EGR 5110: Homework #6

Due on May 15th, 2024 at 11:59pm

Professor Nissenson

Francisco Sanudo

Contents

Background: Optimization and Gradient Ascent with Backtracking Line Search

Optimization is a fundamental problem in mathematics and computer science, where the goal is to find the best solution (maximum or minimum) of an objective function within a given domain. In numerical methods, optimization algorithms are employed to efficiently navigate through the solution space and locate optimal points.

Objective Function

Consider an objective function $f(x, y)$ representing a scalar-valued function of two variables x and y. The goal of optimization is to maximize $f(x, y)$ with respect to (x, y) within a specified region.

Gradient Ascent

Gradient ascent is an iterative optimization algorithm used to maximize a function by following the direction of steepest ascent of the gradient. The gradient $\nabla f(x, y)$ at a point (x, y) points in the direction of the greatest rate of increase of f. Therefore, moving in the direction of the gradient can lead to local maxima of the function.

Backtracking Line Search

Backtracking line search is a technique used in optimization to determine an appropriate step size for gradient-based methods. Instead of using a fixed step size, backtracking line search dynamically adjusts the step size based on certain criteria, ensuring that the function value increases sufficiently in the direction of the gradient.

Armijo Condition

The Armijo condition is a key component of backtracking line search. It ensures that the function value decreases sufficiently with respect to the gradient direction and the chosen step size. The Armijo condition is typically expressed as:

$$f(x + h \cdot g) \leq f(x) + \sigma \cdot h \cdot (g^T g)$$

where h is the step size, g is the gradient vector, σ is a small constant (e.g., $0 < \sigma < 1$), and $g^T g$ denotes the squared norm of the gradient.

Armijo Condition vs. Exact Line Search

The Armijo condition is preferred over exact line search methods for several reasons:

- **Computational Efficiency**: Exact line search methods require costly evaluations of the objective function, especially in high-dimensional spaces. In contrast, the Armijo condition only requires a few function evaluations to determine a suitable step size.
- **Robustness**: The Armijo condition provides a balance between ensuring sufficient decrease in the function value and avoiding excessive computation associated with exact line search methods.
- **Adaptability**: Backtracking line search with the Armijo condition can handle noisy or ill-conditioned objective functions more effectively compared to exact line search methods.

Gradient Ascent Algorithm in MATLAB

Homework #5 required the implementation of a the gradient ascent algorithm with inexact (backtracking) line search to solve a 2D unconstrained optimization problem. Here's a brief explanation of the key components and flow of the code, which can be found in Listing 3:

Inputs

- `f` – Objective function (anonymous function of x and y)
- `xi & yi` – Initial guesses for x and y
- `tol` – Error tolerance for convergence
- `sigma` – Armijo condition constant
- `beta` – Backtracking constant

Outputs

- `xypos` – Array containing the (x, y) coordinates at each step
- `numsteps` – Number of steps required for convergence
- `numfneval` – Number of function evaluations

Optimization Process

1. Inititialization
   - Set initial parameters (`numsteps`, `numfneval`, `maxiter`, `converged`).
   - Initialize the current position `X` with the initial guesses.
2. Gradient Ascent with Backtracking Line Search
   - Iterate until convergence (`converged = true`) or maximum iterations (`maxiter`) are reached.
   - Compute the gradient `g` of the function at the current position `X`.
   - Check the convergence condition based on the gradient norm (`err < tol`).
   - Implement backtracking line search using the Armijo condition to ensure a sufficient decrease in the function value.
   - Update the position `X` based on the computed step size (`h`) and the gradient direction.
   - Record the updated position in `xypos`.
3. Contour Plot
   - After optimization, visualize the path taken (`xypos`) overlaid on the contour plot of the function.
   - Evaluate the function `f` over a grid of `x` and `y` values to create the contour plot.
   - Plot the starting point (`xi`, `yi`), the path (`xypos`), an the optimal solution (`xypos(end,1)`, `xypos(end,2)`)

Utility Functions

- `grad`: Estimates the gradient of `f` using central finite-difference approximation.
- `euclideanNorm`: Computes the Euclidean norm of a vector.
- `applyFigureProperties`: Configures figure properties for plotting.

