Franklin W. Olin College of Engineering

MTH3199 Applied Math for Engineers – Fall 2024
Assignment 2: Newton's Method for Multiple Dimensions

Assigned: Friday, September 20, 2024.
Lab Report Due: Thursday, October 3, 2024 (11:59 PM EST).

Online Resources

- Description of the Jacobian: Wikipedia, Khan Academy, Wolfram MathWorld.
- Newton's method in multiple dimensions: Glyn A. Holton,

Overview

So far, we have spent a lot of time thinking about algorithms that compute the roots of single-input single-output functions. However, what do we do if we'd like to find the root of a vector valued function? For example, how might we construct a numerical method to compute the root of:

$$f(X) = \begin{bmatrix} f_1(X) \\ f_2(X) \\ f_3(X) \end{bmatrix} = \begin{bmatrix} x_1^2 + x_2^2 - 6 - x_3^5 \\ x_1 x_3 + x_2 - 12 \\ \sin(x_1 + x_2 + x_3) \end{bmatrix} \quad \text{where:} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1}$$

As is, none of our root finding algorithms would be able to solve this problem. However, with some alterations, we might just be able to get them to work. We will focus our attention on Newton's method:

$$x_{n+1} = x_n + \Delta x = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

To generalize this update step for multiple dimensions, we need to reexamine how it was derived in the first place. The essence of Newton's method is that each update step corresponds to finding the root of the tangent line approximation of the function at the current estimate x_n:

$$f(x_n + \Delta x) \approx f(x_n) + f'(x_n)\Delta x = 0 \quad \rightarrow \quad \Delta x = -\frac{f(x_n)}{f'(x_n)} \tag{3}$$

$$x_{n+1} = x_n + \Delta x = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4}$$

If the initial guess is sufficiently close to the root, then this linear approximation is accurate enough that x_n will get closer to the root over time. If we expand the definition of f to include functions with multiple inputs and multiple outputs, then $f'(x)$ changes from being a scalar to a matrix called the Jacobian

$$f'(x) = J(X) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}, \qquad J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{5}$$

Thus, the more general form of this linear approximation has the form:

$$f(X_n + \Delta X) \approx f(X_n) + J(X_n)\Delta X \tag{6}$$

where $J(X_n)\Delta X$ is the product of a matrix $J(X_n)$ and a vector ΔX. If we try to find the point where this linear approximation is zero, we get:

$$f(X_n) + J(X_n)\Delta X = 0 \quad \rightarrow \quad \Delta X = -J^{-1}(X_n)f(X_n) \tag{7}$$

This gives us a Newton step of:

$$X_{n+1} = X_n - J^{-1}(X_n)f(X_n) \tag{8}$$

That's all there is to it: instead of dividing by $f'(x_n)$, we are multiplying by the the inverse of the Jacobian (in the case of 1D, these two operations are identical). We will now work through the steps of implementing this multidimensional version of Newton's method.

Instructions

This entire exercise is the activity for Day 5 (Friday, September 19th). Please complete before class on Tuesday, September 24th. The deliverables will be submitted as a part of the Strandbeest lab report, which is due Thursday, October 3rd at 11:59PM.

Part 1: Setting Up Your Test Function

Write out $f(X)$ as a function in MATLAB:

$$f(X) = \begin{bmatrix} f_1(X) \\ f_2(X) \\ f_3(X) \end{bmatrix} = \begin{bmatrix} x_1^2 + x_2^2 - 6 - x_3^5 \\ x_1 x_3 + x_2 - 12 \\ \sin(x_1 + x_2 + x_3) \end{bmatrix} \quad \text{where:} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{9}$$

This function should take in a column vector X as input, and should return both $f(X)$ and $J(X)$ as output:

```
%the function name and input/output variable names
%are just what I chose, you can use whatever names you'd like
function [f_val,J] = test_function01(X)
    %your code here
end
```

I have intentionally omitted the analytical expression for the Jacobian of this function: you need to derive it on your own. As a reminder, partial derivatives are just like regular derivatives, except that you treat all other variables as constants. For instance:

$$\frac{\partial}{\partial x}\left(yx^2 + 3 - \cos(x)\right) = 2yx + \sin(x), \qquad \frac{\partial}{\partial y}\left(yx^2 + 3 - \cos(x)\right) = x^2 \tag{10}$$

Part 2: Basic Newton's Method Implementation

Use the multidimensional version of Newton's method to find a root of $f(X)$. You don't need to write out Newton's method as its own function quite yet (we'll be doing that later in the exercise): we just want to make sure we have a basic thing that works. In general, matrix inversion should be avoided at all costs. Conveniently, MATLAB has a friendly way to solve the problem of $AX = B$: matrix division (it's actually Gaussian elimination, not division):

```
%Don't do this!:
inv(J)*F

%Do this instead!:
J\F
```

Check to see if Newton's method worked properly by plugging this root back into your test function. Is the output a vector of zeros (or numbers close to zero)?

