Jesse Wynn
ME EN 575 Optimization
Homework 3
Unconstrained Optimization
February 5, 2018

Report

I. Description of Optimization Program

This program is a simple MATLAB script that conducts unconstrained optimization using three common methods: steepest descent, conjugate gradient, and the quasi-Newton method with a Broyden Fletcher Goldfarb Shanno (BFGS) update. This report contains results from each of these three methods for a quadratic function of three variables, and for Rosenbrock's function (two variables). For each of these methods, an exact line search routine (fitting a quadratic) was used to determine alpha* in order to take a 'minimizing step'.

Through several tests with the given functions, all three methods were successful at arriving at the optimum value for both the quadratic function of three variables and for Rosenbrock's function. In attempting to solve Rosenbrock's function using steepest descent, a solution could be found after around 1000 iterations but computation is normally halted after 500. It was noted that both the Conjugate Gradient and Steepest Descent methods were sensitive to step size (alpha) when attempting to solve Rosenbrock's function. With a value of alpha = 0.5, Conjugate Gradient and Steepest Descent both solved the quadratic function with relative ease. When tested on Rosenbrock's function, both methods failed. By adjusting alpha to alpha = 0.005, both the Steepest Descent and Conjugate Gradient methods were able to solve Rosenbrock's function at the cost of slightly more objective function evaluations. This particular value for alpha could likely be 'tuned in' to get slightly better performance, but this was not explored. The quasi-Newton BFGS method seemed to be much more robust and worked well with alpha = 0.5 for both the quadratic function and for Rosenbrock's.

As was to be expected, for both functions the quasi-Newton BFGS performed the best, the Conjugate Gradient was in the middle, and the Steepest Descent method performed the most poorly. For these tests performance was quantified by the number of minimizing steps taken (iterations), number of objective function evaluations, and number of function gradient evaluations. For more details on these results, please see section II.

To run this program, simply open fminunDriv.m, set the initial starting point, define the objective and gradient for the function you would like to minimize, choose which method you would like to use to solve (1 = Steepest Descent, 2 = Conjugate Gradient, 3 = BFGS quasi-Newton), set the stop tolerance, and finally hit 'Run'. The function fminunDriv() will call the function fminum() which has been written to solve the optimization problem. Once finished, the optimum value, and location of the optimum will be displayed in the output window along with the total number of iterations, number of objective evaluations, and number of gradient evaluations. If the function has not found an optimum after 500 iterations, it will return and indicate that it failed to reach an optimum.

II. Test Results

Problem 1: Quadratic function of three variables

Table 1: Iteration Data for Conjugate Gradient Method

Iteration	Starting Point	Obj. Value	Search Dir.	Step Length	# Obj
0	[10, 10, 10]'	520	[-93, 6, -8]'	0	1
1	[2.22, 10.50, 9.33]'	154.34	[-2.99, -10.31, -15.42]'	7.82	11
2	[-0.75, 0.26, -5.98]'	-14.40	[0.29, 3.27, -3.85]'	18.66	25
3	[-0.56, 2.44, -8.56]'	-22.39	[0.00, 0.00, 0.00]'	3.39	39

Table 2: Iteration Data for Quasi-Newton Method

Iteration	Starting Point	Obj. Value	Search Dir.	Step Length	# Obj
0	[10, 10, 10]'	520	[-0.99, 0.06, -0.09]'	0	1
1	[2.23, 10.50, 9.33]'	154.34	[-2.99, -10.32, -15.42]'	7.82	12
2	[-0.75, 0.26, -5.98]'	-14.40	[0.29, 3.27, -3.85]'	18.66	20
3	[-0.56, 2.44, -8.56]'	-22.39	[0.00, 0.00, 0.00]'	3.39	25

