Semester project TMA4215

10068 and 728387

30.09.2011

1 Task

We consider minimization problems of the type

$$\min_{\mathbf{x}\in\mathbb{R}^n} g(\mathbf{x}),\ g(\mathbf{x}) := -\mathbf{b}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \frac{1}{12}\mathbf{x}^T C(\mathbf{x})\mathbf{x},$$

here $\mathbf{b}\in\mathbb{R}^n$ and, H is a $n\times n$ symmetric and positive definite matrix and $C(\mathbf{x})$ is a diagonal matrix with diagonal entries $c_i x_i^2, i = 1, ..., n$. Here $c_i > 0$ are the components of a vector $\mathbf{c}\in\mathbb{R}^n$ and x_i are the components of $\mathbf{x}$.

2 Mathematical calculations

At first some calculations are done.

2.1 Positive definition of H

Since we use only one H, the proof is not generally, but only for one special matrix. Let $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \in \mathbb{R}^2$, and $H = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$, with $a, c > 0$.

$$\mathbf{u}^T H\mathbf{u} = au_1^2 + cu_2^2 + 2bu_1u_2$$

If we choose $b = \sqrt{a}\sqrt{c}$, we get

$$= (\sqrt{a}u_1 + \sqrt{c}u_1)^2 > 0,$$

no matter what $\mathbf{u}$ is. H is positive definite with this choice of b.

2.2 Gradient

The gradient can easily be calculated with sums.

$$\nabla g = \nabla\left(-\sum_{i=1}^n b_i x_i + \frac{1}{2}\sum_{i=1}^n\sum_{j=1}^n H_{ij}x_i x_j + \frac{1}{12}\sum_{i=1}^n c_i x_i^4\right)$$

The two sums in the middle are divided into the diagonal element and the not diagonal elements. All not diagonal elements are there twice, because H is symmetric and therefore $H_{ij} = H_{ji}$.

$$\begin{aligned} &= \nabla\left(-\sum_{i=1}^n b_i x_i + \frac{1}{2}\sum_{i=1}^n H_{ii}x_i^2 + \sum_{i=1}^n\sum_{j=1}^{i-1} H_{ij}x_i x_j + \frac{1}{12}\sum_{i=1}^n c_i x_i^4\right) \\ &= -\mathbf{b} + H\mathbf{x} + \frac{1}{3}C\mathbf{x} \end{aligned}$$

2.3 Hessian

The Hessian of g is easily calculable:

$$\nabla^2 g = H + C$$

2.4 Existence of minimum

Let $u \in R^n$ be an arbitrary vector, except $\vec{0}$ then

$$u^T(\nabla^2 g(x))u = u^T(H+C)u = u^T Hu + u^T Cu = u^T Hu + \sum_{i=1}^{n} c_i u_i^2 x_i^2.$$

Since H is positive definite, $u^T Hu > 0$ and because $c_i > 0$, $u^T Cu > 0$, so

$$u^T(\nabla^2 g(x))u > 0$$

That means that the Hessian of g is positive definite and therefore strictly convex and has at most one local minimum.

2.5 Equivalence of steepest decent method and forward Euler method

2.6 Optimal α in the steepest decent method

To find the optimal α,

$$g\left(\mathbf{x}^{(k+1)}\right) = g\left(\mathbf{x}^{(k)} - \alpha^{(k)} \nabla g(\mathbf{x}^{(k+1)})\right)$$

has to be minimal, so $\frac{\partial}{\partial \alpha} g(\mathbf{x}^{(k+1)})$ has to be zero. This leads to the equation

$$\left(\mathbf{b}\nabla g - \mathbf{x}H\mathbf{x} - \frac{1}{3}\mathbf{x}H\nabla g\right) + ((\nabla g)(H+C)\nabla g)\,\alpha$$
$$+\left(-\sum_{i=1}^{n} c_i x_i (\nabla g)_i^3\right)\alpha^2 + \left(\sum_{i=1}^{n} c_i (\nabla g)_i^4\right)\alpha^3 = 0$$

Because the function is cubic, there has to be at least one real zero. The function is always rising. So only one zero is possible

3 Main algorithms

3.1 Generation of the data

The data is generated in the function *data*.

```
function [ b, H, c ] = data
        b = [ 1; 0 ];
        c = [ 200; 400 ];
        H = [ 200, 200 ; 200, 200 ];
end
```