Recommendations

- Ensure the objective `f` is well-defined and behaves appropriately for the optimization task.
- Tune the parameters (`tol`, `sigma`, `beta`, `maxiter`) based on the characteristics of the objective function.
- Verify the gradient approximation (`grad`) accuracy with a smaller perturbation (`delta`) if needed.
- To find a local minima (i.e. gradient descent), simply negate the sign of the gradient when updating the position at each step, and also modify the Armijo condition to ensure that the function **decreases** by a minimum amount at each step.

Simulations

Consider the set of input parameters specified for the optimization algorithm using gradient ascent with backtracking line search. Each scenario is defined by different values of tolerance (tol), Armijo condition constant (σ), and backtracking constant (β). The objective function $f(x, y)$ and initial guesses for x and y are also specified.

Here is an example of how the inputs are defined for the base case:

```
xi = -10; yi = 5;                              % initial guesses
f = @(x,y) -10*(x-2).^2 - 5*(y+3).^2 + 20;      % objective function
tol = 0.01;                                    % tolerance
sigma = 0.0001;                                % armijo condition consant
beta = 0.5;                                    % backtracking constant
```

Listing 1: Function Inputs

Objective function and Optimization Setup

The objective function $f(x, y) = -10(x-2)^2 - 5(y+3)^2 + 20$ represents a quadratic function with a known maximum. The goal is to find the local maximum of $f(x, y)$ starting from the initial guesses $X_i = -10$ and $y_i = 5$.

Scenarios

Table 1 presents ten simulation scenarios with varying parameter values (tol, σ, β) and corresponding results from the optimization algorithm.

- **Steps**: Number of iterations required to reach convergence (within the specified tolerance).
- **Function Evaluations**: Total number of function evaluations performed during optimization.

Noteworthy scenarios marked with an asterisk (*) indicate that a plot was generated and the results are discussed further in the Results section.

Table 1: Ten Scenarios Demonstrating Effects of Varying tol, σ, and β

Scenario	tol	σ	β	Steps	Function evaluations
1	0.01	0.0001	0.5	11	93
2^*	0.01	0.0001	0.9	328	8834
3	0.01	0.0001	0.1	48	333
4	0.01	0.0001	0.01	87	521
5	0.01	0.01	0.5	11	93
6	0.01	0.1	0.5	10	85
7^*	0.01	0.9	0.5	64	716
8	1e-1	0.0001	0.5	9	76
9	1e-4	0.0001	0.5	15	129
10^*	1e-6	0.1	0.5	19	164

Intepretation of Results

The output from the optimization algorithm (Listing 2) for the base case (scenario 1) provides insights into the optimization process:

- **Optimal Point**: The coordinates (x, y) of the local maximum identified by the alogrithm (shown at the final step if convergence is reached).
- **Iterative Process**: Sequence of points (x, y) visited during the optimization iterations, captured in the `xypos` array.
- **Algorithm Performance**:
  - **Number of Steps**: Indicates the convergence behavior of the algorithm.
  - **Function Evaluations**: Reflects the computational cost in terms of objective function evaluations.

```
>> [xypos,numsteps,numfneval] = calcMaxStudent(f,xi,yi,tol,sigma,beta);

Results:
        x          y
  -10.0000    5.0000
    5.0000   -0.0000
    1.2500   -1.8750
    2.1875   -2.5781
    1.7188   -3.1055
    2.0703   -3.0396
    1.9824   -3.0148
    2.0044   -3.0056
    1.9989   -3.0021
    2.0016   -2.9995
    1.9996   -2.9998


Number of steps = 11
Number of function evaluations = 93
```

Listing 2: Optimization Algorithm Output

Discussion of Key Scenarios

In this section, we delve into the results and insights gained from the simulations of scenarios 2*, 7*, and 10*.

Scenario 2*

Scenario 2* utilized a small tolerance (tol $= 0.01$) and a high backtracking constant ($\beta = 0.9$). This combination resulted in a large number of steps (328) and function evaluations (8834) when compared to scenario 1. The optimization algorithm with these parameters displayed a cautious approach, taking smaller steps towards the local maximum due to the significant backtracking. The plot generated for this scenario, seen in Figure 1, shows a detailed path towards convergence, indicating careful adjustment of the step size at each iteration.

Scenario 7*

In scenario 7*, a higher Armijo condition constant ($\sigma = 0.9$) was employed along with standard tolerance (tol $= 0.01$) and backtracking constant ($\beta = 0.5$). Despite the larger σ, the algorithm required 64 steps and 716 function evaluations to converge. This behavior suggests that while a higher σ allows larger steps, it may also necessitate more cautious adjustments during the line search process. Figure 2 illustrates the optimization path, showcasing the impact of the armijo condition on step size decisions.

