Solving Linear Systems: Iterative Methods and Sparse Systems

COS 323

Last time

- Linear system: Ax = b
- Singular and ill-conditioned systems
- Gaussian Elimination: A general purpose method
  - Naïve Gauss (no pivoting)
  - Gauss with partial and full pivoting
  - Asymptotic analysis: $O(n^3)$
- Triangular systems and LU decomposition
- Special matrices and algorithms:
  - Symmetric positive definite: Cholesky decomposition
  - Tridiagonal matrices
- Singularity detection and condition numbers

Today: Methods for large and sparse systems

- Rank-one updating with Sherman-Morrison
- Iterative refinement
- Fixed-point and stationary methods
  - Introduction
  - Iterative refinement as a stationary method
  - Gauss-Seidel and Jacobi methods
  - Successive over-relaxation (SOR)
- Solving a system as an optimization problem
- Representing sparse systems

Problems with large systems

- Gaussian elimination, LU decomposition (factoring step) take O(n^3)
- Expensive for big systems!
- Can get by more easily with special matrices
  - Cholesky decomposition: for symmetric positive definite A; still O(n^3) but halves storage and operations
  - Band-diagonal: O(n) storage and operations
- **What if A is big? (And not diagonal?)**

Special Example: Cyclic Tridiagonal

- Interesting extension: cyclic tridiagonal

$$\begin{bmatrix} a_{11} & a_{12} & & & & a_{16} \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & a_{43} & a_{44} & a_{45} & \\ & & & a_{54} & a_{55} & a_{56} \\ a_{61} & & & & a_{65} & a_{66} \end{bmatrix} x = b$$

- Could derive yet another special case algorithm, but there’s a better way

Updating Inverse

- Suppose we have some fast way of finding A^{-1} for some matrix A
- Now A changes in a special way:

$$A^* = A + uv^T$$

  for some $n \times 1$ vectors u and v
- Goal: find a fast way of computing $(A^*)^{-1}$
  - Eventually, a fast way of solving $(A^*)\,x = b$

Analogue for Scalars

$$Q: \quad \text{Knowing } \frac{1}{\alpha}, \text{ how to compute } \frac{1}{\alpha+\beta}?$$

$$A: \quad \frac{1}{\alpha+\beta} = \frac{1}{\alpha}\left(1 - \frac{\beta/\alpha}{1+\beta/\alpha}\right)$$

Sherman-Morrison Formula

$$\mathbf{A}^* = \mathbf{A} + uv^{\mathrm{T}} = \mathbf{A}(\mathbf{I} + \mathbf{A}^{-1}uv^{\mathrm{T}})$$

$$\left(\mathbf{A}^*\right)^{-1} = (\mathbf{I} + \mathbf{A}^{-1}uv^{\mathrm{T}})^{-1}\,\mathbf{A}^{-1}$$

To check, verify that $(\mathbf{A}^*)^{-1}\mathbf{A}^* = \mathbf{I}$, $\mathbf{A}^*(\mathbf{A}^*)^{-1} = \mathbf{I}$

Sherman-Morrison Formula

$$x = \left(\mathbf{A}^*\right)^{-1} b = \mathbf{A}^{-1} b - \frac{\mathbf{A}^{-1} u\, v^{\mathrm{T}} \mathbf{A}^{-1} b}{1 + v^{\mathrm{T}} \mathbf{A}^{-1} u}$$

So, to solve $\left(\mathbf{A}^*\right) x = b,$

solve $\mathbf{A} y = b, \quad \mathbf{A} z = u, \quad x = y - \dfrac{z\, v^{\mathrm{T}} y}{1 + v^{\mathrm{T}} z}$

Applying Sherman-Morrison

- Let's consider cyclic tridiagonal again:

$$\begin{bmatrix} a_{11} & a_{12} & & & & a_{16} \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & a_{43} & a_{44} & a_{45} & \\ & & & a_{54} & a_{55} & a_{56} \\ a_{61} & & & & a_{65} & a_{66} \end{bmatrix} x = b$$

