Unconstrained Optimization

Landon Wright

February 2, 2018

1 Program Description

2 Testing Results

lets test this out 1/2 1/2 or $\frac{1}{2}$ because then if I have a bunch of stuff on another line it doesn't clash like it will if I use the normal math call. I guess I can see how that would be better... Here is the test text to display

3 Matlab Code

3.1 Fminun Routine

```
function [xopt, fopt, exitflag] = fminun(obj, gradobj, x0, stoptol, algoflag)

  % get function and gradient at starting point
  [n,~] = size(x0); % get number of variables
  f = obj(x0); % get the value of the function at x0
  grad = gradobj(x0);
  x = x0;
  %set starting step length
  alpha = 0.0005;
  incrementCounter = 0;
  gradOld = ones(n,1);
  sOld = zeros(n,1);

  while (any(abs(grad(:)) > stoptol))
    incrementCounter = incrementCounter + 1

    if (algoflag == 1)     % steepest descent
      s = srchsd(grad);
      % find the proper alpha level
      % function [alphaPrime] = aPrime(obj, gradobj, s, f, x)
      alphaPrime = aPrime(obj, gradobj, s, f, x);

    end

    if (algoflag == 2)
      % use conjugate gradient method
      % check that it's not the first round / we wont divide by 0
      if (gradOld' * gradOld == 0)
        sMessy = -grad;
      else
```

```
        % calculate the beta term
        betaCorrection = (grad' * grad) / (gradOld' * gradOld);
        sMessy = -grad + betaCorrection * sOld;
        sOld = sMessy;
        % normalize the s vector
        s = sMessy / norm(sMessy);
        alphaPrime = aPrime(obj, gradobj, s, f, x);
      end

    end

    if (algoflag == 3)
      % use quasi-Newton method
    end


    % take a step
    xnew = x + alphaPrime*s;
    fnew = obj(xnew);
    gradOld = grad;
    grad = gradobj(xnew);
    f = fnew;
    x = xnew;

  end
  grad
  xopt = xnew;
  fopt = fnew;
  exitflag = 0;
end

% get steepest descent search direction as a column vector
function [s] = srchsd(grad)
  mag = sqrt(grad'*grad);
  s = -grad/mag;
end
```

3.2 Alpha_star line search

```
function [alphaPrime] = aPrime(obj, gradobj, s, f, x)
  minVal = f;
  lastStepVal = f;
  alphas = [0, f];
  testStep = 3.1;
  xTest = x;
  incrementer = 2;
  iterTestStep = testStep;
  while (minVal >= lastStepVal)
    % calculate at guessed testStep
    xTest = x + iterTestStep * s;
    fTest = obj(xTest);
    if(fTest >= f & incrementer < 3)
      % disp("Step size to big, recalculating");
      minVal = f;
```

```
      lastStepVal = f;
      iterTestStep = iterTestStep / 10;
      alphas = [0, f];
      incrementer = 2;
      continue;
    end
    % add it to the stored list
    alphas(incrementer,1) = iterTestStep;
    alphas(incrementer,2) = fTest;
    % increment things for the next loop
    minVal = min(minVal, fTest);
    lastStepVal = fTest;
    incrementer = incrementer + 1;
    alphaOpt = iterTestStep;
    iterTestStep = iterTestStep * 2;
  end
  % take the half step between the last two steps
  iterTestStep = (alphas(end, 1) + alphas(end-1, 1)) / 2;
  xTest = x + iterTestStep * s;
  fTest = obj(xTest);
  % store the values of the intermediate step
  alphas(end + 1, :) = alphas(end, :);
  alphas(end - 1, :) = [iterTestStep, fTest];
  % find the index of the minimum function value
  [minVal, minIdx] = min(alphas(:,2));
  % get the three alpha and function values
  alpha2 = alphas(minIdx, 1);
  f2 = alphas(minIdx, 2);
  alpha1 = alphas(minIdx - 1, 1);
  f1 = alphas(minIdx - 1, 2);
  alpha3 = alphas(minIdx + 1, 1);
  f3 = alphas(minIdx + 1, 2);
  [alpha1, alpha2, alpha3];
  % calculate the optimum alpha value
  deltaAlpha = alpha2 - alpha1;
  alphaPrime = (f1 * (alpha2^2 - alpha3^2) + f2 * (alpha3^2 - alpha1^2) ...
        + f3 * (alpha1^2 - alpha2^2)) / (2 * (f1 * ...
        (alpha2 - alpha3) + f2 * (alpha3 - alpha1) + ...
        f3 * (alpha1 - alpha2)));
end
```

