Unconstrained Inverse Kinetics · Week III

CCO · Constraint Continous Optimization

Wednesday 16th 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 BFGS Quasi-Newton method and compare this to the two methods implemented so far.

Quasi-Newton Methods

Many line search methods fall into a group, that can be identified by a direction computation that takes the following form:

$$p_{k+1} = -B^{-1}\nabla f(x)$$

Where B is a positive definite matrix and p_{k+1} is the next search direction. This group of search methods include steepest descent (with $B = I$) and Newton's method (with $B = \nabla f(x)$).

Quasi-Newton methods are a specific sort of line search methods that fall into this group. They work like Newton's method, but instead of using the Hessian, these methods approximate either this or its inverse directly. The direction p_k is hence computed as either:

$$p_{k+1} = -B^{-1}\nabla f(x)$$

or

$$p_{k+1} = -Bi\nabla f(x)$$

Where B and Bi are an approximation of the Hessian and its inverse respectively.

The new direction is now used to update the current solution by:

$$x_{k+1} = x_k + \alpha p_{k+1}$$

Where α is an appropriately chosen step length.

Quasi-Newton methods may use more iterations than those, who compute the hessian and its inverse, however each iteration costs much less in computation time due to the approximations.

Armijo Backtracking

To figure out when a step length can be considered a *good* choice, some restrictions on the resulting point has been suggested. One of these are the Armijo condition. The Armijo condition states that the function value in the new point should be no more than that of the tangent of the current point (relaxed by some constant). This condition is formalized as:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \alpha_k c_1 p_k^T \nabla f(x_k)$$

Where α_k is the current step length, c_1 is a constant relaxing the condition (by raising the gradient of the limiting line) and p_k is the current search direction.

Armijo backtracking is a simple method for finding a point, that meet the Armijo condition (hence the name). The algorithm starts with a large step length (e.g. 1). The step length is then lowered until a point that meets the condition is found.

If the function is descending in the chosen search direction (which is assumed) such a point must necessarily exist. However, it may be very close to the initial x. If so, the step taken can be very short, perhaps nearly nothing at all. Therefore it is important to ensure a minimum step size.

The algorithm can be described with the following pseudo code:

```
while not (Armijo-condition of (x+a pk)) do
  a = a * c1
end
```

BFGS Method

The Broyden–Fletcher–Goldfarb–Shanno method (BFGS) is a Quasi-Newton method. As so it approximates the inverse of the hessian. This approximation B_k is updated in each iteration like so:

$$B_{k+1} = B_k + U_k + V_k$$

Where U_k and V_k are update matrices. Let y_k be specified by:

$$y_k = \nabla f(x_k + \alpha p_k) - \nabla f(x_k)$$

The two update matrices U_k and V_k can now be described as:

$$U_k = \frac{y_k y_k^T}{y_k^T p_k}$$

$$V_k = \frac{B_k p_k (B_k p_k)^T}{p_k^T B_k p_k}$$

These matrices can be used to update the approximation of the hessian, however in my implementation I approximate the inverse of the hessian and hence use two slightly different matrices:

$$U'_k = \frac{(p_k p_k^T)(p_k^T y_k + y_k^T B_k^{-1} y_k}{(p_k^T y_k)^2}$$

$$V'_k = -\frac{B^{-1} y_k p_k^T + p_k y_k^T B_k^{-1}}{p_k^T y_k}$$

The update step of the approximation of the inverse hessian now becomes:

$$B_{k+1}^{-1} = B_k^{-1} + U'_k + V'_k$$

The implementation uses the identity matrix as the initial approximation.

Implementation

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.

Armijo Backtracking

```
function [alpha] = armijo_backtrack(f, t, x, J, pk, goal)
    ro = 0.90;
    c1 = 0.00001;
    alpha = 1;
    e = [0;0;1];
    limit = 1;
    point = 2;
    while (point > limit) && (alpha > 0.01)
        point = f(t, x + alpha*pk, e);
        point = dot(goal - point, goal - point);
        limit = f(t, x, e) + c1*alpha*transpose(J)*pk;
        limit = dot(goal - limit, goal - limit);
        alpha = alpha * ro;
    end
end
```

BFGS

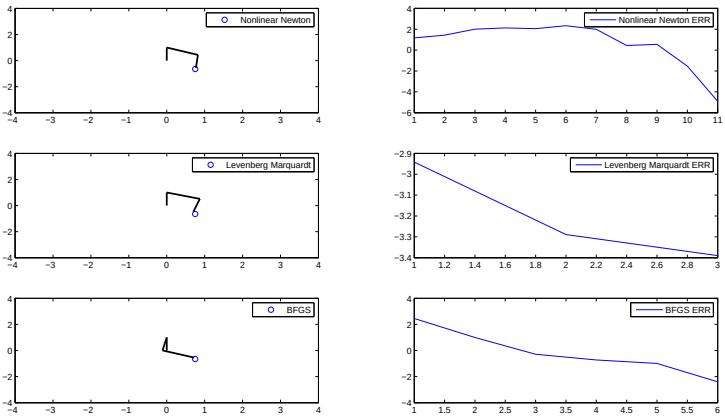
```
function [ reserr angles ] = bfgs( goal, t, angles )
    reserr = [];

    % gradient
    J  = jacobian(t, angles);
    % compute first endpoint
    ep = f(t, angles);
    % approximation of inverse hessian
    Bi = eye(size(J));

    count = 0;
    gradient = J' * (ep-goal);
    while dot(gradient,gradient) > 0.01 && count < 50
        % BFGS Search Direction
        p = - Bi * gradient;
        % Armijo Backtrack
        alpha  = armijo_backtrack(@f, t, angles, J, p, goal);

        % Update EndPoint
        newAngles = angles + alpha*p;
        newEp = f(t, newAngles);
```

```
        % Compute Error
        error  = goal-newEp;
        error  = dot(error, error);
        reserr = [reserr log(error)];

        % Update Gradient and Angles
        newJ = jacobian(t, newAngles);

        % Update Approximation of Inverse Hessian
        y  = newJ'*(newEp-goal) - gradient;
        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 State
        J = newJ;
        ep = newEp;
        angles = newAngles;
        gradient = J' * (ep-goal);
        dot(gradient, gradient)

        count = count + 1;
    end
end
```

