Unconstrained Inverse Kinetics · Week III

CCO · Constraint Continous Optimization

Thursday 24th September, 2009

Johan Sejr Brinch Nielsen

Email: zerrez@diku.dk
Cpr.: 260886-2547

Dept. of Computer Science,
University of Copenhagen

Introduction

The goal of this assignment was to implement the DogLeg method and compare this to the two methods implemented so far.

Trust Region Methods

As we have seen in earlier studies of optimization methods an easy to compute decent direction can be found in the inverse direction of the gradient, that being $-g$. However, using this direction does not come without complications. First, the direction is only guaranteed to decent within some ϵ of the original point. Secondly, even if one finds an ϵ that yields the greatest decent in the objective function and follows this to what is known as the Cauchy point, the method can still be slow; zigzagging its way to a local minimum.

An alternative search direction is Newton's direction. As many other line search methods it uses a search direction described by:

$$p_k = -B^{-1}\nabla f(x_k)$$

Both of these methods are line search methods. They compute a decent direction and then decides on an appropriate step size. The method investigated in this assignment is known as a trust region method. Put shortly, it decides on a step size and then compute the a decent direction within the step size (the "region").

The idea is to use a model function m that approximates the objective function f. Knowing the step size λ, the goal is to find the best search direction in the model function (which is easy to minimize) and then follow this direction in f. Of cause one needs to verify that the model is in fact precise enough to yield a decent in f. If not, the λ is lowered. λ is a measure for the precision of the model function. The region containing points with a distance lower than λ away from x_k is called the "trust" region.

Dog-Leg Method

The Dog-Leg method is based on a combination of the Cauchy point and a Quasi-Newton method. It uses a second degree Taylor expansion as its model. It will choose its search direction based on the following three cases:

1. When the point suggested by Newton's method is within the trust region, this point is selected.

2. If this is not the case and the trust region requests a smaller step than that of the Cauchy point, the direction towards this point is followed as far as the region permits.

3. In the third case, when the region boundary is between the Cauchy point and the Newton point, the intersection between the limiting boundary and the line between these two points are chosen.

As can be seen, the possible directions is given by two joined lines (from p_k to the Cauchy point and further to Newton's point). This gives a bending line (a "dog leg"). Note that since the model function is quadratic, the Newton method will minimize it in a single iteration. So there is never a need to go further than the point found by Newton's method.

Specifically, p_k may take one of the following forms:

1.
$$p_{k+1} = p^B$$

2.
$$p_{k+1} = \frac{\Delta}{\mid p^U \mid} p^U$$

3.
$$p_{k+1} = p^U + \frac{\Delta - \mid p^U \mid}{\mid p^B \mid + 2p^{U\prime} p^B} p^B$$

Where Δ is the size of the trust region, p^B is the point suggested by Newton's method and p^U is the Cauchy point. The two latter points are defined as:

$$p^B = -B^{-1} \cdot g$$

$$p^C = \frac{g'g}{g'Bg} \cdot g$$

Where g is the gradient in the point x_k, B is an approximation of the Hessian and B^{-1} is an approximation of its inverse; both B and B^{-1} are positive definite matrices.

There is one detail in the algorithm yet to be discussed. That is what size to choose for λ and how to adjust this dynamically. Generally, one should lower λ when no decent direction is found and only raise λ when the step taken is as long as λ. In the implementation this is achieved by comparing the actual improvement in the objective function with that of the model (the expected improvement).