Scenario 10*

Scenario 10* explored a scenario with an extremely small tolerance (tol $= 1 \times 10^{-6}$) and a higher Armijo condition constant ($\sigma = 0.1$). These parameters facilitated convergence in 19 steps with 164 function evaluations. The small tolerance forced the algorithm to meticulously approach the maximum, while the moderate σ allowed more substantial steps compared to other scenarios with similar tolerances. Figure 3 illustrates the optimization journey, emphasizing the balance between precision and computational efficiency.

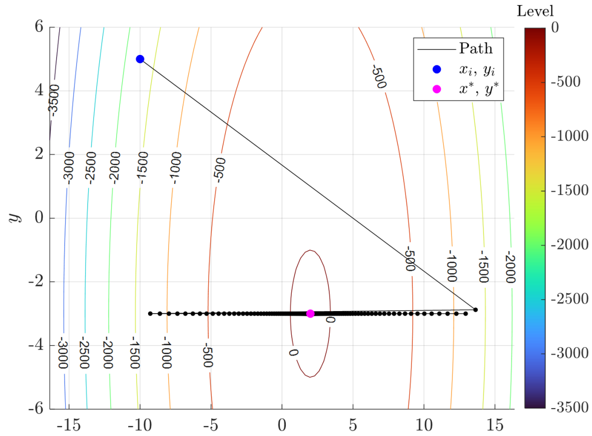

Figure 1: Optimization Path for Scenario 2

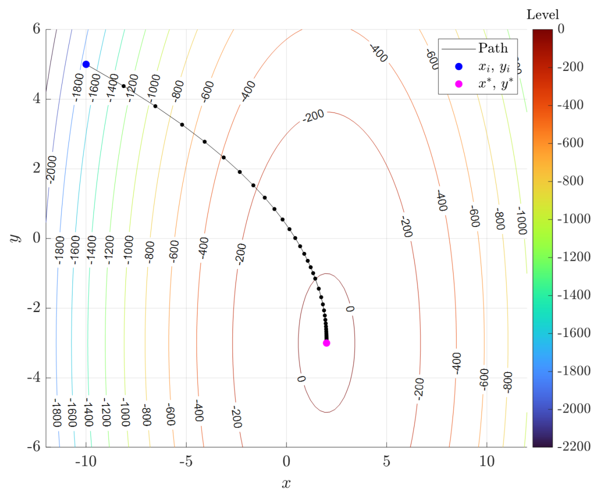

Figure 2: Optimization Path for Scenario 7

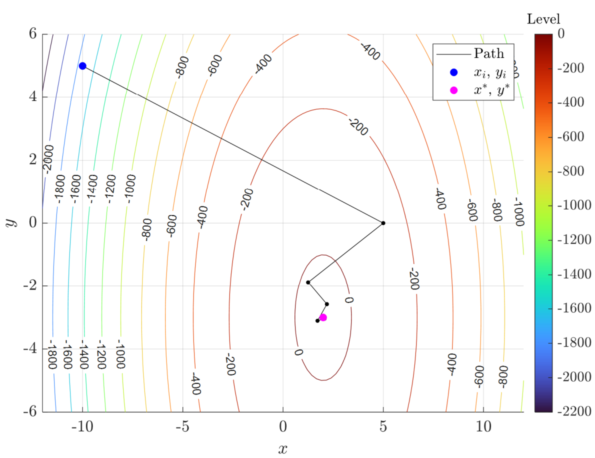

Figure 3: Optimization Path for Scenario 10

MATLAB Code

```
% Written by: Francisco Sanudo
% Date: 5/1/24
% Updated: 5/8/24
%
% PURPOSE
% calcMaxStudent solves a 2D unconstrained optimization problem by finding
% the local maximum of a 2D function f(x,y) using the gradient ascent with
% inexact (backtracking) line search method. The Armijo condition is used
% to ensure the function increases by a minimum amount at each step.
%
% INPUTS
% - f       : Objective function (anonymous func of x and y)
% - xi & yi : Initial guesses for x and y
% - tol     : Error tolerance for convergence
% - sigma   : Armijo condition constant
% - beta    : Backtracking constant
%
% OUTPUTS
% - xypos    : Array that contains the (x,y) coordinates at each step
% - numsteps : Number of steps required to achieve the termination criteria
% - numfeval : Number of function evaluations
%   --> Everytime f(x,y) is invoked, a counter variable increases by 1
%
% EXAMPLE
% xi = -10; yi = 5;                           % initial guesses
% f = @(x,y) -10*(x-2).^2 - 5*(y+3).^2 + 20;  % objective function
% tol = 1e-2;                                 % tolerance
% sigma = 0.0001;                             % armijo condition consant
% beta = 0.5;                                 % backtracking constant
%
% [xypos,numsteps,numfneval] = calcMaxStudent(f,xi,yi,tol,sigma,beta);
%
% OTHER
% .m files required               : MAIN.m (calling script)
% Files required (not .m)         : none
% Built-in MATLAB functions used : numel, zeros
% Utility functions               : grad, eucildeanNorm, applyFigureProperties, myMax
%
% REFERENCES
% Optimization (notes), P. Nissenson
%

function [xypos,numsteps,numfneval] = calcMaxStudent(f,xi,yi,tol,sigma,beta)

% Initialize Variables
numsteps  = 1;              % Set the iteration counter
numfneval = 0;              % Set the funcion evaluations counter
maxiter   = 10000;          % Maximum iterations
converged = false;          % Convergence condition

X = [xi; yi];               % Set initial condition
```

```
xypos = X';                 % Initialize position history

%% Gradient ascent with inexact (backtracking) line search method

% Iterate
while ~converged && numsteps < maxiter
    % Compute gradient
    g = grad(f, X);
    numfneval = numfneval + 4; % Increment function evaluation counter

    % Compute error using gradient norm
    err = euclideanNorm(g);

    if err < tol
        converged = true;
    else
        % Backtracking line search
        h = 1;  % Initial step size

        % Temporary step
        xtemp   = X(1) + g(1)*h;
        ytemp   = X(2) + g(2)*h;

        % Function value at current step
        fcurrent  = f(X(1),X(2));
        numfneval = numfneval + 1; % Increment function evaluation counter