3.3 Driver

```
function [] = fminunDriv()
%----------------Example Driver program for fminun------------------
    clear;

    global nobj ngrad
    nobj = 0; % counter for objective evaluations
    ngrad = 0.; % counter for gradient evaluations
    x0 = [1.; 1.]; % starting point, set to be column vector
    x1 = [10; 10; 10]; % starting point for function 1
    xRosen = [-1.5; 1];
    algoflag = 2; % 1=steepest descent; 2=conjugate gradient; 3=BFGS quasi-Newton
```

```
    stoptol = 1.e-3; % stopping tolerance, all gradient elements must be < stoptol


    % ---------- call fminun----------------
    [xopt, fopt, exitflag] = fminun(@obj1, @gradobj1, x1, stoptol, algoflag);

    xopt
    fopt

    nobj
    ngrad
end

 % function to be minimized
 function [f] = obj(x)
    global nobj
    %example function
    %min of 9.21739 as [-0.673913, 0.304348]T
    f = 12 + 6*x(1) - 5*x(2) + 4*x(1)^2 -2*x(1)*x(2) + 6*x(2)^2;
    nobj = nobj +1;
 end

% get gradient as a column vector
 function [grad] = gradobj(x)
    global ngrad
    %gradient for function above
    grad(1,1) = 6 + 8*x(1) - 2*x(2);
    grad(2,1) = -5 - 2*x(1) + 12*x(2);
    ngrad = ngrad + 1;
 end

 % function 1 to be optimized on the homework
 function [f] = obj1(x)
   global nobj
   f = 20 + 3 * x(1) - 6 * x(2) + 8 * x(3) + 6 * x(1)^2 - 2 * x(1) * x(2) ...
       - x(1) * x(3) + x(2)^2 + 0.5 * x(3)^2;
   nobj = nobj + 1;
 end

 % gradient of function 1
 function [grad] = gradobj1(x)
   global ngrad
   grad(1,1) = 3 + 12 * x(1) - 2 * x(2) - x(3);
   grad(2,1) = -6 - 2 * x(1) + 2 * x(2);
   grad(3,1) = 8 - x(1) + x(3);
   ngrad = ngrad + 1;
 end


 % function 1 to be optimized on the homework
 function [f] = rosen(x)
   global nobj
   f = 100 * (x(2) - x(1)^2)^2 + (1-x(1))^2;
   nobj = nobj + 1;
```

```
end

% gradient of function 1
function [grad] = gradRosen(x)
  global ngrad
  grad(1,1) = 2 * (200 * x(1)^3 - 200 * x(1) * x(2) + x(1) -1);
  grad(2,1) = 200 * (x(2) - x(1)^2);
  ngrad = ngrad + 1;
end
```

ME 575
Homework #3 Unconstrained Optimization
Due Feb 7 at 2:50 p.m.

Description
Write an optimization routine in MATLAB (required) that performs unconstrained optimization. Your routine should include the ability to optimize using the methods,

- steepest descent
- conjugate gradient
- BFGS quasi-Newton

Your program should be able to work on both quadratic and non-quadratic functions of *n* variables. To determine how far to step, you may use a line search method (such as the quadratic fit given in the notes), a trust region method, or a combination of these. You will be evaluated both on how theoretically sound your program is and how well it performs.

You should use good programming style, such as selecting appropriate variable names, making the program somewhat modular (employing function routines appropriately) and documenting your code.

You will provide test results for your program on the two functions given here. In addition, you will turn in your function so we can test it on other functions or on different starting points.

