ORIE 7391: Faster: Algorithmic Ideas for Speeding Up Optimization

Least squares

Professor Udell

Operations Research and Information Engineering
Cornell

February 2, 2022

solving a linear system

given **design matrix** $X \in \mathbf{R}^{m \times n}$, **righthand side** (rhs) $y \in \mathbf{R}^m$.
find w so that

$$Xw = y.$$

solving a linear system

given **design matrix** $X \in \mathbf{R}^{m \times n}$, **righthand side** (rhs) $y \in \mathbf{R}^m$.
find w so that

$$Xw = y.$$

how?

- factor and solve
  - QR
  - singular value decomposition (SVD)
  - Cholesky (for symmetric X)
- iterative methods
  - conjugate gradient (CG)
  - iterative refinement

we will talk about QR and CG

considerations in choosing a method

- one problem, or many righthand sides y with the same design matrix X?
- sparse or dense X?
- symmetric X or rectangular problem?

optimality condition for least squares is a linear system

given $X \in \mathbf{R}^{m \times n}$, $y \in \mathbf{R}^m$. find w to solve

$$\text{minimize} \qquad \|Xw - y\|^2.$$

to solve, take gradient, set to 0. solution x satisfies **normal equations**

$$X^T X w = X^T y.$$

a linear system! (with symmetric positive semidefinite design matrix $X^T X$.)

Outline

QR

Conjugate gradient

Iterative refinement

The fundamental theorem of numerical analysis

Theorem

Never form the inverse (or pseudoinverse) of a matrix explicitly.

(Numerically unstable.)

Corollary: never type `inv(X'*X)` or `pinv(X'*X)` to solve the normal equations.

The fundamental theorem of numerical analysis

Theorem

Never form the inverse (or pseudoinverse) of a matrix explicitly.

(Numerically unstable.)

Corollary: never type `inv(X'*X)` or `pinv(X'*X)` to solve the normal equations.

Instead: compute the inverse using easier matrices to invert, like

- Orthogonal matrices Q:

$$a = Qb \iff Q^T a = b$$

- Triangular matrices R:
  if $a = Rb$, can find b given R and a by solving sequence of simple, stable equations.

The QR factorization

every matrix X can be written using **QR decomposition** as $X = QR$

- $Q \in \mathbf{R}^{n \times d}$ has orthogonal columns: $Q^\top Q = I_d$
- $R \in \mathbf{R}^{d \times d}$ is upper triangular: $R_{ij} = 0$ for $i > j$
- diagonal of $R \in \mathbf{R}^{d \times d}$ is positive: $R_{ii} > 0$ for $i = 1, \ldots, d$
- this factorization always exists and is unique
  (proof by Gram-Schmidt construction)

can compute QR factorization of X in $2nd^2$ flops

The QR factorization

every matrix X can be written using **QR decomposition** as $X = QR$

- $Q \in \mathbf{R}^{n \times d}$ has orthogonal columns: $Q^\top Q = I_d$
- $R \in \mathbf{R}^{d \times d}$ is upper triangular: $R_{ij} = 0$ for $i > j$
- diagonal of $R \in \mathbf{R}^{d \times d}$ is positive: $R_{ii} > 0$ for $i = 1, \ldots, d$
- this factorization always exists and is unique (proof by Gram-Schmidt construction)

can compute QR factorization of X in $2nd^2$ flops

use `scipy.linalg.qr`:

$$\mathsf{Q}, \mathsf{R} = \mathbf{qr}(\mathsf{X})$$

advantage of QR: it's easy to invert R!

QR for least squares

use QR to solve least squares: if $X = QR$,

$$\begin{aligned} X^\top X w &= X^\top y \\ (QR)^\top QRw &= (QR)^\top y \\ R^\top Q^\top QRw &= R^\top Q^\top y \\ R^\top Rw &= R^\top Q^\top y \\ Rw &= Q^\top y \\ w &= R^{-1} Q^\top y \end{aligned}$$

Computational considerations

never form the inverse explicitly: numerically unstable!

instead, use QR factorization:

- compute QR factorization of X ($2nd^2$ flops)
- to compute $w = R^{-1}Q^\top y$
  - form $b = Q^\top y$ ($2nd$ flops)
  - compute $w = R^{-1}b$ by back-substitution (d^2 flops)