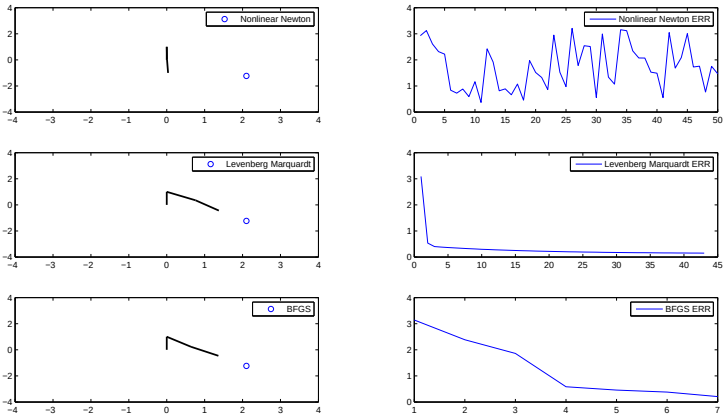
Verification

In this section I compare the performance of the three algorithms (Nonlinear Newton's Method, Levenberg Marquardt and BFGS).

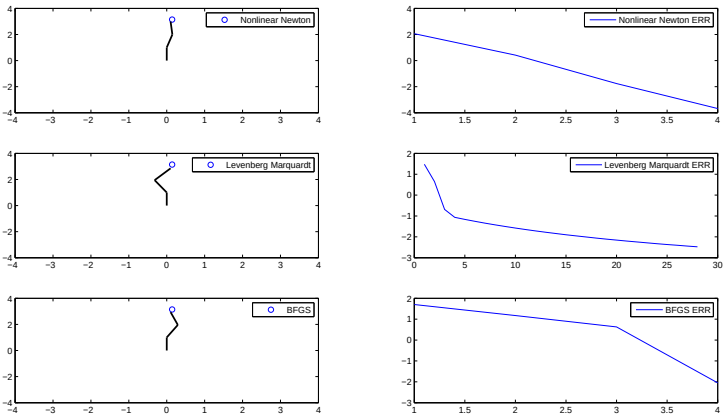
I apologize for the small graphs, but MatLab generates white margins around the graphs and ImageMagick cannot remove them without destroying the vector format. However, since the graphs are in vector graphics, it is possible to enlarge them (e.g. using a zoom feature).



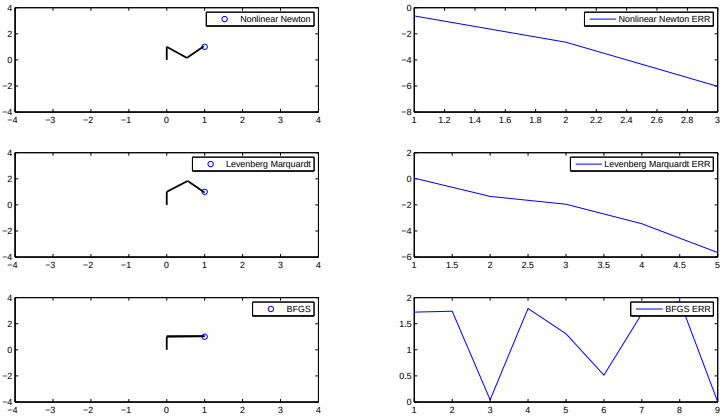
This graph illustrates what I would have expected when I first started to implement BFGS. It uses more iterations than Levenberg, but since its iterations are cheaper it could very well be faster. In this case methods finds a nice solution to the problem in few iterations.



In this case Newton's method fails to find an acceptable solution while both Levenberg and BFGS finds one. This is because the selected point is outside the reach of the "arm". I am not sure if I understand why BFGS stops after just 7 iterations; far from the goal. This should only happen if the gradient is very small.



Here Newton's method performs very well. Somehow this goal fits perfectly with the process of Newton's method. BFGS keeps up using just 4 iterations, however with a somewhat less confident convergence. Levenberg doesn't perform well here with about 27 iterations.



I find this plot very interesting. All the methods manage to find a solution in just few iterations, but look at BFGS's convergence graph. It was very close to the goal after just 3 iterations, but then something happened. What I think, is that α was chosen too large. It jumped over the solution. The Armijo-condition allowed us to jump over the solution. But luckily another one was just around the corner at 9 iterations.