_begin
    cvx_quiet(true);
    variable Z(n, n);
    minimize wt*trace(Sigma*Z) - log_det(Z);
    subject to
         for i=1:n
             Z(i,i) == 1;
         end
    cvx_end
```

```
    x = getXFromOptZ(Sigma, Z, wt);
    Z_iterates = {Z};
    x_iterates = {x};
end

function x = getXFromOptZ(Sigma, Z, wt)
    x = diag(Sigma - pinv(Z)/wt);
end

function [Sigma, objOpt] = getData()
    % Data for the educational testing problem
    randn('state',0);
    Sigma = randn(30,50);
    Sigma = Sigma * Sigma';
    objOpt = 183.7104;
end
```

4.2 Barrier method and descent algorithm Code

```
classdef DescentMethods
methods(Static=true)
function [x_iterates, l_iterates] = descentAlg(x_init,
   searchDirectionFinderFn, stepSizeFinderFn,
   stoppingCriterionFn, l_init, objFunction)
%       Input:
%           x_init
%           searchDirectionFinderFn
%           setpSizeFinderFn
%           stoppingCriterionFn
%           l_init : initial guess for dual variable.
%       Output:
%           x_iterates.
    x_opt = x_init;
    lPassed = false;
    if(nargin> 4 && ~isempty(l_init))
%           l has been passed.
        l_opt = l_init;
        lPassed = true;
        l_iterates = {l_opt};
    end
    n = numel(x_opt);
    x_iterates = {x_init};
    while(true)
        fprintf('.');
        if lPassed
            [searchDirection searchDirection_l]=
```

```
            searchDirectionFinderFn(x_opt, l_opt);
        stepSize = stepSizeFinderFn(x_opt,
            searchDirection, l_opt, searchDirection_l)
            ;
        l_opt = l_opt + stepSize*searchDirection_l;
        x_opt = x_opt + stepSize*searchDirection;
        [bStop] = stoppingCriterionFn(x_opt,
            searchDirection, l_opt);
        l_iterates{end + 1} = l_opt;
    else
        [searchDirection]= searchDirectionFinderFn(
            x_opt);
        stepSize = stepSizeFinderFn(x_opt,
            searchDirection);
        if(stepSize < 10^-12)
            fprintf('Quitting: small step size: %d ',
                stepSize);
            break;
        end
        x_opt = x_opt + stepSize*searchDirection;
        [bStop] = stoppingCriterionFn(x_opt,
            searchDirection);
    end
    x_iterates{end + 1} = x_opt;
    if(nargin > 5 && ~isempty(objFunction))
        fprintf('stepSize: %d obj: %d ', stepSize,
            objFunction(x_opt));
    end
    if(bStop)
        break;
    end
  end
  fprintf('\n');
end

function [x_opt, x_iterates] = steepestDescentHessian(
    x_init, objFn, gradientFn, hessianFn, stepSizeFinderFn
    , cutoff)
%  The newton method
  if(isempty(cutoff))
      cutoff = 10^-5;
  end
  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 searchDirection_l] =
    searchDirection_2ndOrderApproxMinEq(x, l, 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); A*x-b];
    searchDirection_with_l = M\b;
    searchDirection = searchDirection_with_l(1:n);
    searchDirection_l = searchDirection_with_l(n+1:end);
end

function [searchDirection searchDirection_l] =
    searchDirection_2ndOrderApproxMinEq_invH(x, l,
    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);
     searchDirection_l = A*invH*A'\(A*x-b -A*invH*gradient
         );
     searchDirection = -invH*(gradient + A' *
         searchDirection_l);
end


function bStop = stoppingCriterionNewton(x,
    searchDirection, cutoff, gradientFn)
     newtonDecrement = sqrt(-trace(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, l_init, objFn,
    gradientFn, searchDirectionFinderFn, stepSizeFinderFn,
     cutoff)
% The newton method for equality constrained (Ax = b)
    convex optimization problems.
%      Also works with infeasible x.
     stoppingCriterionFnInf = @(x, searchDirection, l)
         optimization.DescentMethods.
         stoppingCriterionNewtonInf(x, l, searchDirection,
         cutoff, gradientFn);
     [x_opt, x_iterates] = optimization.DescentMethods.
         descentAlg(x_init, searchDirectionFinderFn,
         stepSizeFinderFn, stoppingCriterionFnInf, l_init);
end

function [primalValues, dualValues] = barrierSolver(
    wtInit, wtBoostFactor, primalValueInit, dualValueInit,
     centeringProblemSolver, stoppingCriterion)
     if(isempty(wtBoostFactor))
          wtBoostFactor = 30;
     end
     if(isempty(wtInit))
          wtInit = 1;
     end
     primalValues = {primalValueInit};
     dualValues = {dualValueInit};
     wt = wtInit;
     bStop = false;
     while(~bStop)
          [primalValuesNew, dualValuesNew] =
              centeringProblemSolver(wt, primalValues{end});
          numIterates = length(primalValuesNew);
          primalValues = [primalValues primalValuesNew];
          dualValues = [dualValues dualValuesNew];
          wt = wt*wtBoostFactor;
          bStop = stoppingCriterion(primalValues{end},
              dualValues{end}, wt);
     end
end

function figureHandle = plotError(objOpt, x_iterates,
    objFn)
     figureHandle = figure();
     numIterations = length(x_iterates);
     iterations = 1:numIterations;
```