Computational considerations

never form the inverse explicitly: numerically unstable!

instead, use QR factorization:

- compute QR factorization of X ($2nd^2$ flops)
- to compute $w = R^{-1}Q^\top y$
  - form $b = Q^\top y$ ($2nd$ flops)
  - compute $w = R^{-1}b$ by back-substitution (d^2 flops)

in julia (or matlab), the **backslash operator** solves least-squares efficiently (usually, using QR)

```
w = X \ y
```

in python, use `numpy.lstsq`

Demo: QR

https://github.com/ORIE4741/demos/QR.ipynb

Sparse QR

complexity of QR depends on the sparsity of Q and R:

- compute QR factorization of X (?? flops)
- to compute $w = R^{-1}Q^\top y$
  - form $b = Q^\top y$ (**nnz**(Q) flops)
  - compute $w = R^{-1}b$ by back-substitution (**nnz**(R) flops)

Q-less QR

during QR, can compute $Q^\top y$ essentially for free!

- compute QR of $\begin{bmatrix} X & y \end{bmatrix}$.

Q-less QR

during QR, can compute $Q^\top y$ essentially for free!

- compute QR of $\begin{bmatrix} X & y \end{bmatrix}$.

or compute it afterwards without forming Q:

$$\begin{aligned} X^T b &= (QR)^T b = R^T Q^T b \\ R^{-1} X^T b &= Q^T b \end{aligned}$$

Cholesky and QR

consider **Gram matrix** $G = X^T X \succeq 0$. if $X = QR$,

$$G = R^T Q^T QR = R^T R$$

this construction gives **Cholesky factorization**

- factors spd matrix into triangular matrices
- Cholesky factors of $X^T X$ have same structure as R

Sparse QR: exercise

- can you guess the sparsity of R given sparsity of X?
- can you change sparity of R by permuting columns of X?

Sparse QR: exercise

- can you guess the sparsity of R given sparsity of X?
- can you change sparity of R by permuting columns of X?

use ‘colamd‘ in Matlab, equivalents in Python and julia

Chordal fill-in

to analyze fill-in

- consider spd matrix, for simplicity
- interpret matrix as directed graph
- form clique tree
- identify fill-in

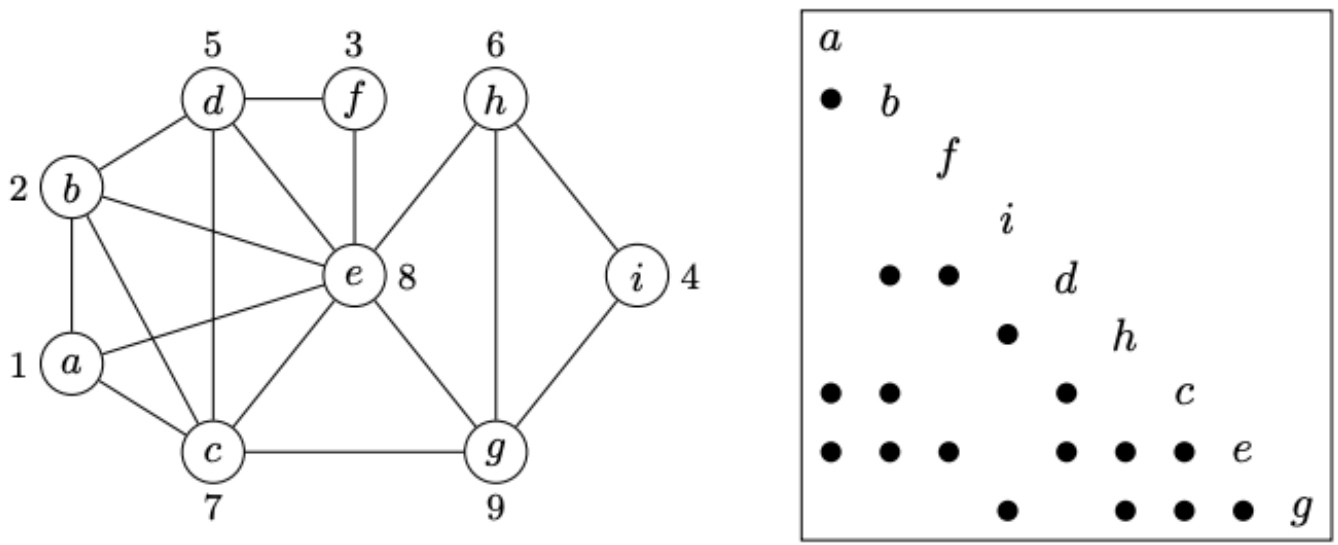

Figure 4.1: *Left.* Filled graph with 9 vertices. The number next to each vertex is the index $\sigma^{-1}(v)$. *Right.* Array representation of the same graph.

