Chapter 7. Large-Scale Unconstrained Optimization

Basic ideas:

* thousands or millions of variables
* Section 5.2 on non-linear CG applies
* Sparse linear algebra or iterative linear algebra
  ⇒ Can get excellent superlinear conv. performance with the right parameters!
* Another variation (sec. 7.2) allows Hessian approximations with just a few n-dim vectors & robust, inexpensive but don't converge fast.
* Partial separability can also give great solutions.

7.1 Inexact Newton Methods

Basic idea:

Solve $\nabla^2 f_k \, p_k^N = -\nabla f_k$

to find an approximate p_k^N.

- Use CG or Lanczos methods that also handle -ve Hessians
- Hessian-free and fast convergence.

Local Convergence of inexact Newton Methods

Define a forcing sequence η_k.

Apply CG iterations until:

$$\|r_k\| \leq \eta_k \|\nabla f_k\| \quad \text{—} \; * $$

where:

$$r_k = \nabla^2 f_k p_k + \nabla f_k$$

Two powerful convergence theorems.

Thm 7.1 Assume:

* $\nabla^2 f(x)$ exists, cts in nghd of x^*
* $\nabla^2 f(x)$ positive def.
* $x_{k+1} = x_k + p_k$ and p_k satisfies (*)

  with: $\eta_k \leq \eta$ and $\eta \in [0,1)$.

If x_0 is sufficiently close to x^*, then:

$x_k \longrightarrow x^*$ and

$$\|\nabla^2 f(x^*)(x_{k+1} - x^*)\| \leq \hat{\eta} \|\nabla^2 f(x^*)(x_k - x^*)\|$$

for some $\hat{\eta}$ with $\eta < \hat{\eta} < 1$.

Thm 7.2 (Excellent!)

Assume conditions of thm 7.1 hold.
Also assume $x_k \to x^*$.

Then

1. Convergence is superlinear if
$$\eta_k \to 0.$$

2. Convergence is quadratic if
$$\eta_k \to 0,\ \eta_k = O(\|\nabla f_k\|),$$
$\nabla^2 f(x)$ is Lipschitz conts for x near x^*.

Comments:

* Can get superlinear convergence for $\eta_k = \min(0.5, \sqrt{\|\nabla f_k\|})$

* Can get quadratic convergence for $\eta_k = \min(0.5, \|\nabla f_k\|)$.

* Our results are great for local analysis. Can also do better for "global" analysis.

Problem for going "global"

* CG only works for pos. def. Hessians

<u>and</u>

* $\nabla^2 f_k$ may not be pos. def. away from x_*

Soln #1:

* Terminate CG for negative curvature
  $\Rightarrow$ still have a descent direction.

<u>and</u>

* make sure step-length satisfies acceptance criteria.

Notation:

$$B_k \underline{p} = -\nabla f_k \quad , \quad B_k = \nabla^2 f_k$$

<u>CG</u>:
* d_j for search directions.
* iterates: z_k (approx. to soln).
* ϵ_k for tolerance

Also, $\eta_k = \min(0.5, \sqrt{\|\nabla f_k\|})$ in example (but can also do quadratic.)

Algo 7.1 (Line-search Newton-CG)

Start at x_0.

for $k = 0, 1, 2, \ldots$

$\epsilon_k = \min(0.5, \sqrt{\|\nabla f_k\|})\,\|\nabla f_k\|$

$z_0 = 0$ (initial CG approx).

$r_0 = \nabla f_k$ (initial residual)

$d_0 = -r_0$ (first CG dirn).

CG:

for $j = 0, 1, 2, \ldots$

if $d_j^T B_k d_j \leq 0$ (-ve curvature).

if $j = 0$

$p_k = -\nabla f_k$, Break out of CG.

else

$p_k = z_j$, Break out of CG.

Positive curvatures:

$\alpha_j = r_j^T r_j / d_j^T B_k d_j$

$z_{j+1} = z_j + \alpha_j d_j$

$r_{j+1} = r_j + \alpha_j B_k d_j$

We can aproximate this product

<u>If</u> $\|r_{j+1}\| < \epsilon_k$

$p_k = z_{j+1}$, <u>Break out of CG.</u>

$$\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j$$

$$d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$$

<u>end</u> (CG iterations)

<u>Set</u> $x_{k+1} = x_k + \alpha_k p_k$ <u>where</u>
α_k satisfies the Wolfe, Goldstein, or Armijo backtracking (start 1-D searching with $\alpha_k = 1$).

end (entire algorithm)

Comments:

* Can also do Pre-conditioning. Why? (to be covered)

* The biggest concern is $B_k d_j$.