Part 3: Numerical Differentiation

Oftentimes, manually computing the analytical expression for the Jacobian can be a big chore. Sometime, you won't even have access to the inner workings of the function, rendering the task impossible. However, there is a convenient alternative: numerical differentiation! Using a small value of h, we can approximate the derivative as:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(X + he_j) - f_i(X - he_j)}{2h} \tag{11}$$

where e_j is the jth standard basis vector (a column vector where the jth element is 1 and all other elements are zero). You may remember the MATLAB example from last time:

```
%example of how to implement finite difference approximation
%for the first and second derivative of a function
%INPUTS:
%fun: the mathetmatical function we want to differentiate
%x: the input value of fun that we want to compute the derivative at
%OUTPUTS:
%dfdx: approximation of fun'(x)
function dfdx = approximate_derivative(fun,x)
    %set the step size to be tiny
    delta_x = 1e-6;

    %compute the function at different points near x
    f_left  = fun(x-delta_x);
    f_0     = fun(x);
    f_right = fun(x+delta_x);

    %approximate the first derivative
    dfdx = (f_right-f_left)/(2*delta_x);
end
```

Your task is to write an updated version of this function that approximates the Jacobian of f, where f takes in a vector as input, and generates a vector as output:

```
%Implementation of finite difference approximation
%for Jacobian of multidimensional function
%INPUTS:
%fun: the mathetmatical function we want to differentiate
%x: the input value of fun that we want to compute the derivative at
%OUTPUTS:
%J: approximation of Jacobian of fun at x
function J = approximate_jacobian(fun,x)
    %your code here
end
```

There are a few things you should consider when writing this function:

- This function should work for any arbitrary pair of input/output dimensions (the input and output can have different dimensions!). Think about how would go about finding the dimensions of the input and the output.
- It's bad practice to have a matrix that is changing size during each iteration of a for loop. You should initialize the Jacobian as a matrix (of the correct size) before filling it in.
- You can approximate an entire column of the Jacobian simultaneously!:

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_j} \\ \frac{\partial f_2}{\partial x_j} \\ \vdots \\ \frac{\partial f_n}{\partial x_j} \end{bmatrix} \approx \frac{f(X + he_j) - f(X - he_j)}{2h} \tag{12}$$

Thus, don't compute each element of the Jacobian individually! I should not see a nested pair of for loops in your implementation (only one for loop please!).

When you are done with your implementation, verify that it works by using it to evaluate the Jacobian of the test function, and comparing the two results. I have also written a function to test your implementation:

```
%function to test numerical differentiation function
function test_numerical_jacobian()
    %number of tests to perform
    num_tests = 100;

    %iterate num_tests times
    for n = 1:num_tests
        %generate a matrix, A, of random size populated with random values
        A_width = randi([1,15]);
        A_height = randi([1,15]);
        A = randn(A_height,A_width);
        %generate a vector, B, same height as A, random values inside
        B =  randn(A_height,1);

        %create a new test function. Jacobian should just be A
        test_fun = @(X) A*X + B;
        X_guess = randn(A_width,1);

        %evaluate numerical Jacobian of test_fun
        %use whatever your function name was here!
        J = approximate_jacobian(test_fun,X_guess);

        %compare with Jacobian of A
        succeed = norm(J-A)/(A_width*A_height);

        %if J is not close to A, print fail.
        if succeed>1e-9
            disp('fail!');
        end
    end
end
```

If your numerical differentiation function passes this test, it should probably be okay.