source: VA15,

Outline

QR

Conjugate gradient

Iterative refinement

Quiz

Given the eigendecomposition $A = V\Lambda V^T$ of a spd matrix $A \in \mathbf{R}^{n\times n}$, the conjugate gradient method generates the same sequence of objective values on (design matrix, righthand side) (A, b) as on (Λ, Vb).

A. true

B. false

Given a concrete matrix $A \in \mathbf{R}^{n\times n}$ (*e.g.*, as an array of numbers), stochastic trace estimation is a useful way to approximate $\mathbf{tr}(A)$

A. true

B. false

Conjugate gradients

symmetric positive definite system of equations

$$Ax - b, \qquad A \in \mathbf{R}^{n \times n}, \qquad A = A^T \succeq 0$$

Conjugate gradients

symmetric positive definite system of equations

$$Ax - b, \qquad A \in \mathbf{R}^{n \times n}, \qquad A = A^T \succeq 0$$

why use conjugate gradients?

- uses only matrix-vector multiplies with A
  - useful for structured (from PDE or graph) or sparse matrices, easy to parallelize, ...
- most useful for problems with $n > 10^5$ or more
- converges exactly in n iterations
- converges approximately much faster
- quick-and-dirty solve is appropriate **inside** inner loop of optimization algo

Conjugate gradients

symmetric positive definite system of equations

$$Ax - b, \qquad A \in \mathbf{R}^{n \times n}, \qquad A = A^T \succeq 0$$

why use conjugate gradients?

- uses only matrix-vector multiplies with A
  - useful for structured (from PDE or graph) or sparse matrices, easy to parallelize, ...
- most useful for problems with $n > 10^5$ or more
- converges exactly in n iterations
- converges approximately much faster
- quick-and-dirty solve is appropriate **inside** inner loop of optimization algo

other variants for indefinite (MINRES) or nonsymmetric matrices (GMRES)

source: presentation of CG inspired by https://stanford.edu/class/ee364b/lectures/conj_grad_slides.pdf

Iterative methods for least squares

define

- (convex) objective $f(x) = (1/2)x^T Ax - x^T b$
- gradient $\nabla f(x) = Ax - b$
- condition number $\kappa(A) = \sigma_n(A)/\sigma_1(A)$
- bound $R \geq \|x_\star\|$ on norm of solution $x_\star$
- goal: find apx solution within accuracy $f(x) - f(x_\star) \leq \epsilon$

Iterative methods for least squares

define

- (convex) objective $f(x) = (1/2)x^T Ax - x^T b$
- gradient $\nabla f(x) = Ax - b$
- condition number $\kappa(A) = \sigma_n(A)/\sigma_1(A)$
- bound $R \geq \|x_\star\|$ on norm of solution $x_\star$
- goal: find apx solution within accuracy $f(x) - f(x_\star) \leq \epsilon$

how many iterations (matvecs) required?

Iterative methods for least squares

define

- (convex) objective $f(x) = (1/2)x^T Ax - x^T b$
- gradient $\nabla f(x) = Ax - b$
- condition number $\kappa(A) = \sigma_n(A)/\sigma_1(A)$
- bound $R \geq \|x_\star\|$ on norm of solution $x_\star$
- goal: find apx solution within accuracy $f(x) - f(x_\star) \leq \epsilon$

how many iterations (matvecs) required?

- conjugate gradient
  - $O\left(\sqrt{\kappa}\log(\frac{1}{\epsilon})\right)$

Iterative methods for least squares

define

- (convex) objective $f(x) = (1/2)x^T Ax - x^T b$
- gradient $\nabla f(x) = Ax - b$
- condition number $\kappa(A) = \sigma_n(A)/\sigma_1(A)$
- bound $R \geq \|x_\star\|$ on norm of solution $x_\star$
- goal: find apx solution within accuracy $f(x) - f(x_\star) \leq \epsilon$

how many iterations (matvecs) required?