"Revolutionary idea!"

Basic idea:

$$\boxed{B_k d_j = \nabla^2 f_k d \approx \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h}}$$

for h "small" but "not too-small" (see lecture on finite differencing).

* This results in evaluating $\nabla f(x_k + hd)$ for each iteration in CG.

* Overhead of $\nabla f(x_k + hd)$ OK since max number of iterations is n.

* Overhead OK compared to the n^2 second derivatives of B_k that are impossible to get!

Why Pre-conditioning?

* It all comes from the rates of convergence of CG.

Recall:

(pos. def.)

Thm 5.5: Suppose B has $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.

Then:

$$\|p_{k+1} - p^*\|_A^2 \leq \frac{(\lambda_{n-k} - \lambda_1)^2}{(\lambda_{n-k} + \lambda_1)^2} \|p_0 - p^*\|_A^2$$

Here, we are looking on how many iterations $(k+1)$ are needed to get convergence.

If $k = n-1$, then $(\lambda_{n-(n-1)} - \lambda_1)^2 = 0$ convergence!

To get there faster, try to modify B to have $\lambda_{n-k} \simeq \lambda_1$ for k "small".

It will clearly work for $\lambda_n = \lambda_1$!

Practical Pre-conditioning (p. 120)

* Symmetric Successive over-relaxation (SSOR) or
* Incomplete Cholesky
* Any great approach will yield:

$$C^{-T} A C^{-1} \approx I$$

which solves for $\hat{x}$ and gets back to x using $x = C^{-1}\hat{x}$.

* Note that in this chapter, the "Revolutionary Idea" allows you to avoid B_k altogether and some of these methods will need B_k.
* Do not use CG if B_k has $\lambda_n >> \lambda_1$.

Forward-diff Solus:

Estimate gradient using:

$$\frac{\partial f}{\partial x_i}(x) \simeq \frac{f(x+\epsilon e_i) - f(x)}{\epsilon} \longrightarrow \Delta$$

where $e_i = [0, \ldots, 0, 1, 0, \ldots, 0]$
ith position.

To get ϵ, follow 8.1.

From p. 614, if numbers are represented by:

$$\sum_{i=1}^{t} d_i 2^{-i} \times 2^e = N$$

where e is the exponent,

$$N = 0.d_1 d_2 \cdots d_t \times 2^e$$

Then $\boxed{\text{unit-roundoff} = u = 2^{-t-1}}$

It only depends on the number of bits used in fractional part.

Then, in Ⓐ, eqn (8.6) recommends $\boxed{\epsilon = \sqrt{u}}$.

So, back in our "revolutionary idea", $h = \sqrt{u}$ as a first approximation.

If we are applying finite-diff for both ∇f_k and $\nabla^2 f_k d$, then the t will be much less! Try $u = 2^{-t'-1}$ with $t' < t$!

Automatic diff also possible with Macsyma. Also see www.autodiff.org.

Trust-Region Newton-CG

Basic idea:

Solve $\min_{p \in \mathbb{R}^n} m_k(p)$, $\|p\| \leq \Delta_k$ — (*)

where: $m_k(p) = f_k + (\nabla f_k)^T p + \frac{1}{2} p^T B_k p$

Algo 7.2 (CG-Steihaug) (for p_k determination)

$z_0 = 0$, $r_0 = \nabla f_k$, $d_0 = -r_0$

if $\|r_0\| < \varepsilon_k$

  return $p_k = 0$ & **declare convergence** to **Algo 4.1** (trust-region method)

Negative Curvature:

if $d_k^T (B_k d_j) \leq 0$

  Find τ that solves (*) for $p_k = z_j + \tau d_j$

  return p_k to Algo 4.1

end

$\alpha_j = r_j^T r_j \,/\, d_j (B_k d_j)$

$z_{j+1} = z_j + \alpha_j d_j$

<u>if</u> $\|z_{j+1}\| \geq \Delta_k$ (outside trust region)

Find $\tau \geq 0$ so that $p_k = z_j + \tau d_j$ satisfies $\|p_k\| = \Delta_k$

<u>return</u> p_k to Algo 4.1.

<u>end</u>

$r_{j+1} = r_j + \alpha_j (B_k d_j)$

<u>if</u> $\|r_{j+1}\| < \epsilon_k$

<u>return</u> $p_k = z_{k+1}$ to Algo 4.1

$\beta_{j+1} = r_{j+1}^T r_{j+1} \,/\, r_j^T r_j$

$d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$

<u>end</u> (CG-method)

See Algo 7.1 for ϵ_k

Notes:

* If $\|\nabla f_k\| \geq \epsilon_k$, we will do better than the Cauchy point: $m_k(p_k) \leq m_k(p_k^c)$
* Thus, the algo is globally convergent.

