UNIVERSITY OF MINNESOTA TWIN CITIES

A tutorial on: Iterative methods for Sparse Matrix Problems

Yousef Saad

University of Minnesota
Computer Science and Engineering

CRM Montreal - May 3, 2008

Outline

Part 1

- Sparse matrices and sparsity
- Basic iterative techniques
- Projection methods
- Krylov subspace methods

Part 2

- Preconditioned iterations
- Preconditioning techniques

Part 3

- Parallel implementations
- Multigrid methods

Part 4

- Eigenvalue problems
- Applications

PROJECTION METHODS

One-dimensional projection processes

Steepest descent – Problem: $Ax = b$, with A SPD

➤ Define: $f(x) = \frac{1}{2}\|x - x^*\|_A^2 = \frac{1}{2}(A(x - x^*), (x - x^*))$

Note: 1. $f(x) = \frac{1}{2}(Ax, x) - (b, x)$ + constant

2. $\nabla f(x) = Ax - b \longrightarrow$ 'descent' direction = $b - Ax \equiv r$

Idea: take a step of the form $x_{new} = x + \alpha r$ which minimizes $f(x)$.

Best $\alpha = (r, r)/(Ar, r)$.

Iteration:

$$\boxed{\begin{array}{l} r \leftarrow b - Ax, \\ \alpha \leftarrow (r, r)/(Ar, r) \\ x \leftarrow x + \alpha r \end{array}}$$

➤ Can show: convergence guaranteed if A is SPD.

Residual norm steepest descent: Now A is arbitrary

- Minimize instead: $f(x) = \frac{1}{2}\|b - Ax\|_2^2$ in direction $-\nabla f$.

$-\nabla f(x) = A^T(b - Ax) = A^T r$.

Iteration:

$$\boxed{\begin{array}{l} r \leftarrow b - Ax, d = A^T r \\ \alpha \leftarrow \|d\|_2^2 / \|Ad\|_2^2 \\ x \leftarrow x + \alpha d \end{array}}$$

- Important Note: equivalent to usual steepest descent applied to normal equations $A^T Ax = A^T b$.

- Converges under the condition that A is nonsingular.

- But convergence can be very slow

Minimal residual iteration: Assume A is positive definite ($A + A^T$ is SPD).

➤ The objective function is still $\frac{1}{2}\|b - Ax\|_2^2$, but the direction of search is $r = b - Ax$ instead of $-\nabla f(x)$

Iteration:

$$\boxed{\begin{array}{l} r \leftarrow b - Ax, \\ \alpha \leftarrow (Ar, r)/(Ar, Ar) \\ x \leftarrow x + \alpha r \end{array}}$$

➤ Each step minimizes $f(x) = \|b - Ax\|_2^2$ in direction r.

➤ Converges under the condition that $A + A^T$ is SPD.

- Common feature of these techniques: $x_{new} = x + \alpha d$, where d = a certain direction.
- α is defined to optimize a certain quadratic function.
- Equivalent to determining α by an orthogonality constraint.

Example

In MR:
$x(\alpha) = x + \alpha d$, with $d = b - Ax$.
$\min_\alpha \ \|b - Ax(\alpha)\|_2$ reached iff $b - Ax(\alpha) \perp r$

- One-dimensional projection methods – can we generalize to m-dimensional techniques?

General Projection Methods

Initial Problem: $b - Ax = 0$

Given two subspaces K and L of $\mathbb{R}^N$ define the *approximate problem:*

Find $\tilde{x} \in K$ such that $b - A\tilde{x} \perp L$

➤ **Leads to a small linear system (‘projected problems’) This is a basic projection step. Typically: sequence of such steps are applied**

- With a nonzero initial guess x_0, the approximate problem is

$$\text{Find} \quad \tilde{x} \in x_0 + K \quad \text{such that} \quad b - A\tilde{x} \perp L$$

Write $\tilde{x} = x_0 + \delta$ and $r_0 = b - Ax_0$. Leads to a system for δ:

Find $\delta \in K$ such that $r_0 - A\delta \perp L$

Matrix representation:

Let

- $V = [v_1, \ldots, v_m]$ a basis of K &
- $W = [w_1, \ldots, w_m]$ a basis of L

Then letting x be the approximate solution $\tilde{x} = x_0 + \delta \equiv x_0 + Vy$ where y is a vector of $\mathbb{R}^m$, the Petrov-Galerkin condition yields,

$$W^T(r_0 - AVy) = 0$$

and therefore

$$\tilde{x} = x_0 + V[W^TAV]^{-1}W^Tr_0$$

Remark: In practice W^TAV is known from algorithm and has a simple structure [tridiagonal, Hessenberg,..]