- Take

$$\mathbf{A} = \begin{bmatrix} a_{11}-1 & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & a_{43} & a_{44} & a_{45} & \\ & & & a_{54} & a_{55} & a_{56} \\ & & & & a_{65} & a_{66}-a_{61}a_{16} \end{bmatrix}, \quad u = \begin{bmatrix} 1 \\ \\ \\ \\ \\ a_{61} \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ \\ \\ \\ \\ a_{16} \end{bmatrix}$$

Applying Sherman-Morrison

- Solve Ay=b, Az=u using special fast algorithm
- Applying Sherman-Morrison takes
  a couple of dot products
- Generalization for several corrections: Woodbury

$$\mathbf{A}^* = \mathbf{A} + \mathbf{U}\mathbf{V}^{\mathrm{T}}$$

$$\left(\mathbf{A}^*\right)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\left(\mathbf{I} + \mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{U}\right)^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}$$

Summary: Sherman-Morrison

- Not just for band-diagonals: S.-M. good for rank-one changes to a matrix whose inverse we know (or can be computed easily)
- O(n^2) (for matrix-vector computations) rather than O(n^3)
- Caution: Error can propogate in repeating S.-M.
- Woodbury formula works for higher-rank changes

Iterative Methods

Direct vs. Iterative Methods

- So far, have looked at *direct methods* for solving linear systems
  - Predictable number of steps
  - No answer until the very end
- Alternative: *iterative methods*
  - Start with approximate answer
  - Each iteration improves accuracy
  - Stop once estimated error below tolerance

Benefits of Iterative Algorithms

- Some iterative algorithms designed for accuracy:
  - Direct methods subject to roundoff error
  - Iterate to reduce error to O(ε)
- Some algorithms produce answer faster
  - Most important class: *sparse matrix* solvers
  - Speed depends on # of *nonzero* elements, not total # of elements

First Iterative Method: Iterative Refinement

- Suppose you've solved (or think you've solved) some system $Ax=b$
- Can check answer by computing *residual*:

$$r = b - Ax_{\text{computed}}$$

- If r is small (compared to b), x is accurate
- What if it's not?

Iterative Refinement

- Large residual caused by error in x:

$$e = x_{\text{correct}} - x_{\text{computed}}$$

- If we knew the error, could try to improve x:

$$x_{\text{correct}} = x_{\text{computed}} + e$$

- Solve for error:

$$\begin{aligned} r &= b - Ax_{\text{computed}} \\ Ax_{\text{computed}} &= A(x_{\text{correct}} - e) = b - r \\ Ax_{\text{correct}} - Ae &= b - r \\ Ae &= r \end{aligned}$$

Iterative Refinement

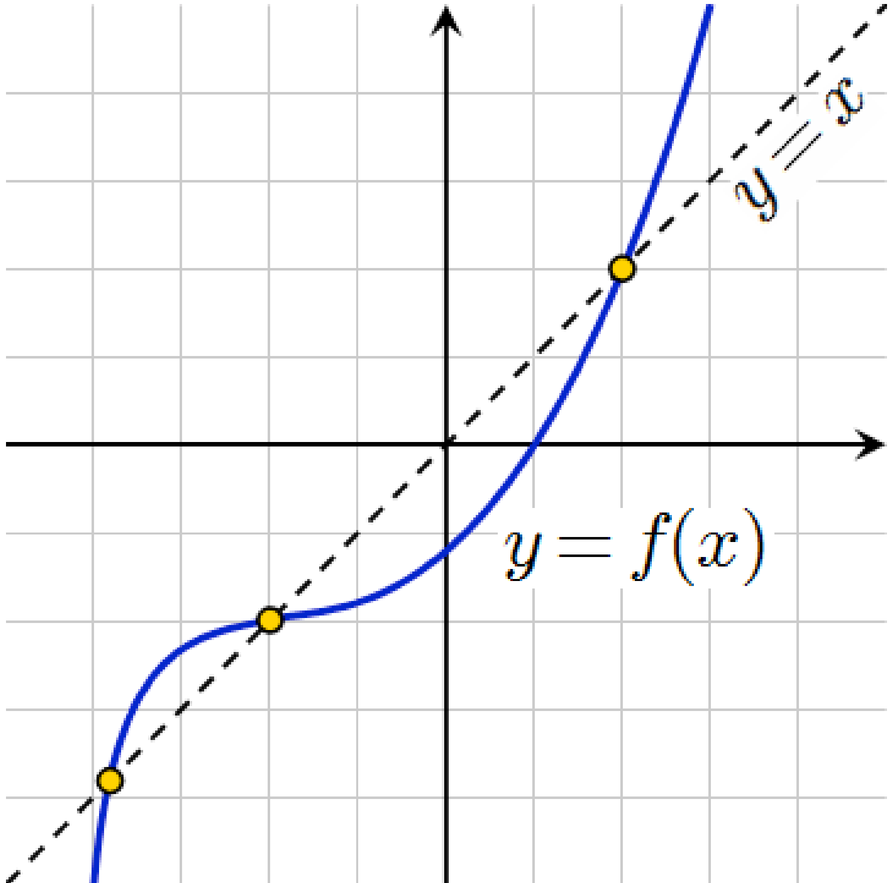
- So, compute residual, solve for e, and apply correction to estimate of x
- If original system solved using LU, this is relatively fast (relative to $O(n^3)$, that is):
  - $O(n^2)$ matrix/vector multiplication + $O(n)$ vector subtraction to solve for r
  - $O(n^2)$ forward/backsubstitution to solve for e
  - $O(n)$ vector addition to correct estimate of x
- Requires 2x storage, often requires extra precision for representing residual