Table 3: Iteration Data for Steepest Descent Method

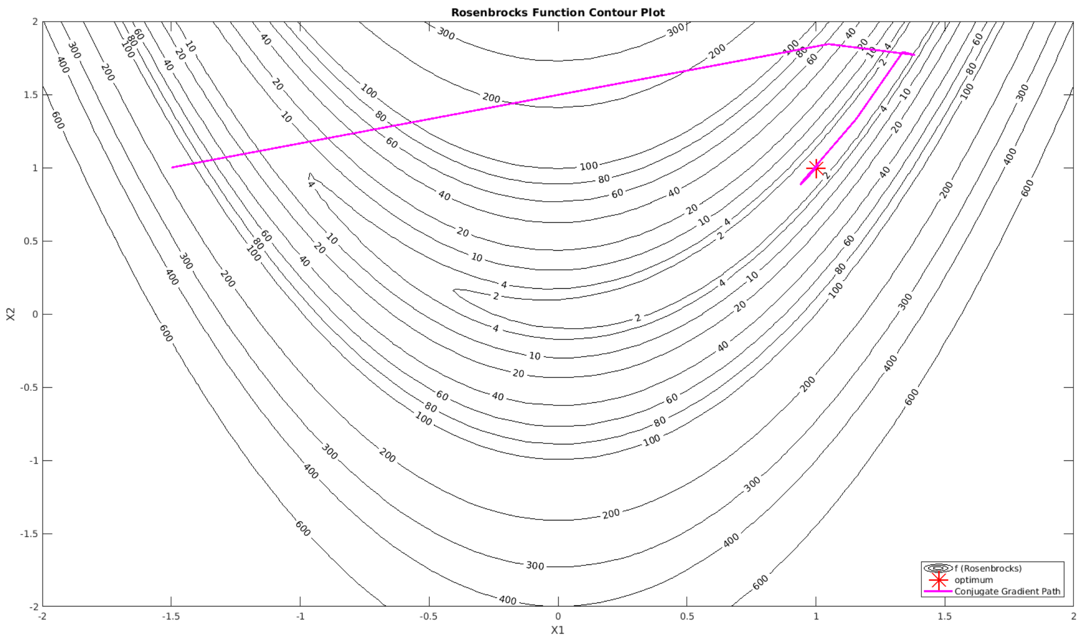
Iteration	Starting Point	Obj. Value	Search Dir.	Step Length	# Obj
0	[10, 10, 10]'	520	[-0.99, 0.06, -0.09]'	0	1
1	[2.23, 10.50, 9.33]'	154.34	[0.03, -0.57, -0.82]'	7.82	18
2	[2.65, 3.33, -0.93]'	38.88	[-0.98, 0.16, -0.15]'	12.53	36
3	[0.19, 3.72, -1.31]'	1.55	[0.12, -0.16, -0.98]'	2.51	51
4	[0.73, 3.02, -5.59]'	-12.99	[-0.98, 0.12, -0.14]'	4.38	68
5	[-0.23, 3.14, -5.74]'	-18.67	[0.10, -0.28, -0.95]'	0.98	83
33	[-0.56, 2.44, -8.56]'	-22.39	[0.11, -0.27, -0.96]	0.00	298

Total Objective and Gradient Evaluations for Each Method:

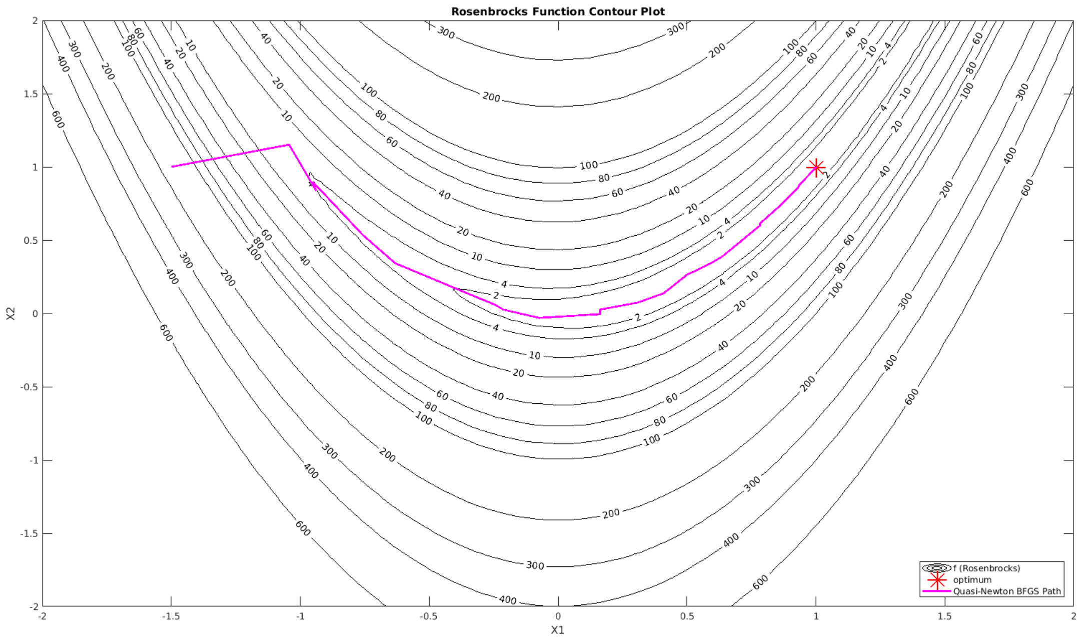
- Steepest Descent: 298 Objective Evaluations and 34 Gradient Evaluations
- Conjugate Gradient: 39 Objective Evaluations and 4 Gradient Evaluations
- BFGS quasi-Newton: 25 Objective Evaluations and 4 Gradient Evaluations