Prototype Projection Method

Until Convergence Do:

1. Select a pair of subspaces K, and L;
2. Choose bases $V = [v_1, \ldots, v_m]$ for K and $W = [w_1, \ldots, w_m]$ for L.
3. Compute

$$r \leftarrow b - Ax,$$
$$y \leftarrow (W^T A V)^{-1} W^T r,$$
$$x \leftarrow x + Vy.$$

Two important particular cases.

1. $L = AK$. then $\|b - A\tilde{x}\|_2 = \min_{z \in K} \|b - Az\|_2$

   $\longrightarrow$ class of minimal residual methods: CR, GCR, ORTHOMIN, GMRES, CGNR, ...

2. $L = K$ $\longrightarrow$ class of Galerkin or orthogonal projection methods. When A is SPD then

$$\|x^* - \tilde{x}\|_A = \min_{z \in K} \|x^* - z\|_A.$$

One-dimensional projection processes

$$K = span\{d\}$$
and
$$L = span\{e\}$$

Then $\tilde{x} \leftarrow x + \alpha d$ and Petrov-Galerkin condition $r - A\delta \perp e$ yields

$$\boxed{\alpha = \frac{(r,e)}{(Ad,e)}}$$

(I) Steepest descent: $K = span(r)$, $L = K$

(II) Residual norm steepest descent: $K = span(A^T r)$, $L = AK$

(III) Minimal residual iteration: $K = span(r)$, $L = AK$

Krylov Subspace Methods

Principle: Projection methods on Krylov subspaces:

$$K_m(A, v_1) = \text{span}\{v_1, Av_1, \cdots, A^{m-1}v_1\}$$

- probably the most important class of iterative methods.
- many variants exist depending on the subspace L.

Simple properties of K_m. Let $\mu =$ deg. of minimal polynomial of v

- $K_m = \{p(A)v | p = \text{polynomial of degree} \leq m-1\}$
- $K_m = K_\mu$ for all $m \geq \mu$. Moreover, K_μ is invariant under A.
- $dim(K_m) = m$ iff $\mu \geq m$.

Arnoldi's Algorithm

- Goal: to compute an orthogonal basis of K_m.
- Input: Initial vector v_1, with $\|v_1\|_2 = 1$ and m.

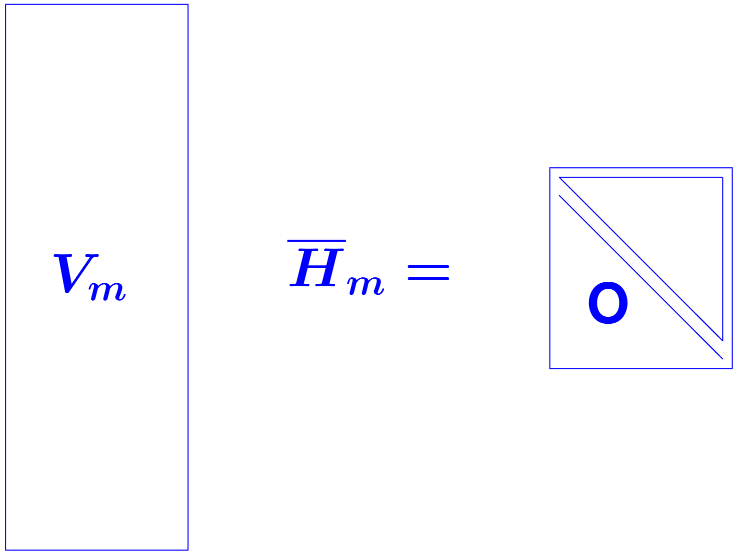
```
For $j = 1, ..., m$ do
    • Compute $w := Av_j$
    • for $i = 1, \ldots, j$, do $\begin{cases} h_{i,j} := (w, v_i) \\ w := w - h_{i,j} v_i \end{cases}$
    • $h_{j+1,j} = \|w\|_2$ and $v_{j+1} = w / h_{j+1,j}$
```

Result of orthogonalization process

1. $V_m = [v_1, v_2, ..., v_m]$ **orthonormal basis of** K_m.