Questions?

Fixed-Point and Stationary Methods

Fixed points

- x* is a fixed point of f(x) if x* = f(x*)



Formulating root-finding as fixed-point-finding

- Choose a $g(x)$ such that $g(x)$ has a fixed point at x^* when $f(x^*) = 0$
  - e.g. $f(x) = x^2 - 2x + 3 = 0$
    $g(x) = (x^2 + 3) / 2$
    if $x^* = (x^{*2} + 3) / 2$ then $f(x^*) = 0$
  - Or, $f(x) = \sin(x)$
    $g(x) = \sin(x) + x$
    if $x^* = \sin(x^*) + x^*$ then $f(x^*) = 0$

Fixed-point iteration

Step 1. Choose some initial x^0

Step 2. Iterate:

```
For $i > 0$:
    $x^{(i+1)} = g(x^i)$
Stop when $x^{(i+1)} - x^i <$ threshold.
```

Example

- Compute pi using

  $f(x) = \sin(x)$

  $g(x) = \sin(x) + x$

Notes on fixed-point root-finding

- Sensitive to starting x^0
- $|g'(x)| < 1$ is sufficient for convergence
- Converges linearly (when it converges)

Extending fixed-point iteration to systems of multiple equations

General form:

Step 0. Formulate set of fixed-point equations

$x_1 = g_1(x_1),\ x_2 = g_2(x_2),\ \ldots\ x_n = g_n(x_n)$

Step 1. Choose $x_1^0,\ x_2^0,\ \ldots\ x_n^0$

Step 2. Iterate:

$x_1^{(i+1)} = g_1(x_1^i),\ x_2^{(i+1)} = g_2(x_2^i)$

Example: Fixed point method for 2 equations

$$f_1(x) = (x_1)^2 + x_1x_2 - 10$$

$$f_2(x) = x_2 + 3x_1(x_2)^2 - 57$$

Formulate new equations:

$$g_1(x_1) = \mathrm{sqrt}(10 - x_1x_2)$$

$$g_2(x_2) = \mathrm{sqrt}((57 - x_2)/3x_1)$$

Iteration steps:

$$x_1^{(i+1)} = \mathrm{sqrt}(10 - x_1^i x_2^i)$$

$$x_2^{(i+1)} = \mathrm{sqrt}((57 - x_2^i)/3x_1^i)$$

Stationary Iterative Methods for Linear Systems

- Can we formulate g(x) such that $x^* = g(x^*)$ when $\mathbf{A}x^* - b = 0$?
- Yes: let $\mathbf{A} = \mathbf{M} - \mathbf{N}$ (for any satisfying **M, N**) and let $g(x) = \mathbf{G}x + c = \mathbf{M}^{-1}\mathbf{N}x + \mathbf{M}^{-1}b$
- Check: if $x^* = g(x^*) = \mathbf{M}^{-1}\mathbf{N}x^* + \mathbf{M}^{-1}b$ then