* From Thm 7.3:

$$0 = \|z_0\|_2 < \cdots < \|z_j\|_2 < \cdots < \|p_k\|_2 \leq \Delta_k$$

so that the algorith increases $\|\cdot\|$ as it goes.

* Can do at-least ½ (in decreasing obj fun) as good as opt if $B_k = \nabla^2 f_k$ pos. def.

* Pre-conditioning can speed things up (see Algo 7.3) for Cholesky.

* Can do Lanczos for negative Curvature solutions.

7.2 Limited-Memory Quasi-Newton Methods

L-BFGS

Recall BFGS:

$$x_{k+1} = x_k - \alpha_k (H_k \nabla f_k)$$

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T$$

Where:

$$\rho_k = 1 / y_k^T s_k, \quad V_k = I - \rho_k y_k s_k^T$$

$$s_k = x_{k+1} - x_k, \quad y_k = \nabla f_{k+1} - \nabla f_k$$

Modify using:

* most recent $\{s_i, y_i\}$ to represent H_k
* Use H_k from previous iteration (unlike before) to approximate new H_k.

Use (by direct subst into defs):

$$H_k = (V_{k-1}^T \cdots V_{k-m}^T)\, H_k^0\, (V_{k-m} \cdots V_{k-1})$$
$$+ \rho_{k-m} (V_{k-1}^T \cdots V_{k-m+1}^T)\, s_{k-m} s_{k-m}^T\, (V_{k-m+1} \cdots V_{k-1})$$
$$+ \rho_{k-m+1} (V_{k-1}^T \cdots V_{k-m+2}^T)\, s_{k-m+1} s_{k-m+1}^T\, (V_{k-m+2} \cdots V_{k-1})$$
$$+ \cdots$$
$$+ \rho_{k-1} s_{k-1} s_{k-1}^T$$

Compute $H_k \nabla f_k$ using $\begin{cases} H_k^0 \text{ (sparse!)} \\ \{s_i, y_i\}_{i=k-1,\ldots,k-m} \end{cases}$

Algo 7.4

$q \leftarrow \nabla f_k$

for $i = k-1, k-2, \ldots, k-m$

$\quad \alpha_i \leftarrow \rho_i s_i^T q$

$\quad q \leftarrow q - \alpha_i y_i$

end

$r \leftarrow H_k^0 q$

for $i = k-m, k-m+1, \ldots, k-1$

$\quad \beta \leftarrow \rho_i y_i^T r$

$r \leftarrow r + s_i(\alpha_i - \beta)$

end

return r $(= H_k \nabla f_k)$

Computanional complexity

* $4mn$ multiplies from: $\simeq$
  m iterations $*$ (n mults in $(s_i^T \underline{q})$
  $+ 1$ mults in $\rho_i (s_i^T \underline{q})$
  $+ (n+1)$ mults in $\rho_i (y_i r)$
  $+ n$ mults in $\alpha_i y_i$
  $+ n$ mults in $s_i(\alpha_i - \beta)$
* Should also count memory accesses, number of adds/subs.
* H_k^0 should be diagonal.
* Do $(H_k^0 \underline{q}) = r$ by solving $B_k^0 r = \underline{q}$ from an initial approx to Hessian.

The recommendation is to use:

$$H_k^0 = \gamma_k I, \quad \gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \quad (*)$$ (see (6.21))

L-BFGS

Start at x_0, $m > 0$.

$k \leftarrow 0$.

Repeat

Choose H_k^0 based on $(*)$.

Use Algo 7.4 to get:
$p_k \leftarrow -H_k \nabla f_k$

Choose α_k so that
$x_{k+1} \leftarrow x_k + \alpha_k p_k$ satisfies
Wolfe conds (or strong ones)

if $k > m$
Discard $\{s_{k-m}, y_{k-m}\}$
and use $s_k \leftarrow x_{k+1} - x_k$, $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$
end

$k \leftarrow k+1$

until convergence

<u>Same as std BFGS if:</u>

* during first $m-1$ iterations
* $H_k^0 = H_0$ at each iteration

* <u>Investigate by varying m, n</u>
* excellent for non-sparse Hessians
* Vary m, n to give small values of <u>number of gradient evals</u> (nfg)
* larger m work better.
* Small m can have problems.

* Book example:

  $m = 3, 5$ fails

  $m = 17$ can be better than $m = 29$

  $n = 1000, 1500, 110$.