2. $AV_m = V_{m+1}\overline{H}_m$

3. $V_m^T A V_m = H_m \equiv \overline{H}_m -$ **last row.**



Arnoldi's Method $(L_m = K_m)$

- Petrov-Galerkin condition when $L_m = K_m$, shows:

$$x_m = x_0 + V_m H_m^{-1} V_m^T r_0$$

- Select $v_1 = r_0 / \|r_0\|_2 \equiv r_0/\beta$ in Arnoldi's algorithm, then:

$$\boxed{x_m = x_0 + \beta V_m H_m^{-1} e_1}$$

Equivalent	* FOM [YS, 1981] (above formulation)
algorithms:	* Young and Jea's ORTHORES [1982].
	* Axelsson's projection method [1981].

Minimal residual methods ($L_m = AK_m$)

- **When $L_m = AK_m$, we let $W_m \equiv AV_m$ and obtain:**

$$x_m = x_0 + V_m[W_m^T AV_m]^{-1} W_m^T r_0$$

- **Use again $v_1 := r_0/(\beta := \|r_0\|_2)$ and:** $AV_m = V_{m+1}\bar{H}_m$

$$x_m = x_0 + V_m[\bar{H}_m^T \bar{H}_m]^{-1} \bar{H}_m^T \beta e_1 = x_0 + V_m y_m$$

where y_m minimizes $\|\beta e_1 - \bar{H}_m y\|_2$ over $y \in \mathbb{R}^m$. Hence, (Generalized Minimal Residual method (GMRES) [Saad-Schultz, 1983]):

$$x_m = x_0 + V_m y_m \quad \textbf{where} \quad y_m : \min_y \|\beta e_1 - \bar{H}_m y\|_2$$

Equivalent methods:
- **Axelsson's CGLS**
- **Orthomin (1980)**
- **Orthodir**
- **GCR**

Restarting and Truncating

Difficulty: As m increases, storage and work per step increase fast.

First remedy: Restarting. Fix the dimension m of the subspace

ALGORITHM : 1 . *Restarted GMRES (resp. Arnoldi)*

```
1. Start/Restart: Compute $r_0 = b - Ax_0$, and $v_1 = r_0/(\beta := \|r_0\|_2)$.
2. Arnoldi Process: generate $\bar{H}_m$ and $V_m$.
3. Compute    $y_m = H_m^{-1}\beta e_1$ (FOM), or
              $y_m = argmin\|\beta e_1 - \bar{H}_m y\|_2$ (GMRES)
4. $x_m = x_0 + V_m y_m$
5. If $\|r_m\|_2 \leq \epsilon\|r_0\|_2$ stop else set $x_0 := x_m$ and go to 1.
```

Second remedy: Truncate the orthogonalization

The formula for v_{j+1} is replaced by

$$h_{j+1,j}v_{j+1} = Av_j - \sum_{i=j-k+1}^{j} h_{ij}v_i$$

$\longrightarrow$ **each v_j is made orthogonal to the previous k v_i's.**

$\longrightarrow$ **x_m still computed as $x_m = x_0 + V_m H_m^{-1} \beta e_1$.**

$\longrightarrow$ **It can be shown that this is again an oblique projection process.**

➤ **IOM (Incomplete Orthogonalization Method) = replace orthogonalization in FOM, by the above truncated (or ‘incomplete’) orthogonalization.**

The direct version of IOM [DIOM]:

Writing the LU decomposition of H_m as $H_m = L_m U_m$ we get

$$x_m = x_0 + \boxed{V_m U_m^{-1}} \; \boxed{L_m^{-1} \beta e_1} \equiv x_0 + P_m z_m$$