Verification

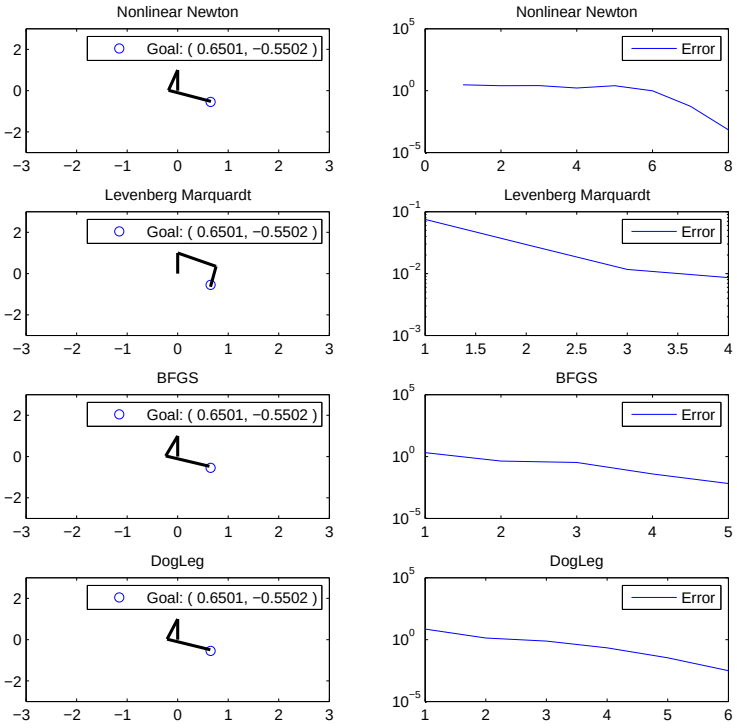
The implementation provided contains an odd bug. Apparently, the following expression becomes negative:

$$\frac{g'g}{g'Bg}$$

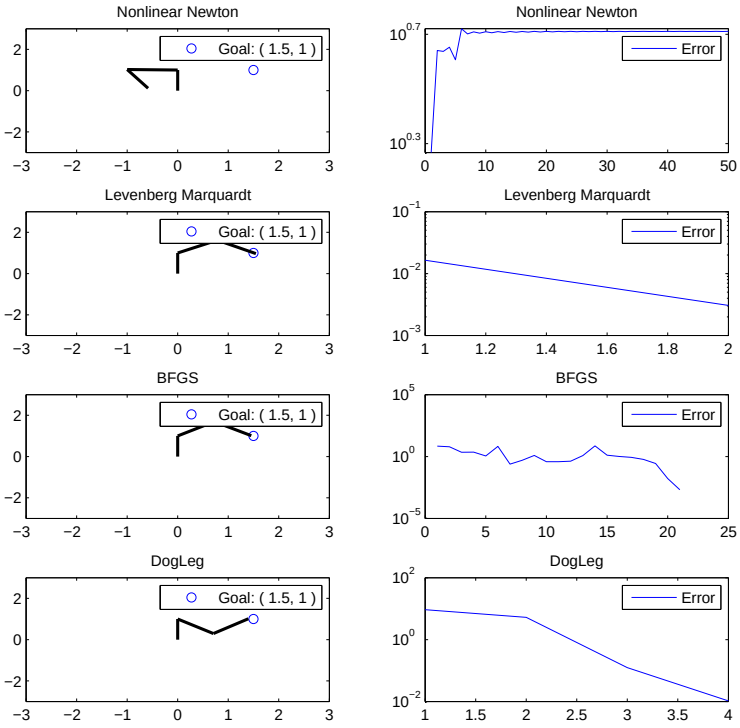
Because of this, the Cauchy point is computed to be in the opposite direction of $-g$, hence in a non-decending direction.

I have failed to figure out why this bug occurs. All I can say is, that it is periodically and that it affects the results presented in this section. The fix is quick and dirty. I have used the "abs" function to ensure non-negativity of "B".

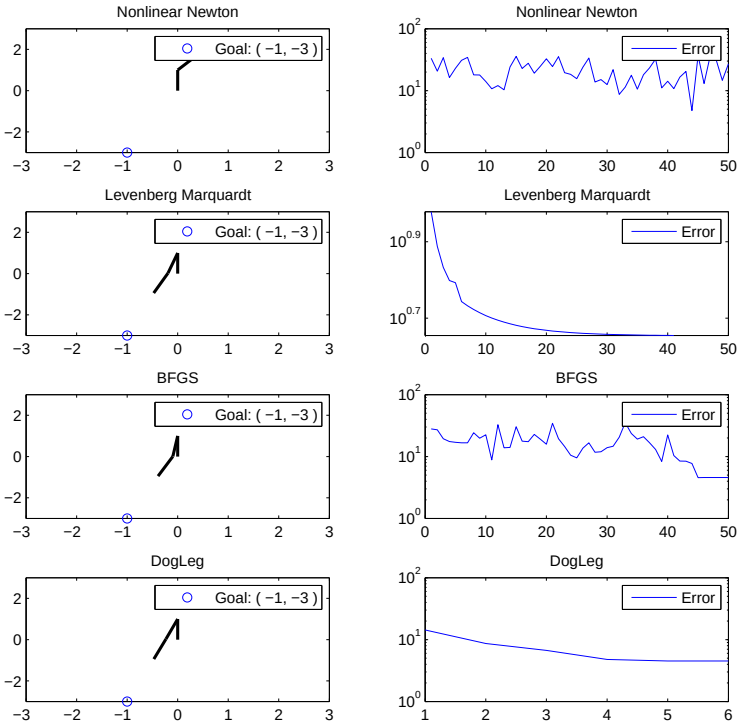
In the following I present and discuss my experimental results.