        % Armijo Condition
        while f(xtemp, ytemp) - fcurrent <  sigma*(g'*g)*h
            % Reduce step size using backtracking
            h = beta * h;

            % Increment function evaluation counter
            numfneval = numfneval + 1;

            % Update temporary variables
            xtemp = X(1) + g(1)*h;
            ytemp = X(2) + g(2)*h;
        end

        % Update position
        X = X + h*g;

        % Update position history
        xypos = [xypos; X'];

        % Increment iteration counter
        numsteps = numsteps + 1;
    end
end

% Increment function evaluation counter (to account for final step when
% convergence is reached)
```

```
numfneval = numfneval + 1;

%% Contour plot that shows the path taken to the maximum

% Extract x and y coordinates from xypos
x_coords = xypos(:, 1);
y_coords = xypos(:, 2);

% Calculate maximum magnitudes of x and y coordinates
max_x_magnitude = myMax(abs(x_coords));
max_y_magnitude = myMax(abs(y_coords));

% Calculate x and y ranges (20% larger than max magnitude)
x_range = [-1, 1] * (1.2 * max_x_magnitude);
y_range = [-1, 1] * (1.2 * max_y_magnitude);

% Calculate grid spacing based on the modified ranges
dx = abs((x_range(end) - x_range(1))) / 100;
dy = abs((y_range(end) - y_range(1))) / 100;

% Define grid for contour plot
x = x_range(1):dx:x_range(2);
y = y_range(1):dy:y_range(2);

% Evaluate the function at each point on grid
%   Each row corresponds to a constant y value and each column corresponds
%   to a constant x value. This means that z(i, j) represents the function
%   value at x(j) and y(i)

z  = zeros(numel(y),numel(x));

for i = 1:numel(x)
    for j = 1:numel(y)
        z(i,j) = f(x(j), y(i));
    end
end

% Create figure and apply figure properties
f = figure;
position = [0.2, 0.2, 0.5, 0.6];
applyFigureProperties(f, position)

hold on;  % plot over same axes
ax = gca; % get current axes

% Plot contours
contour(x,y,z,'ShowText','on');

% Plot the path to the optimal solution
plot(x_coords,y_coords,'k')
plot(x_coords,y_coords,'ko','MarkerSize',3,'MarkerFaceColor','k')

% Plot starting point
```

```
plot(xi,yi,'bo','Markersize',6,'MarkerFaceColor','b')

% Plot the optimal solution
plot(x_coords(end),y_coords(end),'mo','Markersize',6,'MarkerFaceColor','m')

% Axis properties
set(ax,'TickLabelInterpreter','latex')
title('Gradient Ascent w/ Armijo Condition')
xlabel('$x$')
ylabel('$y$')
legend('','Path','','$x_i$, $y_i$','$x^*$, $y^*$')
grid on
% axis equal

% Colormap & colorbar properties
colormap(ax,'turbo');
cb = colorbar;
cb.TickLabelInterpreter = 'latex';
title(cb,'$\rm Level$','Interpreter','latex')

end

%-----------------------------------------------------------------------------

function g = grad(f,X)
% grad estimates the gradient of the function f using a centered finite-
% difference approximation.
%
% Input:
%   f: Anonymous function representing the objective function f(x, y)
%   X: Input vector containing the current point (x, y)
%
% Output:
%   g: Estimated gradient vector of f at the point X

n = numel(X);           % Dimension of input vector X
g = zeros(n, 1);        % Initialize gradient vector
delta = 0.00001;         % Perturbation parameter

% Estimate gradient
for i = 1:n
    % Create a copy of X to perturb
    X_perturbed = X;

    % Perturb the i-th element
    X_perturbed(i) = X_perturbed(i) + delta;

    % Calculate the forward difference
    f_forward = f(X_perturbed(1),X_perturbed(2));

    % Perturb in the negative direction
    X_perturbed(i) = X_perturbed(i) - 2 * delta;

