Constrained Inverse Kinetics · Week V

CCO · Constraint Continous Optimization

Thursday 1st October, 2009

Johan Sejr Brinch Nielsen

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

Dept. of Computer Science,
University of Copenhagen

Introduction

In this week assignment, the optimization problem is expanded with equality and inequality contains. The goal is the implement at modified version of one of the previous optimization methods, that can handle these new constraints.

Constrained Optimization

The unconstrained optimization problem is formalized as:

$$\min f(x)$$

Where x is free to take any value in $\mathbb{R}^n$.

By introducing equality and inequality constrains, this problem is expanded to:

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0 \quad \text{for } i \in E \\ & c_i(x) \geq 0 \quad \text{for } i \in I \end{aligned}$$

Where E is the set of equality constraint indexes, while I is the set of inequality constraint indexes. $c_i(x)$ is the ith constraint applied to vector x.

It is possible to solve an optimization problem with only equality constraints, by approximating $f(x)$ with the lagrangian $L(x, \lambda)$. This yields a quadratic function which can be solved by finding x^* and λ^* that fulfills the Karush-Kuhn-Tucker conditions, which are necessary conditions for a minimum.

Active Set Methods

Active set methods is a way of solving problems with inequality constraints by transforming them into problems with equality constraints only.

Consider again problems the general constrained optimization problem:

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0 \quad \text{for } i \in E \\ & c_i(x) \geq 0 \quad \text{for } i \in I \end{aligned}$$

The idea of active set methods is to only consider inequality constraints that are being violated - the active constraints. When an inequality constraint is active, it is represented by an equality constrained forcing its value to zero. This is formalized as:

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0 \quad \text{for } i \in E \\ & c_i(x) = 0 \quad \text{for } i \in W_k \end{aligned}$$

Where W_k are the active inequality constraints in the kth iteration.

One of the simpler ways to keep track of active and nonactive inequality constraints, is to simply add a constraint to the active set whenever it is

violated. When the chosen direction is going to violate constraint j we enforce it. This will make the approximation follow the constraint border.

Whenever an inequality constraint is not violated it is not part of the active set. There is no reason to enforce a constraint that is far away. Intuitively, this approach can be thought of as letting the optimization loose until it brakes the rules. When a rule is broken we enforce it.

Gradient-Project Method

In the specific problem of this weeks programming case, we are to implement upper and lower bounds on the angles between each arm. This specific inequality constraint has the interesting property that it can be tested on each angle individually. It categorizes as a "box" constraint.

The Gradient-Project method specializes in precisely this kind of constraints. The method uses the negative gradient $-g$ as its direction (just as the gradient descent). If one or more constraints are violated, the direction vector is projected onto one or more constraint lines, to ease the constraints; the violated constraints are added to the active set and enforced. Since this changes the direction of the step, it is necessary to test whether a better objective value is found in the new point. If not, the point is rejected and the step size lowered. If so, the point is accepted and a new iteration begins.

Assuming that the first point is located in the feasible region, the method will always be able to find a better objective value inside the feasible region. There must exist some step size small enough to yield a point inside the feasible region together with a better objective value. The last fact is due to the use of the negative gradient as direction.

In the implementation, any starting point outside the feasible region will be moved inside the region if no feasible point is found in the search direction.

The process of projection is implemented using max and min functions on each angle individually. This restricts each angles to be within the given interval. The angles are normalized to the interval $[-\pi; \pi]$, before the constraints are enforced. Without normalization many of the angles will hit either the upper or lower bound, even though they are in fact inside the interval.

Verification

Verification has been performed using 20 angles. I have chosen to increase the number of angles, because it makes it more clear how the constraints effects the solution.

Instead of making several plots, I have made four movies with 75 frames each. I can recommend watching these with a player that supports control over the movie speed (frame per second). The following mplayer invocation works very well:

```
mplayer -fps 1 themovie.mpg
```

All movies shows the same optimization problem, but under different conditions. In each movie, the constraints on the angles are adjusted between

each frame by the following term:

$$-\frac{\pi}{i} \leq \theta \leq \frac{\pi}{i}$$

Where i is the current frame. Hence, the angles are squeezed from the full circle interval $[-\pi; \pi]$ to $[-\pi/75; \pi/75]$.