Problem 2: Rosenbrock's Function

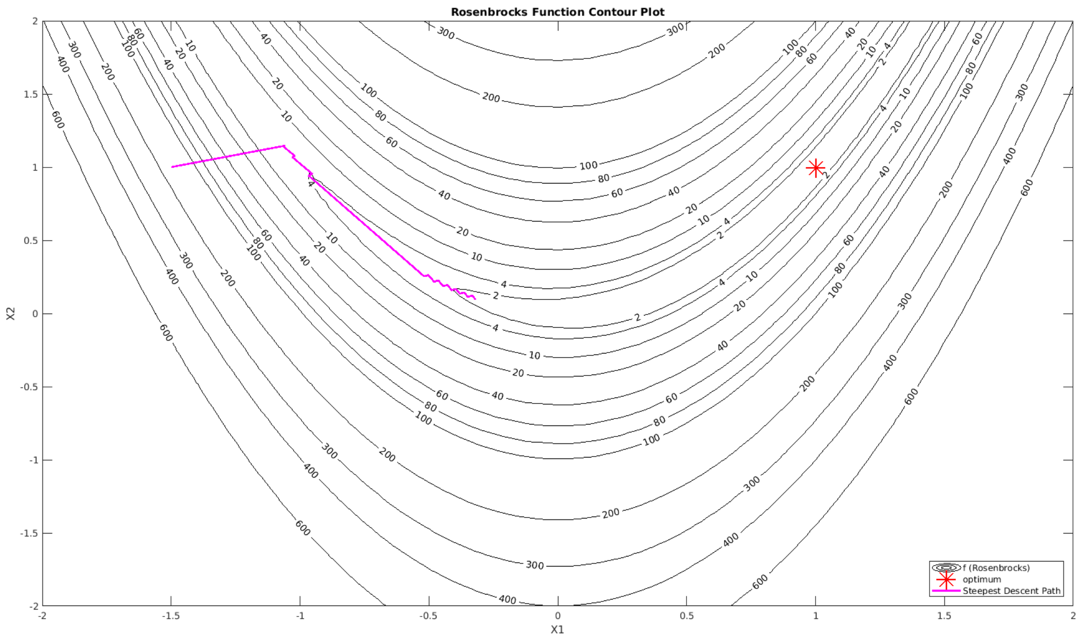
Plot 1: Conjugate Gradient Path to the optimum of Rosenbrock's function



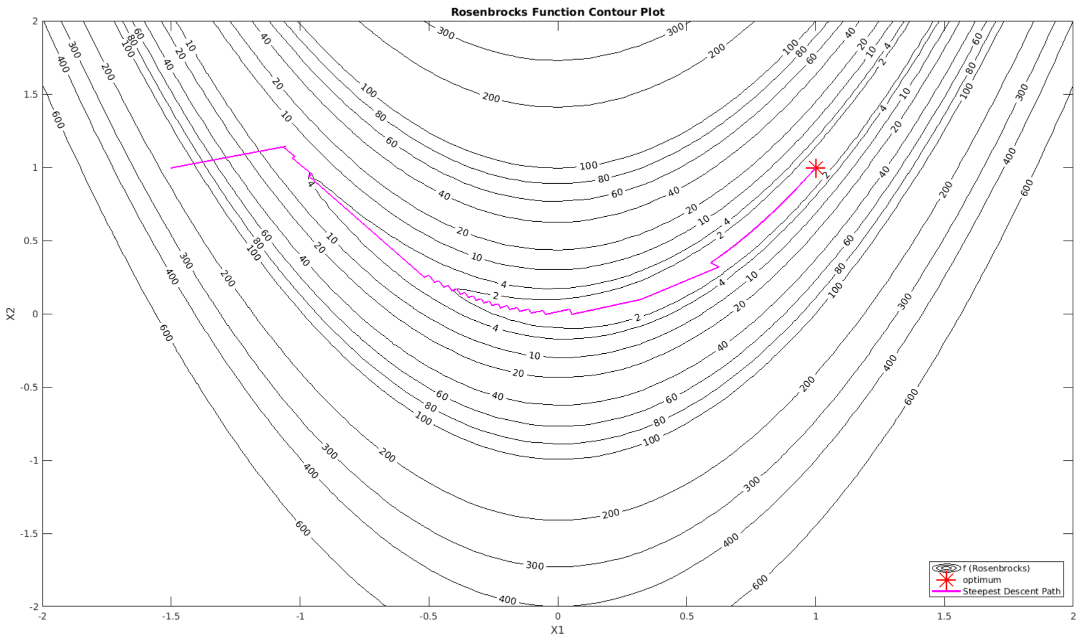
Plot 2: BFGS quasi-Newton Path to the optimum of Rosenbrock's function