- conjugate gradient
  - $O\left(\sqrt{\kappa}\log(\frac{1}{\epsilon})\right)$
- gradient descent (GD)
  - $O\left(\kappa\log(1/\epsilon)\right)$

Iterative methods for least squares

define

- (convex) objective $f(x) = (1/2)x^T Ax - x^T b$
- gradient $\nabla f(x) = Ax - b$
- condition number $\kappa(A) = \sigma_n(A)/\sigma_1(A)$
- bound $R \geq \|x_\star\|$ on norm of solution $x_\star$
- goal: find apx solution within accuracy $f(x) - f(x_\star) \leq \epsilon$

how many iterations (matvecs) required?

- conjugate gradient
  - $O\left(\sqrt{\kappa}\log(\frac{1}{\epsilon})\right)$
- gradient descent (GD)
  - $O\left(\kappa\log(1/\epsilon)\right)$
- accelerated gradient descent
  - $O\left(\sqrt{\kappa}\log(\frac{R^2}{\epsilon})\right)$ more generalizable, but more parameters to tune

source: [KV16, Bub14]

Residual

define **residual** $r = b - Ax$ at putative solution x

- $r = -\nabla f(x) = A(x_\star - x)$

Residual

define **residual** $r = b - Ax$ at putative solution x

- $r = -\nabla f(x) = A(x_\star - x)$

measures of error:

- objective function $f(x) - f(x_\star)$
- norm of residual $\|r\|$
- norm of gradient $\|\nabla f(x)\|$
- in terms of r, can compute error in objective

$$\begin{aligned} f(x) - f(x_\star) &= \|x - x_\star\|_A \\ &= \frac{1}{2}(x - x_\star)^T A(x - x_\star) \\ &= \frac{1}{2}(r)^T A^{-1}(r) \\ &= \|r\|_{A^{-1}} \end{aligned}$$

Krylov subspace

the Krylov subspace of dimension k is

$$\mathcal{K}_k = \mathrm{span}\{b, Ab, \ldots, A^{k-1}b\} = \mathrm{span}\{p_k(A)b \mid \mathit{degree}(p) < k\}$$

Krylov subspace

the Krylov subspace of dimension k is

$$\mathcal{K}_k = \operatorname{span}\{b, Ab, \ldots, A^{k-1}b\} = \operatorname{span}\{p_k(A)b \mid \mathit{degree}(p) < k\}$$

the iterates of the **Krylov sequence** $x^{(1)}, x^{(2)}, \ldots,$ minimize objective over successive Krylov subspaces

$$x^{(k)} = \operatorname*{argmin}_{x \in \mathcal{K}_k} f(x) = \operatorname*{argmin}_{x \in \mathcal{K}_k} \|Ax - b\| = \operatorname*{argmin}_{x \in \mathcal{K}_k} \|x - x_\star\|_A$$

the CG algorithm generates the Krylov sequence

Properties of Krylov sequence

- $f(x^{(k+1)}) \leq f(x^{(k)})$ (but $\|r\|$ can increase)
- $x^{(n)} = x_\star$
- $x^{(k)} = p_k(A)b$, where p_k is a polynomial with degree $< k$
- less obvious: there is a two-term recurrence

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} + \beta_k (x^{(k)} - x^{(k-1)})$$

Properties of Krylov sequence

- $f(x^{(k+1)}) \leq f(x^{(k)})$ (but $\|r\|$ can increase)
- $x^{(n)} = x_\star$
- $x^{(k)} = p_k(A)b$, where p_k is a polynomial with degree $< k$
- less obvious: there is a two-term recurrence

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} + \beta_k (x^{(k)} - x^{(k-1)})$$

  - α_k and β_k are determined by the CG algorithm

Properties of Krylov sequence

- $f(x^{(k+1)}) \leq f(x^{(k)})$ (but $\|r\|$ can increase)
- $x^{(n)} = x_\star$
- $x^{(k)} = p_k(A)b$, where p_k is a polynomial with degree $< k$
- less obvious: there is a two-term recurrence

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} + \beta_k (x^{(k)} - x^{(k-1)})$$