➤ **Structure of L_m, U_m when $k = 3$**

$$L_m = \begin{pmatrix} 1 & & & & & & \\ x & 1 & & & & & \\ & x & 1 & & & & \\ & & x & 1 & & & \\ & & & x & 1 & & \\ & & & & x & 1 & \\ & & & & & x & 1 \end{pmatrix} \quad U_m = \begin{pmatrix} x & x & x & & & & \\ & x & x & x & & & \\ & & x & x & x & & \\ & & & x & x & x & \\ & & & & x & x & x \\ & & & & & x & x \\ & & & & & & x \end{pmatrix}$$

$$p_m = u_{mm}^{-1}[v_m - \sum_{i=m-k+1}^{m-1} u_{im} p_i] \qquad z_m = \begin{bmatrix} z_{m-1} \\ \zeta_m \end{bmatrix}$$

Result: Can update x_m at each step:

$$x_m = x_{m-1} + \zeta_m p_m$$

Note: Several existing pairs of methods have a similar link: they are based on the LU, or other, factorizations of the H_m matrix

- CG-like formulation of IOM called DIOM [Saad, 1982]
- ORTHORES(k) [Young & Jea '82] equivalent to DIOM(k)
- SYMMLQ [Paige and Saunders, '77] uses LQ factorization of H_m.
- Can add partial pivoting to LU factorization of H_m

The Symmetric Case: Observation

Observe: When A is real symmetric then in Arnoldi's method:

$$\boxed{H_m = V_m^T A V_m}$$

must be symmetric. Therefore

THEOREM. When Arnoldi's algorithm is applied to a (real) symmetric matrix then the matrix H_m is symmetric tridiagonal.

In other words:

1) $h_{ij} = 0$ for $|i - j| > 1$

2) $h_{j,j+1} = h_{j+1,j}, \quad j = 1, \ldots, m$

- **We can write**

$$H_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \beta_4 & \\ & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot & \cdot \\ & & & & \beta_m & \alpha_m \end{pmatrix} \tag{1}$$

The v_i's satisfy a three-term recurrence [Lanczos Algorithm]:

$$\boxed{\beta_{j+1} v_{j+1} = A v_j - \alpha_j v_j - \beta_j v_{j-1}}$$

$\longrightarrow$ **simplified version of Arnoldi's algorithm for sym. systems.**

$$\boxed{\text{Symmetric matrix + Arnoldi} \longrightarrow \text{Symmetric Lanczos}}$$

The Lanczos algorithm

ALGORITHM : 2 ▪ ***Lanczos***

```
1. Choose an initial vector $v_1$ of norm unity.
       Set $\beta_1 \equiv 0, v_0 \equiv 0$
2. For $j = 1, 2, \ldots, m$ Do:
3.     $w_j := Av_j - \beta_j v_{j-1}$
4.     $\alpha_j := (w_j, v_j)$
5.     $w_j := w_j - \alpha_j v_j$
6.     $\beta_{j+1} := \|w_j\|_2$. If $\beta_{j+1} = 0$ then Stop
7.     $v_{j+1} := w_j / \beta_{j+1}$
8. EndDo
```

Lanczos algorithm for linear systems

➤ **Usual orthogonal projection method setting:**

- $L_m = K_m = span\{r_0, Ar_0, \ldots, A^{m-1}r_0\}$
- **Basis $V_m = [v_1, \ldots, v_m]$ of K_m generated by the Lanczos algorithm**

➤ **Three different possible implementations.**

(1) Arnoldi-like; (2) Exploit tridigonal nature of H_m (DIOM); (3) Conjugate gradient.

.... following what was done for DIOM..

The Conjugate Gradient Algorithm (A S.P.D.)