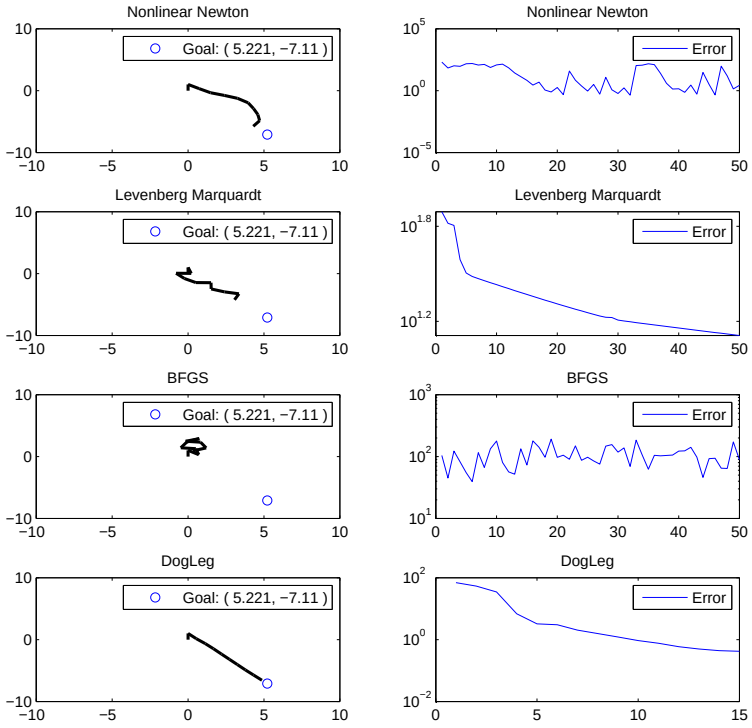
This first problem is solved effeciently by all methods. Apparently not much of a challenge, but a nice test to see that something does in fact work; despite the bug. The number of iterations are too few to say anything useful, except that the Dog-Leg method does not perform significantly worse than the others.



Here Newton's method alone fails to find the point, while Dog-Leg that is a combination of a Quasi-Newton method and the steepest descent finds a solution in just 4 iterations. Levenberg Marquardt is even faster with just two iterations.



Here is an interesting plot. While Newton's method is simple off course, Levenberg-Marquardt and BFGS spends all 50 iterations finding a solution. Dog-Leg uses 6. This suggests that Dog-Leg is more stabile than the other methods. It uses very few iterations to reach the goal even as the problems vary. It seems to be a very good all-round method.



As a final experiment I increase the number of angles to 10. Here it is clear that the newly implemented Dog-Leg wins. In just 15 iterations it has a path to the goal. The other three methods use all 50 iterations and even though Newton's method is close, it's result is clearly worse than that of Dog-Leg.

Conclusion

I have described and implemented the Dog-Leg method, though some irritating bug has crawled into the code. This has stolen a significant amount of my time (tracing and fixing) resulting in less experimentation with the method as I had wanted.

Also, the applied fix affects the chosen direction in a way that may render the results presented in the verification section invalid.

However, despite all this, the implemented method is extremely fast in the tested problems and uses just a few iterations, even when the other methods are in trouble. So, not just does it avoid calculation of the hessian and inverses - hence, has fast iterations - it will also find the solution in less iterations. It's simply too good to be true. I know where I will put my money :-)

The method may work differently, when correctly implemented.

Source Code

This section contains the MatLab code for the BFGS implementation. I also updated the source code for the other two methods slightly, however I considered the changes too small to include here.