  - α_k and β_k are determined by the CG algorithm
  - looks like accelerated gradient descent update

Properties of Krylov sequence

- $f(x^{(k+1)}) \leq f(x^{(k)})$ (but $\|r\|$ can increase)
- $x^{(n)} = x_\star$
- $x^{(k)} = p_k(A)b$, where p_k is a polynomial with degree $< k$
- less obvious: there is a two-term recurrence

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} + \beta_k (x^{(k)} - x^{(k-1)})$$

  - α_k and β_k are determined by the CG algorithm
  - looks like accelerated gradient descent update
  - can derive recurrence from optimality conditions: each new iterate $x^{(k+1)}$ must have gradient (residual) orthogonal to $\mathcal{K}_k$

Properties of Krylov sequence

- $f(x^{(k+1)}) \leq f(x^{(k)})$ (but $\|r\|$ can increase)
- $x^{(n)} = x_\star$
- $x^{(k)} = p_k(A)b$, where p_k is a polynomial with degree $< k$
- less obvious: there is a two-term recurrence

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} + \beta_k (x^{(k)} - x^{(k-1)})$$

  - α_k and β_k are determined by the CG algorithm
  - looks like accelerated gradient descent update
  - can derive recurrence from optimality conditions: each new iterate $x^{(k+1)}$ must have gradient (residual) orthogonal to $\mathcal{K}_k$