Plot 3: Steepest Descent Path (first 20 steps) of Rosenbrock's function



Plot 4: Steepest Descent Path to the optimum of Rosenbrock's function



Total Objective and Gradient Evaluations for Each Method:

- Steepest Descent: 6997 Objective Evaluations and 1096 Gradient Evaluations
- Conjugate Gradient: 206 Objective Evaluations and 31 Gradient Evaluations
- BFGS quasi-Newton: 175 Objective Evaluations and 25 Gradient Evaluations

III. Appendix

MATLAB Script

```
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
% alpha0 = 0.005;

% set the initial search direction and starting step length
if (algoflag == 1)     % steepest descent
    s = srchsd(grad, 1);
    alpha0 = 0.005;

elseif (algoflag == 2)    % conjugate gradient
    s = srchsd(grad, 0);  % don't normalize search dir.
    alpha0 = 0.005;

elseif (algoflag == 3)    % BFGS Quasi-Newton
    N = eye(n);
    s = srchsd(grad, 1);
    s = N*s;
    alpha0 = 0.5;

else
    disp('Error: Valid values for algoflag are 1, 2, or 3.')
    return;
end

% initialize variables
alpha = alpha0;
num_iterations = 0;
max_iterations = 1e4;
plot_func = 0;
max_data_rows = 100;

if plot_func == 1
    % initialize array to store x_values
    x_vals = zeros(max_data_rows,2);
    x_vals(1,:) = x0';
end


%% Steepest Descent
if algoflag == 1  % steepest descent
    while num_iterations < max_iterations

        % start going along the function in the direction of s
        xnew = x + alpha*s;
```

```
% evaluate the function at xnew
fnew = obj(xnew);

% if we're making progress going in the s direction...
if fnew < f

    % double alpha and then keep going
    alpha = alpha*2;

    % set f equal to fnew and go again
    f = fnew;

    % Here we know we've gone too far.  Now we can conduct a
    % line search, find the approprate alpha, and start working
    % in a new search direction.
elseif fnew > f

    a1 = alpha/4;
    a2 = alpha/2;
    a3 = alpha;
    a4 = (a3 + a2)/2;

    % Now we need to pick the 3 points we will use
    % to bracket the minimum and perform the quadratic
    % line search.

    % evaluate the function at each
    f1 = obj(x + a1*s);
    f2 = obj(x + a2*s);
    f3 = obj(x + a3*s);
    f4 = obj(x + a4*s);

    % find the minimum point
    [~, idx] = min([f1, f2, f3, f4]);

    if idx == 2
        a1 = a1;
        a2 = a2;
        a3 = a4;

        f1 = f1;
        f2 = f2;
        f3 = f4;

    elseif idx == 4
        a1 = a2;
        a2 = a4;
        a3 = a3;

        f1 = f2;
        f2 = f4;
        f3 = f3;

    elseif idx == 1

        % in this case we need to go one more step back in the
        % past. we also need to compute these in reverse order.
```

```
                a3 = a2;
                a2 = a1;
                a1 = a1/2;


                f3 = f2;
                f2 = f1;
                f1 = obj(x + a1*s);

            else
                % this case shouldn't happen
                % disp('Unexpected case: idx containing minimum value = 3')