Testing
1) Test your program on the following quadratic function of three variables:

$$f = 20 + 3x_1 - 6x_2 + 8x_3 + 6x_1^2 - 2x_1x_2 - x_1x_3 + x_2^2 + 0.5x_3^2$$

The conjugate gradient and quasi-Newton methods should be able to solve this problem in three iterations. Show your data for each iteration (starting point, function value, search direction, step length, number of evaluations of objective) for these two methods starting from the point $\mathbf{x}^T = [10, 10, 10]$. In addition, give data for five steps of steepest descent. At the optimum the absolute value of all elements of the gradient vector should be below 1.e-5. Indicate the total number of objective evaluations and gradient evaluations taken by each method.

2) Test your program on the following non-quadratic function (Rosenbrock's function) of two variables:

$$f = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

Starting from the point $\mathbf{x}^T = [-1.5, 1]$, show the steps of the methods on a contour plot. Show 20 steps of steepest descent. Continue with the conjugate gradient and quasi-Newton methods until the optimum is reached at $\mathbf{x}^T = [1, 1]$ and the absolute value of all elements of the gradient vector is below 1.e-3. Indicate the total number of objective evaluations and gradient evaluations taken by each method.

MATLAB Stuff

Your function will be called `fminun`. It will receive and pass back the following arguments:

```
[xopt, fopt, exitflag] = fminun(@obj, @gradobj, x0, stoptol, algoflag);

Inputs:
@obj = function handle for the objective we are minimizing (we will use
obj) The calling statement for obj looks like,

f = obj(x);

@gradobj = function handle for the function that evaluates gradients of
the objective (we will use gradobj, see example) The calling statement for
gradobj looks like,

grad = gradobj(x);

Note that grad is passed back as a column vector.

x0 = starting point (column vector)

stoptol = stopping tolerance. I will set this to 1.e-3, unless this proves
too restrictive. The absolute value of all elements of your gradient
vector should be less than this value at the optimum.

algoflag = 1 for steepest descent, =2 for conjugate gradient, =3 for
quasi-Newton.

Outputs:
xopt = optimal value of x (column vector)

fopt = optimal value of the objective

exitflag = 0 if algorithm terminated successfully; otherwise =1. Your
algorithm should exit (=1) if it has exceeded more than 500 evaluations of
the objective.
```

Attached is an example "driver" routine and example `fminun` routine to get you started. You can copy these from Learning Suite > Content > MATLAB Examples.

Grading

Grading: Your routine will be graded based on two criteria: 1) Soundness of your methodology and implementation as evidenced by your write-up and code (50%), and performance on test functions (50%). The performance score will be based 70% on accuracy (identifying the optimum) and 30% on efficiency (number of objective evaluations plus *n**number of gradient evaluations).

Turn in, in one report through Learning Suite:

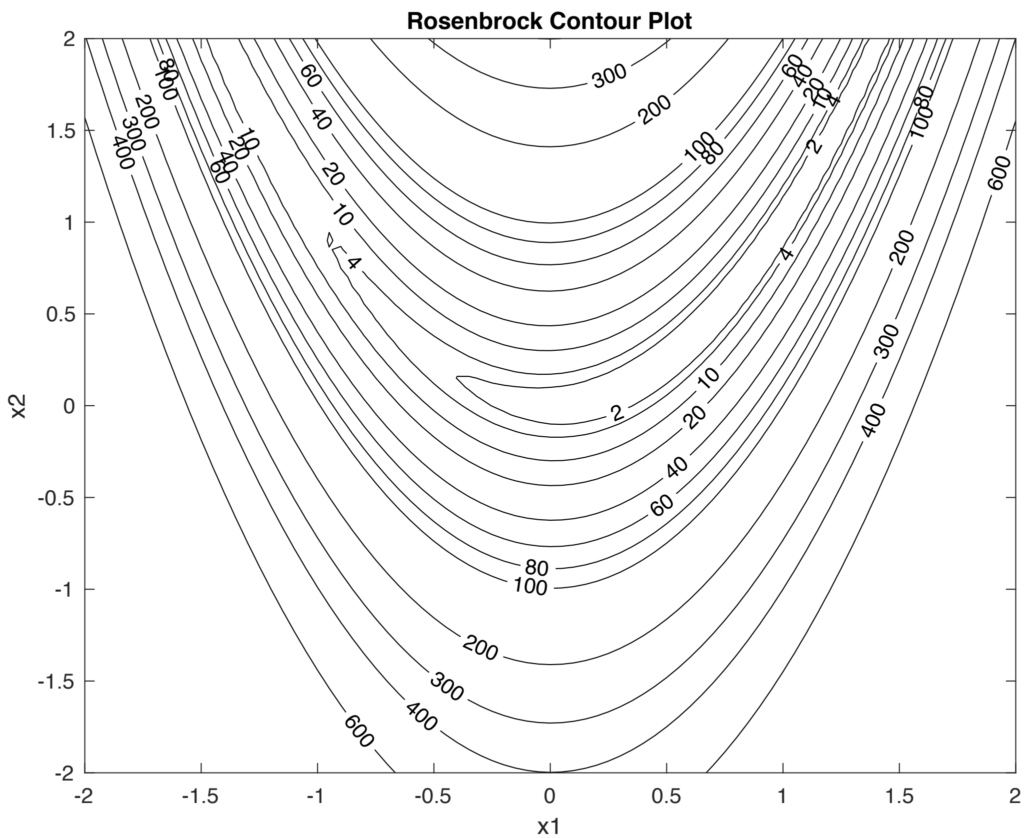
1. A brief written description of your program, including discussion of each of the three methods. This should not be longer than a page (single-spaced). You may include equations if you wish. Discuss how you implemented the methods.
2. The requested results from testing on the two functions.