$$\begin{aligned} \mathbf{A}x^* &= (\mathbf{M} - \mathbf{N})(\mathbf{M}^{-1}\mathbf{N}x^* + \mathbf{M}^{-1}b) \\ &= \mathbf{N}x^* + b + \mathbf{N}(\mathbf{M}^{-1}\mathbf{N}x^* + \mathbf{M}^{-1}b) \\ &= \mathbf{N}x^* + \mathbf{b} - \mathbf{N}x^* \\ &= b \end{aligned}$$

So what?

- We have an update equation:
  $x^{(k+1)} = \mathbf{M}^{-1}\mathbf{N}x^{k} + \mathbf{M}^{-1}b$
- Only requires inverse of M, not A
- (FYI: It's "stationary" because **G** and c do not change)

Iterative refinement is a stationary method!

- $x^{(k+1)} = x^k + e$

  $= x^k + A^{-1} r$ for *estimated* A^{-1}

- This is equivalent to choosing

  $g(x) = \mathbf{G}x + c = \mathbf{M}^{-1}\mathbf{N}x + \mathbf{M}^{-1}b$

  where $\mathbf{G} = (\mathbf{I} - \mathbf{B}^{-1}\,\mathbf{A})$ and $c = \mathbf{B}^{-1}\,b$

  (if $\mathbf{B}^{-1}$ is our most recent estimate of $\mathbf{A}^{-1}$)

So what?

- We have an update equation:
  $x^{(k+1)} = \mathbf{M}^{-1}\mathbf{N}x^{k} + \mathbf{M}^{-1}b$
- Only requires inverse of M, not A
- We can choose M to be nicely invertible (e.g., diagonal)

Jacobi Method

- Choose M to be the diagonal of A
- Choose N to be M – A = -(L + U)
  - Note that A != LU here
- So, use update equation:

  $$x^{(k+1)} = D^{-1} (b - (L + U)x^{k})$$

Jacobi method

- Alternate formulation: Recall we've got

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + \cdots + & a_{mn}x_n & = & b_m \end{array}$$

- Store all x_i^k
- In each iteration, set

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}$$

Gauss-Seidel

- Why make a complete pass through components of x using only x_i^k, ignoring the $x_i^{(k+1)}$ we've already computed?

Jacobi: $$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}$$

G.S.: $$x_i^{(k+1)} = \frac{b_i - \sum_{j>i} a_{ij} x_j^{(k)} - \sum_{j<i} a_{ij} x_j^{(k+1)}}{a_{ii}}$$

Notes on Gauss-Seidel

- Gauss-Seidel is also a stationary method
  $A = M - N$ where $M = D + L$, $N = -U$
- Both G.S. and Jacobi may or may not converge
  - Jacobi: Diagonal dominance is sufficient condition
  - G.S.: Diagonal dominance or symmetric positive definite
- Both can be **very slow to converge**

Successive Over-relaxation (SOR)

- Let $x^{(k+1)} = (1-w)x^{(k)} + w\, x_{GS}^{(k+1)}$
- If $w = 1$ then update rule is Gauss-Seidel
- If $w < 1$: Under-relaxation
  - Proceed more cautiously: e.g., to make a non-convergent system converge
- If $1 < w < 2$: Over-relaxation
  - Proceed more boldly, e.g. to accelerate convergence of an already-convergent system
- If $w > 2$: Divergence. ☹

Questions?