            end

            % find the 'optimal' alpha using a quadratic line search
            a_star = qline_search(a1, f1, a2, f2, a3, f3);

            % set a new 'x' value using a_star and the current search dir
            x = x + a_star*s;

            % sl = norm(a_star*s);

            % evaluate the gradient at this new 'x' location
            grad = gradobj(x);

            % find our new search direction to start going in
            s = srchsd(grad, 1);

            % reset alpha down to a small number
            alpha = alpha0;

            % increment the iteration count
            num_iterations = num_iterations + 1;

            % store data if we're going to plot
            if plot_func == 1 && num_iterations < max_data_rows
                x_vals(num_iterations + 1,:) = x';
            end

            % check to see if the gradient is close enough to zero
            % i.e. we're at the optimum.
            if norm(grad) < stoptol
                break
            end

        else
            % f isn't changing so we've reached an optimum?
            break
        end


    end
end

%% Conjugate Gradient
if algoflag == 2  % conjugate gradient
    while num_iterations < max_iterations
```

```
% start going along the function in the direction of s
xnew = x + alpha*s;

% evaluate the function at xnew
fnew = obj(xnew);

% if we're making progress going in the s direction...
if fnew < f

    % double alpha and then keep going
    alpha = alpha*2;

    % set f equal to fnew and go again
    f = fnew;

    % Here we know we've gone too far.  Now we can conduct a
    % line search, find the approprate alpha, and start working
    % in a new search direction.
elseif fnew > f

    a1 = alpha/4;
    a2 = alpha/2;
    a3 = alpha;
    a4 = (a3 + a2)/2;

    % Now we need to pick the 3 points we will use
    % to bracket the minimum and perform the quadratic
    % line search.

    % evaluate the function at each
    f1 = obj(x + a1*s);
    f2 = obj(x + a2*s);
    f3 = obj(x + a3*s);
    f4 = obj(x + a4*s);

    % find the minimum point
    [~, idx] = min([f1, f2, f3, f4]);

    if idx == 2
        a1 = a1;
        a2 = a2;
        a3 = a4;

        f1 = f1;
        f2 = f2;
        f3 = f4;

    elseif idx == 4
        a1 = a2;
        a2 = a4;
        a3 = a3;

        f1 = f2;
        f2 = f4;
        f3 = f3;

    elseif idx == 1
```

```
        % in this case we need to go one more step back in the
        % past. we also need to compute these in reverse order.

        a3 = a2;
        a2 = a1;
        a1 = a1/2;


        f3 = f2;
        f2 = f1;
        f1 = obj(x + a1*s);

    else
        % this case shouldn't happen
        % disp('Unexpected case: idx containing minimum value = 3')
    end

    % find the 'optimal' alpha using a quadratic line search
    a_star = qline_search(a1, f1, a2, f2, a3, f3);