Due to technical problems, the movies are malformed. The scale is wrong. They look flat, but are in fact $[-12; 2]$ in the x axis and $[012]$ in the y axis. Due to another technical problem, no axis are shown in the movies.

In the movies, the goal is not always reached. I can find two possible reasons for this:

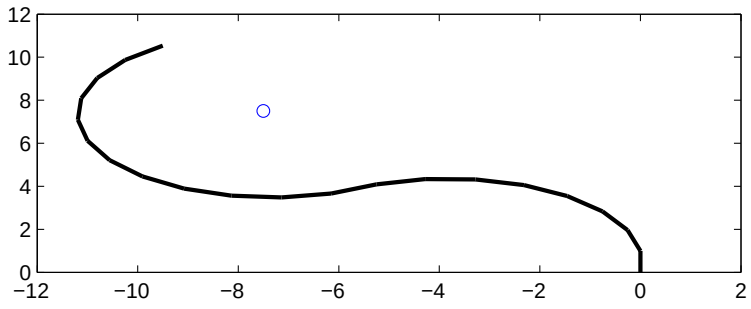
1. The number of iterations are too low. I run each approximation in at most 200 iterations. If the initial guess is too far from the goal this could lead to a result far from the goal.

2. The unconstrained optimization runs directly into a constraint line without changing direction (step size equals zero).

Movie 1 - zero free angles

In this movie all angles are affected by the constraints equally and hence, none of them can be chosen freely.

The interesting part in this movie is to see how the arm bends while the constraints tightens in. In the end the angles are too small for the arm to reach the goal. Instead it shoots over, as is seen in the following image:

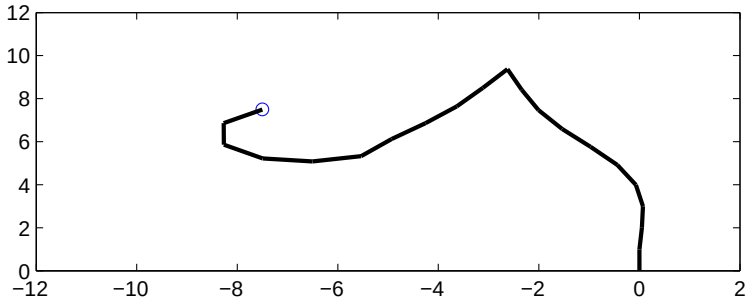


Movie 2 - one free angle

In this movie angle number 10 is unaffected of the constraints and can therefore be chosen freely.

Here, the interesting part is to see how the optimization exploits this in its attempt to reach the goal.

An example of such a "crack" is clearly visible in the image below:

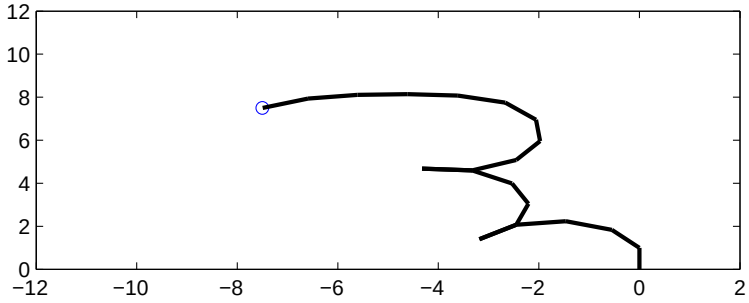


Movie 3 - two free angles

In this movie angles number 5 and 10 are unaffected of the constraints and can hence be chosen freely.