Part 4: Generalization

At this point, you should have some working code that implements the multidimensional Newton's method to find the root of the test function. As in the previous exercise, your implementation would have a lot more utility if it was not specifically hard-coded for this specific test function! Your task is to once again generalize the code that you have written to work for any input functions (and initial guess). To do so, refactor the solver portion of your code so that it is in the form of a function that takes a mathematical function and an initial guess as input arguments:

You are encouraged to reuse the code from your previous implementation of Newton's method. Once you have finished your implementation, let's do some work to make the solvers a bit more sophisticated. Please include the following:

- **Early termination conditions:** There is no need for the solver to continue to iterate if the solution is sufficiently correct, or if additional iterations will not change the solution by much. As such, at every step, your solvers should terminate if:

$$|\Delta X| = |X_{n+1} - X_n| < A_{thresh}, \quad |f(X_n)| < B_{thresh} \tag{13}$$

where A_{thresh} and B_{thresh} are a pair of very small threshold values (I recommend 10^{-14}). I would recommend using MATLAB's norm function to compute $|V|$.

- **Matrix inversion safeguards:** This is analogous to the division safeguard for 1D Newton, except that we care about matrix inversion instead of division. If the **determinant of the Jacobian** is zero, or very close to zero, this will result in a massive or undefined step size, which is really bad. In these cases, I would recommend terminating the loop instead of letting everything break. As such, before computing the next step, add a check to see if the denominator in the update step is zero (which would result in an undefined step size), or if the updated step size $|X_{n+1} - X_n|$ would be huge (also bad), in which case the program should terminate early. You can use MATLAB's det function to compute the determinant of a matrix. I would recommend that instead of testing the determinant of the Jacobian directly, you instead test for $\det(JJ^T) = 0$. This will allow your solver to also work for functions where the number of outputs is less than the number of inputs.
- A **numerical differentiation** option. Your implementation of multidimensional Newton's method should provide an option to use a Jacobian generated by the input function, as well as an option to compute a numerical approximation of the Jacobian. The choice between the two options should be a boolean input to your function. In the template I have provided, the function is set up so that this input argument itself is optional (the default setting is numerical differentiation), but how you choose to go about implementing this feature is up to you.
- **Exit flags (not required):** An exit flag is an additional output to a function that indicates whether or not the function completed successfully, or (if it was unsuccessful) how exactly it failed. MATLAB's exit flags are usually integers, whose value corresponds to success or the type of failure mode.

```
%Your Implementation of Newton's method
%INPUTS:
%fun: the mathematical function for which we want to compute the root
%   note that the output of fun may include the derivative
%   i.e. [fval,dfdx] = fun(x)
%   or not, i.e. fval = fun(x)
%x_guess the initial guess for Newton's method
%varargin: optional input corresponding to whether or not the solver
%should use numerical differentation to compute the Jacobian or
%use a Jacobian generated by fun. No input assumes numerical by default
%   true->fun is assumed to return [fval,J]
%   false->fun is assumed to only return fval
%OUTPUTS:
%x: the estimate of the root computed by the function
function x = multi_newton_solver(fun,x_guess,varargin)

    %true if supposed to use analytical jacobian, false otherwise
    use_analytical_jacobian = nargin==3 && varargin{1}(1);

    %your code here

end
```

Part 5: Testing

It turns out that, if you implemented multidimensional Newton's method correctly, it will work for functions that have fewer outputs than inputs. Check to see if your solver can correctly find a root (there are many) of the following test function:

$$f(X) = \begin{bmatrix} f_1(X) \\ f_2(X) \end{bmatrix} = \begin{bmatrix} 3x_1^2 + .5x_2^2 + 7x_3^2 - 100 \\ 9x_1 - 2x_2 + 6x_3 \end{bmatrix} \quad \text{where:} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{14}$$

I have provided a implementation of this function in MATLAB on the next page:

```
function [f_out,dfdx] = test_function02(X)
    x1 = X(1);
    x2 = X(2);
    x3 = X(3);

    y1 = 3*x1^2 + .5*x2^2 + 7*x3^2-100;
    y2 = 9*x1-2*x2+6*x3;

    f_out = [y1;y2];
    dfdx = [6*x1,1*x2,14*x3;9,-2,6];
end
```

Additionally, I have created a simple test problem with a fun animation for you to test your solver on. Consider a projectile fired with some speed v_0 at some angle θ on a planet with gravitational acceleration g. The position (p_x, p_y) of the projectile as a function of time is given by:

$$p_x = v_0 \cos(\theta)t + p_{x0}, \quad p_y = -\frac{1}{2}gt^2 + v_0 \sin(\theta)t + p_{y0} \tag{15}$$