3. Hardcopy of your MATLAB code.

In addition, email your MATLAB code to Jacob Greenwood (jacobgwood@gmail.com) . Additional information about this will be provided.

Suggestions:

Get started now! Start with a relatively simple routine and work forward, always having a working program. I would suggest you start off with a relatively straight-forward step length approach (such as the quadratic fit given in class) and then you can get more fancy after you have a program working for all three methods, if you want to. A working, straightforward program is much better than a non-working, fancy program.



Rosenbrock's function. The optimum is at $(\mathbf{x}^*)^T = [1, 1]$ where $f^* = 0$. Start at the point, $(\mathbf{x}^0)^T = [-1.5, 1]$

Driver Routine:

```
%-----------------Example Driver program for fminun------------------
    clear;

    global nobj ngrad
    nobj = 0; % counter for objective evaluations
    ngrad = 0.; % counter for gradient evaluations
    x0 = [1.; 1.]; % starting point, set to be column vector
    algoflag = 1; % 1=steepest descent; 2=conjugate gradient; 3=BFGS quasi-Newton
    stoptol = 1.e-3; % stopping tolerance, all gradient elements must be < stoptol


    % ---------- call fminun----------------
    [xopt, fopt, exitflag] = fminun(@obj, @gradobj, x0, stoptol, algoflag);

    xopt
    fopt

     nobj
    ngrad

     % function to be minimized
     function [f] = obj(x)
        global nobj
        %example function
        f = 12 + 6*x(1) - 5*x(2) + 4*x(1)^2 -2*x(1)*x(2) + 6*x(2)^2;
        nobj = nobj +  1;
     end

    % get gradient as a column vector
     function [grad] = gradobj(x)
        global ngrad
        %gradient for function above
        grad(1,1) = 6 + 8*x(1) - 2*x(2);
        grad(2,1) = -5 - 2*x(1) + 12*x(2);
        ngrad = ngrad + 1;
     end
```

Example fminun function:

```
function [xopt, fopt, exitflag] = fminun(obj, gradobj, x0, stoptol, algoflag)

   % get function and gradient at starting point
   [n,~] = size(x0); % get number of variables
   f = obj(x0);
   grad = gradobj(x0);
   x = x0;

   %set starting step length
   alpha = 0.5;

   if (algoflag == 1)     % steepest descent
      s = srchsd(grad)
   end

   % take a step
   xnew = x + alpha*s;
   fnew = obj(xnew);

   xopt = xnew;
   fopt = fnew;
   exitflag = 0;
end

% get steepest descent search direction as a column vector
function [s] = srchsd(grad)
   mag = sqrt(grad'*grad);
   s = -grad/mag;
end
```