Run

```
% Make sure we got a clean environment to work in
close all;
clear all;

% Setup a default configuration
num_angles = 10;
t      = [ zeros(1,num_angles); ones(1,num_angles) ];
angles = [ ones(num_angles,1) * pi/4 ];

% Try to find end-effector position
e = f(t, angles);

% Get x and y coordinates
x = e(1);
y = e(2);

% Verify if f worked as we expected
%if ( (x  + 1.7071) > 0.001 )
%    error('x-test failed.');
%end

%if (  (y  - 1.7071) > 0.001 )
%    error('y-test failed.');
%end


% points to test
points = [
%.6501 -.5502;
%    1.5 1;
%    -0.1011 -1.2345;
%    -1 -3               % outside reach
    5.221 -7.11
];


for i = 1:length(points)
    fig = figure(i);

    x = points(i,1);
```

```
    y = points(i,2);

    plotit('Nonlinear Newton',    @nonlinear_newton,    t, x, y, angles, 0, 4);
    plotit('Levenberg Marquardt', @levenberg_marquardt, t, x, y, angles, 1, 4);
    plotit('BFGS',                @bfgs,                t, x, y, angles, 2, 4);
    plotit('DogLeg',              @dogleg,              t, x, y, angles, 3, 4);
end
%saveas(fig, 'graph.eps', 'eps2c');
```

Plotit

```
function plotit(name, method, t, gx, gy, angles, i, N)
    [res new_angles] = method([gx; gy; 1], t, angles);

    subplot(N, 2, 2*i+1);
    plot(gx, gy, 'bo');
    legend(['Goal: ( ', num2str(gx), ', ', num2str(gy), ' )']);
    draw_chain(t, new_angles);

    NA = length(angles);
    axis([-NA NA -NA NA]);
    title(name);

    subplot(N, 2, 2*i+2);
    semilogy(res);
    legend('Error');
    title(name);
end
```

DogLeg Method

```
function [ reserr angles ] = dogleg( goal, t, angles )

    % Initial configuration
    c_delta = 0.5
    delta   = 0.5

    % Compute first endpoint
    ep = f(t, angles);

    % Gradient
    J = jacobian(t, angles);
    g = J' * (ep-goal);

    % Initial Hessian and its Inverse
    B  = eye(size(J, 2));
    Bi = eye(size(J, 2));
```

```
% Init Error Variables
reserr = [];
error  = dot(ep-goal,ep-goal);

% Iteration Counter
count = 0;
q = 1;
% Approximation local minimum
while dot(g,g) > 0.01 && count < 50
    count = count + 1

    % Newton and Cauchy points
    pB = - Bi * g;
    pU = - ((g'*g) / (g' * abs(B) * g)) * g;

    g'*B*g
    ((g'*g) / (g' * B * g))
    assert((g'*g) / (g'*B*g) >= 0)

    % Compute direction vector
    p = zeros(length(angles));
    if norm(pB) <= delta
        p = pB;
    elseif norm(pU) >= delta
        p = (delta / norm(pU)) * pU;
    else
        alpha = (delta - norm(pU)) / ...
                (norm(pB) + 2*dot(pU,pB));
        p = pU + alpha*pB;
    end

    % New Angles and Endpoint
    newAngles = angles + p;
    newEp = f(t, newAngles);

    % Compute new Error
    newError  = goal-newEp;
    newError  = dot(newError, newError);

    % Expected error
    newExpError = error + g' * p + .5 * p' * B * p;
    error = error
    newExpError = newExpError

    % Compare new error with previous
    q = (error - newError) / (error - newExpError);
    if q < 0.25
```

```
            % change region
            delta = 0.25 * norm(p);
        else
            if q > 0.75 && norm(p) == delta
                delta = min(2*delta, 2)
            end
        end
        if q < 0
            % REJECT
            continue;
        end

        % Update Approximation of Hessian
        newJ = jacobian(t, newAngles);
        y  = newJ'*(newEp-goal) - g;
        vA = (y * y') / (y' * p);
        vB = (B*p * p'*B) / (p' * B * p);
        B = B + vA - vB;
        B = abs(B);

        % Update Approximation of Inverse Hessian
        vA = ((p*p')*(p'*y + y'*Bi*y))/((p'*y)^2);
        vB = (Bi*y*p' + p*y'*Bi)/(p'*y);
        Bi = Bi + vA - vB;

        % Update Gradient
        g  = newJ' * (newEp - goal);

        % Update State
        J  = newJ;
        ep = newEp;
        angles = newAngles;
        error = newError;

        reserr = [ reserr error ];
    end
    error
    dot(g,g)
end
```