```
    y = [];
    for iteration = iterations
        y(iteration, 1) = objFn(x_iterates{iteration});
    end
    y = abs(objOpt*ones(numIterations, 1)-y);
    figureHandle = semilogy(iterations, y);
    ylabel('log(|f_0(x) - p*|)');
    xlabel('iteration');
%       keyboard
end

function [objMin, xBest, otherReturnValsBest] =
    discreteSequentialMinimizationScalar(domain, objFn,
    bOtherReturnVals)
% Does discrete minimization. Sequential search for the
    minimum. Assumes discrete quasiconvexity of objFn.
otherReturnValsBest = [];
otherReturnVals = [];

xBest = domain(1);
if(bOtherReturnVals)
    [objMin, otherReturnValsBest] = objFn(xBest);
else
    [objMin] = objFn(xBest);
end

objOld = objMin;
for x = domain(2:end)
    if(bOtherReturnVals)
        [obj, otherReturnVals] = objFn(x);
    else
        [obj] = objFn(x);
    end

%       fprintf(1, 'parameter: %d, obj: %d \n', x, obj);
    if(obj > objOld)
        display('Searched parameter long enough!')
        break;
    end
    if(objMin > obj)
        objMin = obj;
        xBest = x;
        otherReturnValsBest = otherReturnVals;
    end
    objOld = obj;
end
```

```
end

function [objMin, xBest] = discreteSequentialMinimization
    (domainSets, objFn)
%  Does discrete minimization. Sequential search for the
    minimum. Assumes discrete quasiconvexity of objFn.
    Also see discreteScalarSequentialMinimization.
    numVars = length(domainSets);
    if(numVars == 1)
        [objMin, xBest] = optimization.DescentMethods.
            discreteSequentialMinimizationScalar(
            domainSets, objFn, false);
        return;
    end

    cellLengths = cellfun(@length, domainSets);

    unfixedVariables = find(cellLengths > 1, 1, 'first');
    numUnfixedVariables = numel(unfixedVariables);
    if(numUnfixedVariables == 0)
        xBest= cell2mat(domainSets)';
        objMin = objFn(xBest);
        return;
    end

    unfixedVariable = unfixedVariables(1);
    fprintf('Exploring parameter %d\n', unfixedVariable);
    objFnNew = @(value)optimization.DescentMethods.
        discreteSequentialMinimization(functionals.
        Functionals.fixVariableInDomainSets(domainSets,
        unfixedVariable, value), objFn);


    [objMin, xBest, xBestOtherVars] = optimization.
        DescentMethods.
        discreteSequentialMinimizationScalar(domainSets{
        unfixedVariable}, objFnNew, true);
    xBest = xBestOtherVars;
end

function testDiscreteSequentialMinimization()
    objFn = @(x)sum(x);
    domainSets = {[6; (2:5)'], [2; -1; 5], [3;6]};
    [objMin, xBest] = optimization.DescentMethods.
        discreteSequentialMinimization(domainSets, objFn)
end
```

```
function testClass
    display 'Class definition is ok';
end


end
end
```

4.3 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*trace(
            gradient'*searchDirection))
%               fprintf('Found step size: %d %d\n',
    stepSize, trace(gradient'*searchDirection));
%               fprintf('Took: %d to %d\n',objFnSlice(0),
    objFnSlice(stepSize));
            break;
        end
        stepSize = shrinkageFactor*stepSize;
    end
end
```

```
function stepSize = backtrackingSearchEq(x,
    lagrangeMultiplier, gradientFn, searchDirection,
    lagrangeMultiplierSearchDirection, secantScaler,
    shrinkageFactor, domainMembershipFnSlice, A, b)
%    Backtracking search for equality constrained
    optimization problems with infeasible start.
    stepSize = 1;
    residualFn = @(x, lagrangeMultiplier)[gradientFn(x) +
        A'*lagrangeMultiplier; A*x - b];
    while(true)
        is_tInDomain = domainMembershipFnSlice(stepSize);
        if(is_tInDomain && residualFn(x + shrinkageFactor
            *searchDirection, lagrangeMultiplier +
            shrinkageFactor*
            lagrangeMultiplierSearchDirection) < (1 -
            secantScaler*t) * residualFn(x,
            lagrangeMultiplier))
            break;
        end
        stepSize = shrinkageFactor*stepSize;
    end
end


function stepSize = backtrackingSearchWrapper(x,
    searchDirection, objFn, gradientFn, secantScaler,
    shrinkageFactor, domainMembershipFn)
    if(isempty(secantScaler))
        secantScaler = 0.01;
    end

    if(isempty(shrinkageFactor))
        shrinkageFactor = 0.5;
    end
    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
```