    % set a new 'x' value using a_star and the current search dir
    x_plus = x + a_star*s;

    % sl = norm(a_star*s);

    % evaluate the gradient at this new 'xplus' location
    grad_plus = gradobj(x_plus);

    % compute beta
    beta = (grad_plus'*grad_plus)/(grad'*grad);

    % find our new conjugate gradient search direction to go in
    s_plus = -grad_plus + beta * s;

    % reset alpha down to a small number
    alpha = alpha0;

    % re-assign variable names for the next loop
    x = x_plus;
    s = s_plus;
    grad = grad_plus;

    % increment the iteration count
    num_iterations = num_iterations + 1;

    % store data if we're going to plot
    if plot_func == 1 && num_iterations < max_data_rows
        x_vals(num_iterations + 1,:) = x';
    end

    % check to see if the gradient is close enough to zero
    % i.e. we're at the optimum.
    if norm(grad) < stoptol
        break
    end

else
    % f isn't changing so we've reached an optimum?
```

```
            break
        end


    end
end

%% BFGS quasi-Newton
if algoflag == 3  % bfgs quasi-Newton
    while num_iterations < max_iterations

        % start going along the function in the direction of s
        xnew = x + alpha*s;

        % evaluate the function at xnew
        fnew = obj(xnew);

        % if we're making progress going in the s direction...
        if fnew < f

            % double alpha and then keep going
            alpha = alpha*2;

            % set f equal to fnew and go again
            f = fnew;

            % Here we know we've gone too far.  Now we can conduct a
            % line search, find the approprate alpha, and start working
            % in a new search direction.
        elseif fnew > f

            a1 = alpha/4;
            a2 = alpha/2;
            a3 = alpha;
            a4 = (a3 + a2)/2;

            % Now we need to pick the 3 points we will use
            % to bracket the minimum and perform the quadratic
            % line search.

            % evaluate the function at each
            f1 = obj(x + a1*s);
            f2 = obj(x + a2*s);
            f3 = obj(x + a3*s);
            f4 = obj(x + a4*s);

            % find the minimum point
            [~, idx] = min([f1, f2, f3, f4]);

            if idx == 2
                a1 = a1;
                a2 = a2;
                a3 = a4;

                f1 = f1;
                f2 = f2;
                f3 = f4;
```

```matlab
elseif idx == 4
    a1 = a2;
    a2 = a4;
    a3 = a3;

    f1 = f2;
    f2 = f4;
    f3 = f3;

elseif idx == 1

    % in this case we need to go one more step back in the
    % past. we also need to compute these in reverse order.

    a3 = a2;
    a2 = a1;
    a1 = a1/2;


    f3 = f2;
    f2 = f1;
    f1 = obj(x + a1*s);

else
    % this case shouldn't happen
    % disp('Unexpected case: idx containing minimum value = 3')
end

% find the 'optimal' alpha using a quadratic line search
a_star = qline_search(a1, f1, a2, f2, a3, f3);