3.2 Function, gradient and Hessian of g

```
function [ g ] = g ( X )
        [ b, H, c ] = data;
        dim = size(H,1);
        C = zeros ( dim, dim );
        for i = 1 : dim
                C(i,i) = c(i) * X(i) * X(i);
        end
        g = - b' * X + 0.5 * X' * H * X + 1/12 * X' * C * X;
end
```

```
function [ nablaG ] = grad( X )
        [ b, H, c ] = data;
        dim = size ( H, 1 );
        for i = 1 : dim
            C(i,i) = c(i) * X(i) * X(i);
        end
        nablaG = - b + H * X + 1/3 * C * X;
end
```

```
function [ hessG ] = hessian( X )
        [ ~, H, c ] = data;
        dim = size ( H, 1 );
        C = zeros ( dim, dim );
        for i = 1 : dim
                C(i,i) = c(i) * X(i) * X(i);
        end
        hessG = H + C;
end
```

3.3 Minimum searching algorithm

All methods work similar. The only line which has to be changed is line 9. The real implementation is more complicated and is shown in the next chapter.

```
 1  X = [ 4; -1 ];
 2  maxiterations = 1000;
 3  tol = 1e-6;
 4
 5  norm_old = norm ( grad ( X) );
 6  condition = 1;
 7  while condition
 8          maxiterations = maxiterations - 1;
 9          X = X - alpha * grad ( X ); % steepest decent method with constant alpha
10          % X = X - optimalAlpha ( X ) * grad ( X ); % with optimal alpha
11          % X = X - linsolve ( hessian ( X ), grad ( X ) ); % newton method
12          residual = norm ( grad ( X ) ) / norm_old;
13          condition = (maxiterations > 0) && ( residual > tol );
14  end
```

4 Structure of the project

There are two main methods, *compareCPU* and *plotAll*, which are pretty similar. The functions called by *plotAll* return the necessary data to generate graphs, and the functions called by *compareCPU* is optimized to get the CPU time.

4.1 compareCPU

In this method the maximal number of iterations, the tolerance and the starting point is defined. *compareCPU* calls the function *fastmethod* with different parameters named *method*. If this parameter is *steepest-newton*, first steepest decent method is called, then the newton method. If this parameter is *newton*, *steepest* or a number, the line 9 of the last code fragment is changed with the method delta. If a number is given, it is used as step size for the steepest decent method with constant α.

4.2 plotAll

In this method there are the same definitions as in *compareCPU*. Then it calls the function *plot*, which does nearly the same as *fastmethod*, except it does not count the CPU time, but returns the information to generate the graphs.

5 Results

All the values of the table are average values of a 1000 values sample, calculated from the starting point [4;-1]. As we can see, regardless of the tolerance and the maximun number of iterations, Newton's method is the fastest method. When we have a large number of iterations, the combination of steepest and Newton's method is the most precise (smallest residual). For this last method, the precision is increasing while the tolerance is decreasing but obviously, the cputime increases too. The first method, steepest descent with arbitrary alpha appears to be the worst method. Indeed, the cputime and the residual are greater there. But, what is the interest of steepest method with the optimal alpha ? Actually, this method gives us a precise idea of where is the minimum of the function, without being very precise. The Newton's method can give us the same idea, but if we start far from the answer, the time of execution will be very higher. Moreover, if there are several points where the gradient equals to zero, we can't be sure that we are precisely in a global minimum or a local extremum. That is why it seems to be very useful to combine the two methods, because with the steepest descent we approach the solution to a certain point (and we are sure that is the solution) and with the Newton's method applied to this point we can go much more closer to the minimum. For instance, here with the tolerance 10e-6 and 1000 iterations, the precision of the Newton-steepest method is 1 million times more precise than the Newton's method alone.

As a conclusion, we have to choose the method accorded to the result we expect : if we just want a quick idea of the solution, we can just use the steepest descent with an alpha, arbitrary or optimal. But, if we prefer to be certain of the solution, we have to mix Newton's and steespest methods. The Newton's method is between the last ones : very fast and quite precise but we cannot be completely sure of the solution, given that it will converge to the closest zero."