* may outperform Hessian-Free such as Newton-CG that uses finite diff or automatic diff

Weakness:

* L-BFGS converges slowly on ill-cond problems, specifically if the eigenvalues are widely distributed
* Non-linear CG may compete with L-BFGS on certain apps.

Memoryless BFGS

Start from (5.46):

$$s_k = \alpha_k p_k$$

Require $\hat{H}_{k+1} y_k = s_k$ to get

$$H_{k+1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right)\left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k}$$

which is L-BFGS with one update.

... related to other CG, but is better.

General Limited-Memory Updating

Useful for:

* Constrained Optimization: ch. 18 SQP, Ch19. Interior-Point Methods for Non-linear Prog (hot)
* Updating the Hessian B_k (instead of its inverse H_k)

Assume $B_k^{-1} = H_k$

Thm 7.4 Let B_0 symmetric pos def. Assume

* $s_i^T y_i > 0$, $i = 0, \dots, k-1$ are stored
* B_k updated using $\{s_i, y_i\}$ (BFGS)

We then have:

$$B_k = B_0 - [B_0 S_k \;\; Y_k] \begin{bmatrix} S_k^T B_0 S_k & L_k \\ L_k^T & -D_k \end{bmatrix} \begin{bmatrix} S_k^T B_0 \\ Y_k^T \end{bmatrix}$$

where: S_k, Y_k are $n \times k$

$$S_k = \begin{bmatrix} \uparrow & & \uparrow \\ s_0 & \cdots & s_{k-1} \\ \downarrow & & \downarrow \end{bmatrix}, \quad Y_k^T = \begin{bmatrix} \leftarrow & y_0 & \rightarrow \\ & \vdots & \\ \leftarrow & y_{k-1} & \rightarrow \end{bmatrix}$$

$$(L_k)_{i,j} = \begin{cases} s_{i-1}^T y_{j-1} & i > j \\ 0, & \text{otherwise} \end{cases}$$

Small: $k \times k$ matrix

$$D_k = \operatorname{diag}[s_0^T y_0 \cdots s_{k-1}^T y_{k-1}]$$

Here, $s_i^T y_i > 0$ gives that the middle matrix is invertible (not shown here)

Basic idea: keep the most recent m pairs. Update looks like:

$$B_k = \delta_k I + \underbrace{[\;\;]}_{n \times (2k)} \underbrace{[\;\;]}_{(2k)\times(2k)} \underbrace{[\;\;]}_{(2k)\times n}$$

$k = 1, \ldots, m$

when it is "filled up"

starting from:

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$$

After we have had at-least m iterations, we continue with the last m-vectors:

$$S_k = \begin{bmatrix} \uparrow & & \uparrow \\ s_{k-m} & \cdots & s_{k-1} \\ \downarrow & & \downarrow \end{bmatrix}, \quad Y_k^T = \begin{bmatrix} \leftarrow & y_{k-m} & \rightarrow \\ & \vdots & \\ \leftarrow & y_{k-1} & \rightarrow \end{bmatrix}$$

$n \times m$ $\qquad$ $m \times n$

<u>Computational complexity</u>

* B_k updating requires $2mn + O(m^3)$
* $B_k v$ ops requires $(4m+1)n + O(m^2)$ multiplies
* m is small.

<u>Key advantage:</u>

* B_k can be used for trust-region <u>and</u> constrained opt., esp. interior point code of section 19.3 ("HOT").
* Codes: L-BFGS-B, IPOPT, KNITRO (see page 183).

7.4 Algorithms for partially separable functions

Basic ideas:

* Use a separable function expression decomposition and sparse matrix-vector, vector-vector ops to compute everything.

Eg: $f(x) = f_1(x_1, x_3) + f_2(x_2, x_4, x_6) + f_3(x_5)$

More generally, if possible, we may have:

$$f(x) = \sum_{i=1}^{ne} f_i(x)$$

where: f_i depends on "few" x_i.

Then:

$$\nabla f(x) = \sum_{i=1}^{ne} \nabla f_i(x)$$

$$\nabla^2 f(x) = \sum_{i=1}^{ne} \nabla^2 f_i(x)$$

We still need to construct sparse approximations to B_k, ∇f_k and then solve:

$$B_k p_k = -\nabla f_k$$

Success stories:

* LANCELOT uses SR1 update ...
* AMPL detects partially separable structure of f and uses it in working with $\nabla^2 f(x)$

7.5 Perspectives on SW

* Free: TN/TNBC [220], TNPACK [275]
* L-BFGS + (Thm 7.4 & Section 19.3 of book) (prob 7.1)