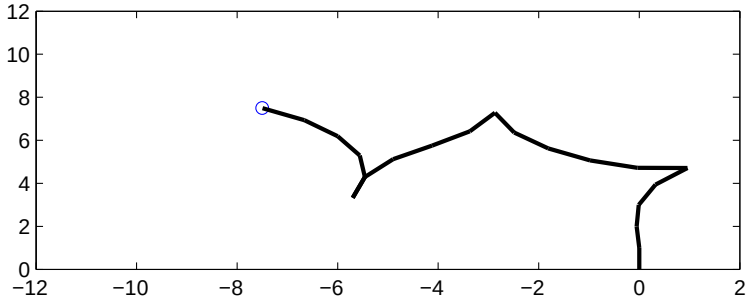
Here, the interesting part is to see how these two free angles can be used in cooperation to find a path to the goal.

An example of both angles being used in a "free" matter is shown below:



Movie 4 - three free angles

In this movie angles number 5, 10 and 15 are unconstrained. This allows for three possible "cracks". An example of such three "crakcs" are shown here:



Conclusion

I have implemented and described the Gradient-Project method and shown how different inequality constraints affects the solutions found. Both with wide angles moving to small angles and with several free angles.

Unfortunately the Gradient-Project method only works on "box" constraints, where each constraint can be tested against each variable individually.

Source Code

This section contains the MatLab code for the Gradient-Projection implementation. I have also added a function for generating movies.

Gproject

```
function [ reserr angles ] = gproject( goal, t, angles, bounds)
    if nargin < 4
        bound  = ones(length(angles),1) * 2*pi;
        bounds = [ -bound bound ];
    end

    reserr = [];

    % gradient
    J  = jacobian(t, angles);
    % compute first endpoint
    ep = f(t, angles);

    count = 0;
    gradient = J' * (ep-goal);
    while dot(gradient,gradient) > 0.01 && count < 200
        % Gradient Descent
        p = -gradient;

        % Armijo Backtrack wrt. lower and uppers bounds
        newAngles = gproject_backtrack(@f, t, angles, J, p, goal, bounds);

        % Update EndPoint
        newEp = f(t, newAngles);

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

        % Update State
        J = jacobian(t, newAngles);;
        ep = newEp;
```

```
        angles = newAngles;
        gradient = J' * (ep-goal);

        count = count + 1;
    end
end
```

Gproject Backtrack

```
function [angles] = gproject_backtrack(f, t, x, J, pk, goal, bounds)

    ro = 0.90;
    alpha = 2;
    e = [0;0;1];

    limit = f(t, x, e);
    limit = dot(goal - limit, goal - limit);
    point = limit + 1;

    % normalize angles
    for i = 1:length(x)
        while x(i) > pi
            x(i) = x(i) - 2*pi;
        end
        while x(i) < -pi
            x(i) = x(i) + 2*pi;
        end
    end

    while (point > limit)
        point = f(t, max(min(x + alpha*pk, ...
                             bounds(:,2)), bounds(:, 1)), e);
        point = dot(goal - point, goal - point);

        alpha = alpha * ro;
    end

    angles = max(min(x + alpha*pk, bounds(:,2)), bounds(:,1));
end
```

Gproject Movie

```
function [frames] = gproject_movie(num_angles, iters)

    pointx = -0.75*num_angles/2;
    pointy = 0.75*num_angles/2;

    M = moviein(iters);
```

```
    f = figure(1);
    angles = ones(num_angles,1) * pi/4;

    for i = 1:iters
        i

        dom = pi / i;

        bound  = ones(num_angles, 1) * dom;
        bounds = [ -bound bound ];

        bounds( 5,:) = [ -pi pi ];
        bounds(10,:) = [ -pi pi ];
        bounds(15,:) = [ -pi pi ];

        t = [ zeros(1,num_angles); ones(1,num_angles) ];

        % enforce the new constraints
        angles = max(min(angles, bound), -bound);
        [res angles] = gproject([pointx; pointy; 1], t, angles, ...
                                bounds);

        plot(pointx, pointy, 'bo');
        draw_chain(t, angles);
        axis([-12 2 0 12]);

        M(:,i) = getframe;
    end

    mpgwrite(M, jet, 'movie.mpg');
end
```