% set a new 'x' value using a_star and the current search dir
x_plus = x + a_star*s;

% sl = norm(a_star*s);

% evaluate the gradient at this new 'xplus' location
grad_plus = gradobj(x_plus);

 % find delta x
delta_x = x_plus - x;

% find gamma
gamma = grad_plus - grad;

% find N_plus using BFGS Update
N_plus = N + ((1 + ((gamma'*N*gamma)/(delta_x'*gamma)))...
    *((delta_x*delta_x')/(delta_x'*gamma)))...
    - ((delta_x*gamma'*N + N*gamma*delta_x')/(delta_x'*gamma));

% compute new search direction
s_plus = -N_plus * grad_plus;

% reset alpha down to a small number
alpha = alpha0;

% re-assign variable names for the next loop
x = x_plus;
```

```
            s = s_plus;
            grad = grad_plus;
            N = N_plus;

            % increment the iteration count
            num_iterations = num_iterations + 1;

            % store data if we're going to plot
            if plot_func == 1 && num_iterations < max_data_rows
                x_vals(num_iterations + 1,:) = x';
            end

            % check to see if the gradient is close enough to zero
            % i.e. we're at the optimum.
            if norm(grad) < stoptol
                break
            end

        else
            % f isn't changing so we've reached an optimum?
            break
        end
    end
end

%%%%%%%%%%%%%%%%%
% function exit %
%%%%%%%%%%%%%%%%%

if num_iterations >= max_iterations
    max_iter_str = num2str(max_iterations);
    err_str = strcat('Failed to converge to an optimum after ', max_iter_str, ' iterations.');
    disp([newline, err_str])
    return;
end

if plot_func == 1
    plot_rosenbrock(algoflag, num_iterations, x_vals)
end



num_iterations
xopt = x;
fopt = obj(x);
exitflag = 0;

end

% get steepest descent search direction as a column vector
function [s] = srchsd(grad, normalized)
if normalized == 1
    mag = sqrt(grad'*grad);
    s = -grad/mag;
else
    s = -grad;
end
end
```

```matlab
% quadratic line search
function a_star = qline_search(a1, f1, a2, f2, a3, f3)

num = f1*(a2^2 - a3^2) + f2*(a3^2 - a1^2) + f3*(a1^2 - a2^2);
den = 2*(f1*(a2 - a3) + f2*(a3 - a1) + f3*(a1 -a2));

a_star = num/den;
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:

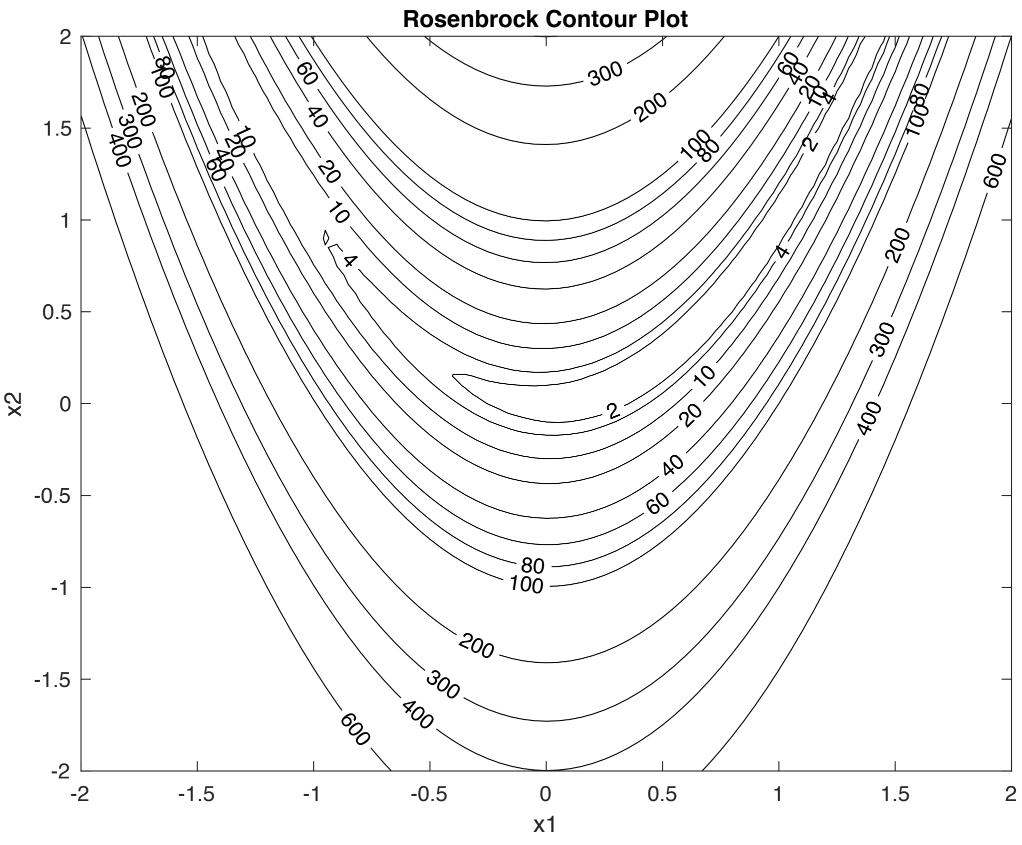
1. A brief written description of your program, including discussion of each of the three methods. This should not be longer than a page (single-spaced). You may include equations if you wish. Discuss how you implemented the methods.
2. The requested results from testing on the two functions.

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.

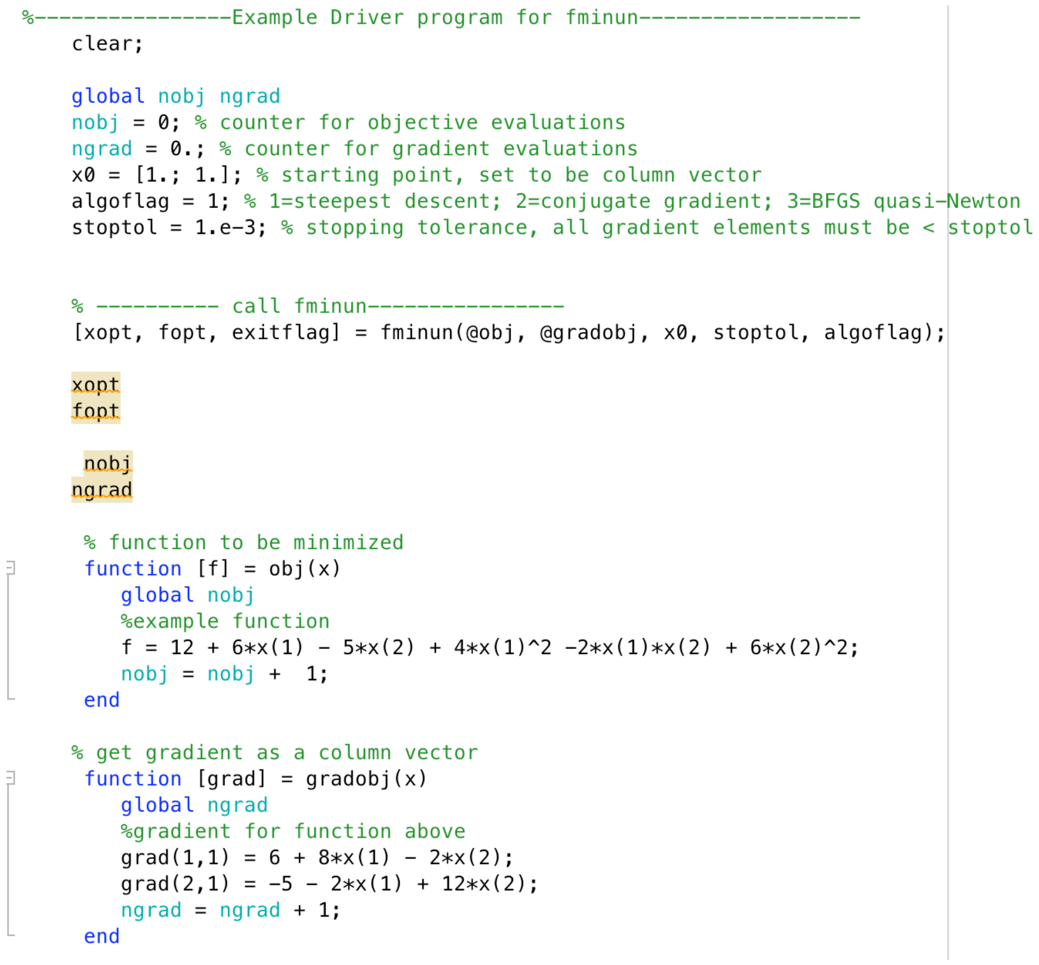
Suggestions:

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.



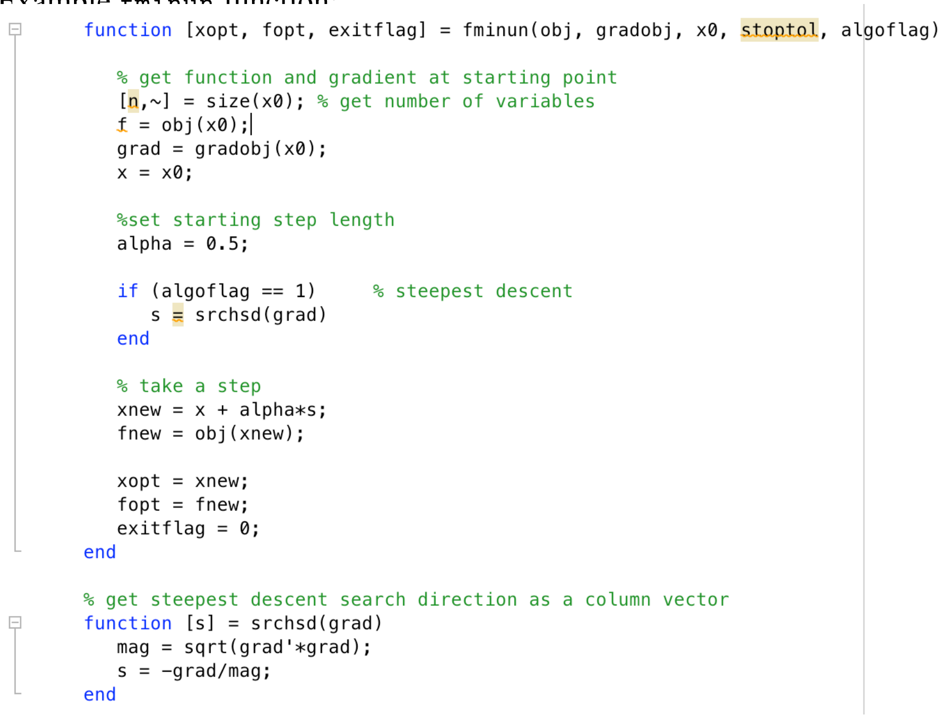
Rosenbrock's function. The optimum is at $(\mathbf{x}^*)^T = [1, 1]$ where $f^* = 0$. Start at the point, $(\mathbf{x}^0)^T = [-1.5, 1]$

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
```