One more method: Conjugate Gradients

- Transform problem to a function minimization!

$$\text{Solve } Ax=b$$

$$\Rightarrow \text{ Minimize } f(x) = x^T A x - 2b^T x$$

- To motivate this, consider 1D:

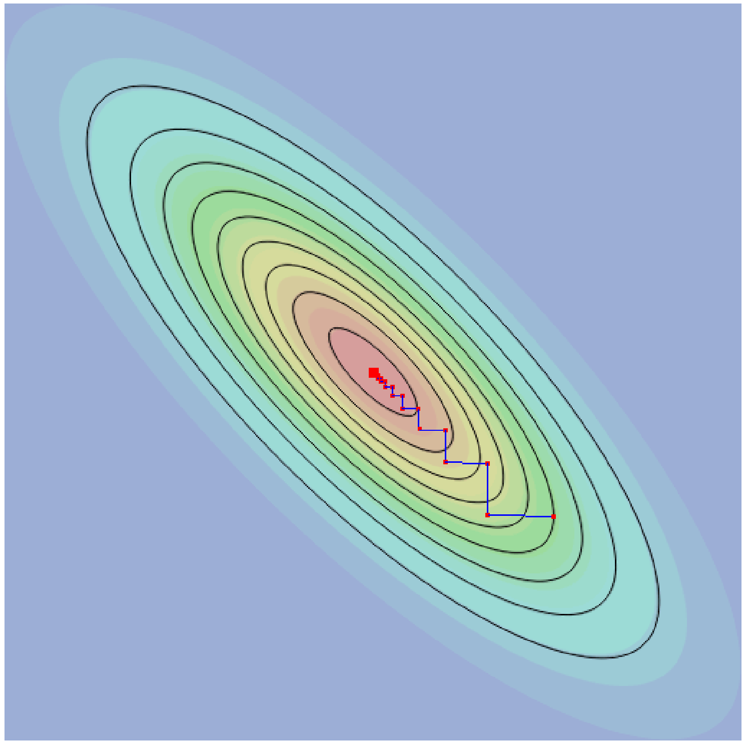
$$f(x) = ax^2 - 2bx$$

$$df/dx = 2ax - 2b = 0$$

$$ax = b$$

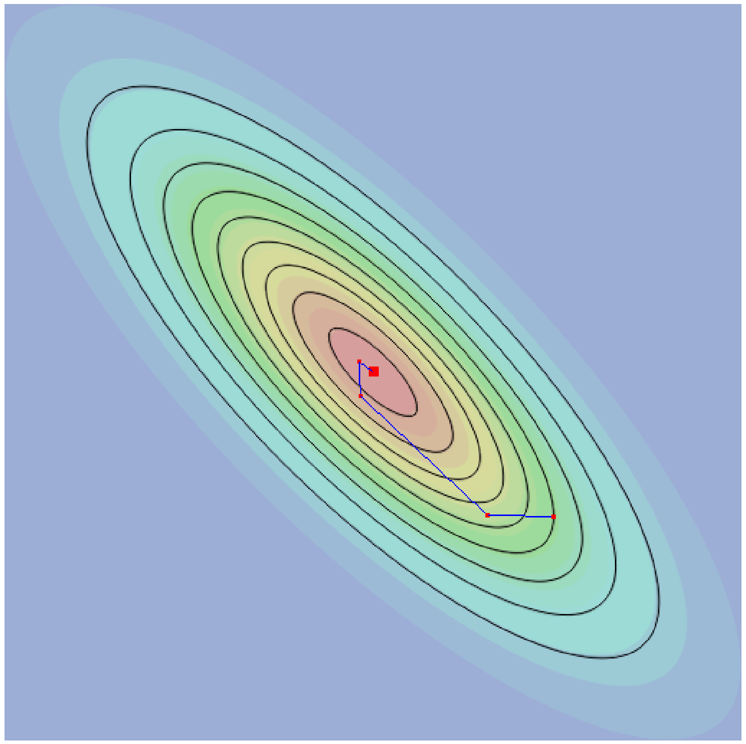
Conjugate Gradient for Linear Systems

- Preferred method: conjugate gradients
- Recall: plain gradient descent has a problem…