For this problem, we will assume the values:

$$g = 2.3 \text{ m/sec}^2, \quad v_0 = 14 \text{ m/sec}, \quad p_{x0} = 2 \text{ m}, \quad p_{y0} = 4 \text{ m}, \tag{16}$$

I have implemented this motion function for you in MATLAB:

```
%projectile motion function
%theta is angle projectile is fired at (in radians)
%t is time in seconds
function V_p = projectile_traj(theta,t)
    g = 2.3; %gravity in m/s^2
    v0 = 14; %initial speed in m/s
    px0 = 2; %initial x position
    py0 = 4; %initial y position

    %compute position vector
    V_p = [v0*cos(theta)*t+px0; -.5*g*t.^2+v0*sin(theta)*t+py0];
end
```

We would like to find an initial firing angle θ so that the projectile will hit a flying target. The motion (p_x, p_y) of the target is given by:

$$p_x = 7\cos\left(3t - \frac{\pi}{7}\right) + 2\cos\left(5t + \frac{3\pi}{2}\right) + 28, \quad p_y = 9\sin\left(3t - \frac{\pi}{7}\right) + .7\sin\left(5t + \frac{3\pi}{2}\right) + 21 \tag{17}$$

I have implemented this motion function for you in MATLAB:

```
%function describing motion of a flying target
%t is time in seconds
function V_t = target_traj(t)
    a1 = 7; %x amplitude in meters
    b1 = 9; %y amplitude meters
    omega1 = 3; %frequency in rad/sec
    phi1 = -pi/7; %phase shift in radians


    a2 = 2; %x amplitude in meters
    b2 = .7; %y amplitude meters
    omega2 = 5; %frequency in rad/sec
    phi2 = 1.5*pi; %phase shift in radians

    x0 = 28;  %x offset in meters
    y0 = 21;  %y offset in meters
```

```
    %compute position vector
    V_t = [a1*cos(omega1*t+phi1)+a2*cos(omega2*t+phi2)+x0;...
           b1*sin(omega1*t+phi1)+b2*sin(omega2*t+phi2)+y0];
end
```

Your task is to use your implementation of multidimensional Newton's method to compute the initial firing angle θ that will allow the projectile to hit the target, as well as the predicted time of collision, t_c. I have written a MATLAB function that will allow you to visualize whether or not your solution was correct:

```
%visualize the motion of the projectile and the target
%theta is the initial firing angle of the projectile
%t_c is the predicted collision time
function run_simulation(theta,t_c)
    %create the plot window, set the axes size, and add labels
    hold on;
    axis equal; axis square;
    axis([0,50,0,50])
    xlabel('x (m)')
    ylabel('y (m)')
    title('Simulation of Projectile Shot at Target')

    %initialize plots for the projectile/target and their paths
    traj_line_proj = plot(0,0,'g--','linewidth',2);
    traj_line_targ = plot(0,0,'k--','linewidth',2);
    t_plot = plot(0,0,'bo','markerfacecolor','b','markersize',8);
    p_plot = plot(0,0,'ro','markerfacecolor','r','markersize',8);

    %position lists
    %used for plotting paths of projectile/target
    V_list_proj = [];
    V_list_targ = [];

    %iterate through time until a little after the collision occurs
    for t = 0:.005:t_c+1.5
        %set time so that things freeze once collision occurs
        t_input = min(t,t_c);

        %compute position of projectile and target
        V_p = projectile_traj(theta,t_input);
        V_t = target_traj(t_input);

        %update the position lists
        V_list_proj(:,end+1) = V_p;
        V_list_targ(:,end+1) = V_t;

        %index used for tail of target path
        i = max(1,size(V_list_targ,2)-300);

        %update plots
        set(t_plot,'xdata',V_t(1),'ydata',V_t(2));
        set(p_plot,'xdata',V_p(1),'ydata',V_p(2));
        set(traj_line_proj,'xdata',V_list_proj(1,:),'ydata',V_list_proj(2,:));
        set(traj_line_targ,'xdata',V_list_targ(1,i:end),'ydata',V_list_targ(2,i:end));

        %show updated plots
        drawnow;
    end
end
```

Deliverables and Submission Guidelines

This exercise is the first part of the larger Strandbeest assignment. The lab report for the Strandbeest assignment will be due on Thursday, October 3rd at 11:59 PM EST. In this lab report, make sure to document how you went about implementing multidimensional Newton's method. What were the issues that you ran into during implementation? How did you make sure that your implementation worked correctly? Describe the tests that you performed and what you learned from them.