exercise: derive CG update from Krylov optimality condition

Coordinate descent does not solve in n iterations

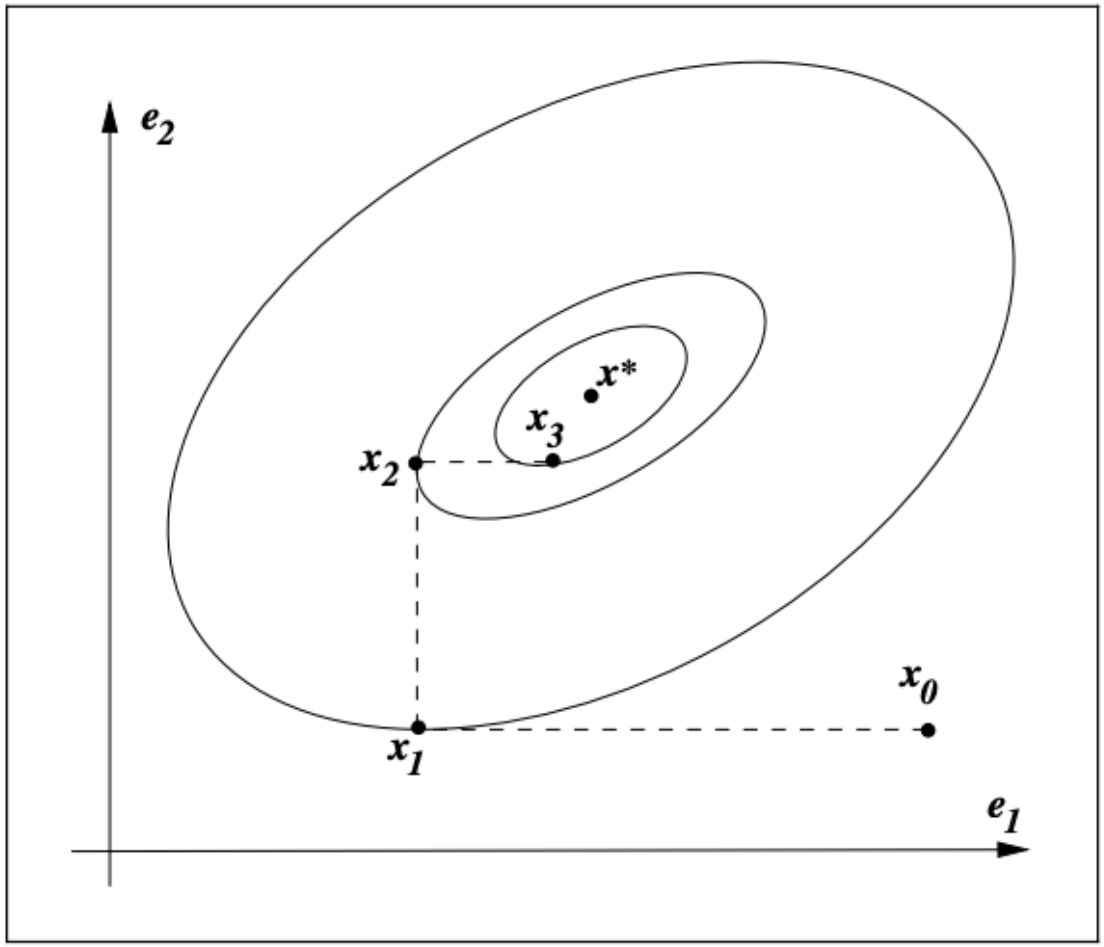

Figure 5.2 Successive minimization along coordinate axes does not find the solution in n iterations, for a general convex quadratic.

CG converges in $\mathrm{Rank}(A)$ iterations

write (don't compute!) SVD of $A = V \Lambda V^T$ with

- $r = \mathrm{Rank}(A)$
- $\Lambda \in \mathbf{R}^r \times r$ diagonal
- $V \in \mathbf{R}^{n \times r}$: orthonormal: $V^T V = I_r$

CG converges in $\mathrm{Rank}(A)$ iterations

write (don't compute!) SVD of $A = V\Lambda V^T$ with

- $r = \mathrm{Rank}(A)$
- $\Lambda \in \mathbf{R}^r \times r$ diagonal
- $V \in \mathbf{R}^{n \times r}$: orthonormal: $V^T V = I_r$

characteristic polynomial of Λ:

$$\xi(s) = \det(sI_r - \Lambda) = (s - \lambda_1) \cdots (s - \lambda_r) = s^r + \alpha s^{r-1} + \cdots + \alpha_r$$

Cayley-Hamilton theorem

$$\begin{aligned} \xi(\Lambda) = 0 &= \Lambda^r + \alpha_1 \Lambda^{r-1} + \cdots + \alpha_r I_r \\ \Lambda^{-1} &= -(1/\alpha_r)(\Lambda^{r-1} + \alpha_1 \Lambda^{r-2} + \cdots + \alpha_{r-1} I_r) \end{aligned}$$

CG converges in $\mathrm{Rank}(A)$ iterations

write (don't compute!) SVD of $A = V\Lambda V^T$ with

- $r = \mathrm{Rank}(A)$
- $\Lambda \in \mathbf{R}^r \times r$ diagonal
- $V \in \mathbf{R}^{n \times r}$: orthonormal: $V^T V = I_r$

characteristic polynomial of Λ:

$$\xi(s) = \det(sI_r - \Lambda) = (s - \lambda_1) \cdots (s - \lambda_r) = s^r + \alpha s^{r-1} + \cdots + \alpha_r$$

Cayley-Hamilton theorem

$$\begin{aligned} \xi(\Lambda) = 0 &= \Lambda^r + \alpha_1 \Lambda^{r-1} + \cdots + \alpha_r I_r \\ \Lambda^{-1} &= -(1/\alpha_r)(\Lambda^{r-1} + \alpha_1 \Lambda^{r-2} + \cdots + \alpha_{r-1} I_r) \end{aligned}$$

write $A^{-1} = V\Lambda^{-1}V^T$ in terms of this decomposition:

$$\begin{aligned} A^{-1} = V\Lambda^{-1}V^T &= = -(1/\alpha_r)(V\Lambda^{r-1}V^T + \alpha_1 V\Lambda^{r-2}V^T + \cdots + \alpha_r \\ &= -(1/\alpha_r)(A^{r-1} + \alpha_1 A^{r-2} + \cdots + \alpha_{r-1} I) \end{aligned}$$

in particular, $x_\star = A^{-1}b \in \mathcal{K}_r$

Outline

QR

Conjugate gradient

Iterative refinement

Iterative refinement

want to solve $Ax = b$.

given approximate solution $Ax^{(0)} \approx b$, for $k = 1, \ldots,$

- compute residual $r^{(k)} = b - Ax^{(k)}$
- use any method to solve $A\delta^{(k)} = r^{(k)}$
- $x^{(k+1)} = x^{(k)} + \delta^{(k)}$


Sébastien Bubeck.
Convex optimization: Algorithms and complexity, 2014.


Sahar Karimi and Stephen A. Vavasis.
A unified convergence bound for conjugate gradient and accelerated gradient, 2016.