Conjugate Gradient for Linear Systems

- … that's solved by conjugate gradients



- Walk along direction

$$d_{k+1} = -g_{k+1} + \beta_k d_k$$

- Polak and Ribiere formula:

$$\beta_k = \frac{g_{k+1}^{\mathrm{T}}(g_{k+1} - g_k)}{g_k^{\mathrm{T}} g_k}$$

Conjugate Gradient is easily computable for linear systems

- If A is symmetric positive definite:
  - At any point, **gradient is negative residual**

$$f(x) = x^{\mathrm{T}} \mathbf{A}\, x - 2b^{\mathrm{T}} x$$

$$\text{so} \quad \nabla f(x) = 2\left(\mathbf{A}x - b\right) = -2r$$

  - Easy to compute: just A multiplied by a vector
- For any search direction s$_k$, can **directly compute minimum** in that direction:

$$x_{k+1} = x_k + \alpha_k x_k$$

$$\text{where} \quad \alpha_k = r_k^T r_k / s_k^T A s_k$$

Conjugate Gradient for Linear Systems

- Just a few matrix-vector multiplies (plus some dot products, etc.) per iteration
- For *m* nonzero entries, each iteration O(max(*m*,*n*))
- Conjugate gradients may need *n* iterations for "perfect" convergence, but often get decent answer well before then
- For non-symmetric matrices: biconjugate gradient

Representing Sparse Systems

Sparse Systems

- Many applications require solution of large linear systems (n = thousands to millions or more)
  - Local constraints or interactions: most entries are 0
  - Wasteful to store all n^2 entries
  - Difficult or impossible to use $O(n^3)$ algorithms
- Goal: solve system with:
  - Storage proportional to # of *nonzero* elements
  - Running time $<< n^3$

Sparse Matrices in General

- Represent sparse matrices by noting which elements are nonzero
- Critical for Av and A^Tv to be efficient: proportional to # of nonzero elements
  - Useful for both conjugate gradient and Sherman-Morrison

Compressed Sparse Row Format

- Three arrays
  - Values: actual numbers in the matrix
  - Cols: column of corresponding entry in values
  - Rows: index of first entry in each row
  - Example: (zero-based! C/C++/Java, not Matlab!)

$$\begin{bmatrix} 0 & 3 & 2 & 3 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

values	3 2 3 2 5 1 2 3
cols	1 2 3 0 3 1 2 3
rows	0 3 5 5 8

Compressed Sparse Row Format

$$\begin{bmatrix} 0 & 3 & 2 & 3 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

values	3 2 3 2 5 1 2 3
cols	1 2 3 0 3 1 2 3
rows	0 3 5 5 8

- Multiplying Ax:

```
for (i = 0; i < n; i++) {
    out[i] = 0;
    for (j = rows[i]; j < rows[i+1]; j++)
        out[i] += values[j] * x[ cols[j] ];
}
```

Summary of Methods for Linear Systems

Method	Benefits	Drawbacks
Forward/backward substitution	Fast (n^2)	Applies only to upper- or lower-triangular matrices
Gaussian elimination	Works for any [non-singular] matrix	$O(n^3)$
LU decomposition	Works for any matrix (singular matrices can still be factored); can re-use L, U for different b values; once factored uses only forward/backward substitution	$O(n^3)$ initial factorization (same process as Gauss)
Cholesky	$O(n^3)$ but with ½ storage and computation of Gauss	Still $O(n^3)$; only for symmetric positive definite
Band-diagonal elimination	$O(w^2n)$ where w = band width	Only for band diagonal

Method	Benefits	Drawbacks
Sherman-Morrison	Update step is $O(n^2)$	Only for rank-1 changes; degrades with repeated iterations (then use Woodbury instead)
Iterative refinement	Can be applied following any solution method	Requires 2x storage, extra precision for residual
Jacobi	More appropriate than elimination for large/sparse systems; can be parallelized	Can diverge when not diagonally dominant; slow
Gauss-Seidel	More appropriate than elimination for large/sparse; a bit more powerful than Jacobi	Can diverge when not diagonnally dominant or symmetric/positive-definite; slow; can't parallelize
SOR	Potentially faster than Jacobi, Gauss-Seidel for large/sparse systems	Requires parameter tuning
Conjugate gradient	Fast(er) for large/sparse systems; often doesn't require all *n* iterations	Requires symmetric positive definite (otherwise use bi-conjugate)