ME 575
Homework #3 Addendum

Please note the following changes to the assignment:

1. The due date for the assignment is now Feb 10 at 11:50 p.m.
2. The due date for the first project is now Feb 17 at 11:50 p.m.
3. Although I would like for you to solve problem 2, I don't want this problem to become an infinite time sink. Thus the grading will be 70% on problem 1 and 30% on problem 2.

Suggestions and comments:

1. Start with the simplest way to determine the steplength that is reasonable. For example, for steepest descent, you could fit a quadratic to the last three points, whether equally spaced or not, using Equation 3.21. (Note that you still have the case where your first step results in a greater function value.)
2. For the conjugate gradient method, the appropriate equations to determine the search direction are,

$$\mathbf{s}^{k+1} = -\nabla f^{k+1} + \beta^{\mathrm{k}}\mathbf{s}^{\mathrm{k}} \tag{1}$$

   Where β^{k}, a scalar, is given by

$$\beta^{k} = \frac{\left(\nabla f^{k+1}\right)^{\mathrm{T}} \nabla f^{k+1}}{\left(\nabla f^{k}\right)^{\mathrm{T}} \nabla f^{k}} \tag{2}$$

   To compute the search direction in (1), don't use a normalized $\mathbf{s}^{\mathrm{k}}$ (where you make it a unit vector by dividing by the magnitude). This will result in an incorrect $\mathbf{s}^{\mathrm{k+1}}$. I still used a normalized **s** in my line search, but not in the calculation of the new search direction.

3. You will want to use MATLAB's debugging features, specifically the ability to set breakpoints and halt execution. To do this, set a breakpoint at the calling statement for `fminun` and then *step in* to the function. This will allow you to set and access breakpoints in your routine.

4. As a check of the derivatives for Rosenbrock's function, at the starting point, $\mathbf{x}^{\mathrm{T}}$=[–1.5, 1], the derivatives are $\nabla f^{\,\mathrm{T}}$=[–755, –250]