```

```
    % Calculate the backward difference
    f_backward = f(X_perturbed(1),X_perturbed(2));

    % Estimate partial derivative w.r.t. i-th element using central
    % difference formula
    g(i) = (f_forward - f_backward) / (2 * delta);
end

end

%-----------------------------------------------------------------------

function euclidean_norm_value = euclideanNorm(vec)
% euclideanNorm computes the Euclidean norm (magnitude) of an n-dimensional vector.
%
% Input:
%   vec: An n-dimensional vector (column vector)
%
% Output:
%   euclidean_norm_value: The Euclidean norm of the input vector `vec`

% Square each element of vector
squared_elements = vec.^2;

% Initialize sum_of_squares
sum_of_squares = 0;

% Loop through each element
for i = 1:numel(vec)
    % Square the current element
    squared_element = vec(i)^2;

    % Add the squared element to the sum_of_squares
    sum_of_squares = sum_of_squares + squared_element;
end

% Compute the Euclidean norm
euclidean_norm_value = sqrt(sum_of_squares);

end

%-----------------------------------------------------------------------

function [maximum, idx] = myMax(A)
% myMax calculates the largest element of an array A and returns its value
% along with the index of that element in the array.
%
% Input:
%   A: Array of numeric values
%
% Output:
%   maximum: Largest element in the array A
%   idx: Index of the largest element in the array A
```

```
% Initialize variables to track the maximum value and its index
maximum = A(1);  % Assume the first element is the maximum initially
idx = 1;         % Index of the assumed maximum

% Iterate through the array to find the actual maximum value and its index
for j = 2:numel(A)
    if A(j) > maximum
        % Update the maximum value and its index if a larger value is found
        maximum = A(j);
        idx = j;
    end
end

end

%------------------------------------------------------------------------

function applyFigureProperties(figHandle, position)
% applyFigureProperties sets specific properties for a given figure handle.
%
% Inputs:
%   - 'figHandle': Handle to the figure (obtained using 'figure' or 'gcf')
%   - 'position': Position vector [left, bottom, width, height] in normalized units
%                 specifying the figure's position and size
% Output:
%   None (modifies the specified figure properties directly)
%
% Example Usage:
%   applyFigureProperties(fig, [0.2, 0.2, 0.5, 0.6]);

set(figHandle, ...
    'Units', 'normalized', ...
    'Position', position, ...
    'DefaultTextInterpreter', 'latex', ...
    'DefaultLegendInterpreter', 'latex', ...
    'DefaultAxesFontSize', 14);

end
```

Listing 3: Gradient Ascent Algorithm with Inexact (Backtracking) Line Search for 2D Optimization