- **Note: the p_i's are A-orthogonal**
- **The r_i''s are orthogonal.**
- **And we have $x_m = x_{m-1} + \xi_m p_m$**

So there must be an update of the form:

1. $p_m = r_{m-1} + \beta_m p_{m-1}$
2. $x_m = x_{m-1} + \xi_m p_m$
3. $r_m = r_{m-1} - \xi_m A p_m$

ALGORITHM : 3 . *Conjugate Gradient*

```
Start: $r_0 := b - Ax_0$, $p_0 := r_0$.
Iterate: Until convergence do,
    $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
    $x_{j+1} := x_j + \alpha_j p_j$
    $r_{j+1} := r_j - \alpha_j Ap_j$
    $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
    $p_{j+1} := r_{j+1} + \beta_j p_j$
EndDo
```

- $r_j = scaling \times v_{j+1}$. The r_j's are orthogonal.
- The p_j's are A-conjugate, i.e., $(Ap_i, p_j) = 0$ for $i \neq j$.

METHODS BASED ON LANCZOS BIORTHOGONALIZATION

ALGORITHM : 4 . *Lanczos Bi-Orthogonalization*

```
1. Choose two vectors $v_1, w_1$ such that $(v_1, w_1) = 1$.
2. Set $\beta_1 = \delta_1 \equiv 0$, $w_0 = v_0 \equiv 0$
3. For $j = 1, 2, \ldots, m$ Do:
4.     $\alpha_j = (Av_j, w_j)$
5.     $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6.     $\hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7.     $\delta_{j+1} = |(\hat{v}_{j+1}, \hat{w}_{j+1})|^{1/2}$. If $\delta_{j+1} = 0$ Stop
8.     $\beta_{j+1} = (\hat{v}_{j+1}, \hat{w}_{j+1})/\delta_{j+1}$
9.     $w_{j+1} = \hat{w}_{j+1}/\beta_{j+1}$
10.    $v_{j+1} = \hat{v}_{j+1}/\delta_{j+1}$
11. EndDo
```

- **Extension of the symmetric Lanczos algorithm**
- **Builds a pair of biorthogonal bases for the two subspaces**

$$\mathcal{K}_m(A, v_1) \quad \text{and} \quad \mathcal{K}_m(A^T, w_1)$$

- **Different ways to choose $\delta_{j+1}, \beta_{j+1}$ in lines 7 and 8.**

Let

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \delta_2 & \alpha_2 & \beta_3 & & \\ & . & . & . & \\ & & \delta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \delta_m & \alpha_m \end{pmatrix}.$$

- $v_i \in \mathcal{K}_m(A, v_1)$ **and** $w_j \in \mathcal{K}_m(A^T, w_1)$**.**

If the algorithm does not break down before step m, then the vectors $v_i, i = 1, \ldots, m$, and $w_j, j = 1, \ldots, m$, are biorthogonal, i.e.,

$$(v_j, w_i) = \delta_{ij} \quad 1 \leq i,\ j \leq m\ .$$

Moreover, $\{v_i\}_{i=1,2,\ldots,m}$ is a basis of $\mathcal{K}_m(A, v_1)$ and $\{w_i\}_{i=1,2,\ldots,m}$ is a basis of $\mathcal{K}_m(A^T, w_1)$ and

$$\begin{aligned} AV_m &= V_m T_m + \delta_{m+1} v_{m+1} e_m^T, \\ A^T W_m &= W_m T_m^T + \beta_{m+1} w_{m+1} e_m^T, \\ W_m^T A V_m &= T_m \ . \end{aligned}$$

The Lanczos Algorithm for Linear Systems

ALGORITHM : 5 . ***Lanczos Alg. for Linear Systems***

```
1. Compute $r_0 = b - Ax_0$ and $\beta := \|r_0\|_2$
2. Run $m$ steps of the nonsymmetric Lanczos Algorithm i.e.,
3.     Start with $v_1 := r_0/\beta$, and any $w_1$ such that
       $(v_1, w_1) = 1$
4.     Generate the pair of Lanczos vectors $v_1, \ldots, v_m$,
       and $w_1, \ldots, w_m$
5.     and the tridiagonal matrix $T_m$ from Algorithm ??.
6. Compute $y_m = T_m^{-1}(\beta e_1)$ and $x_m := x_0 + V_m y_m$.
```

➤ **BCG can be derived from the Lanczos Algorithm similarly to CG**

ALGORITHM : 6 . BiConjugate Gradient (BCG)

```
1.  Compute $r_0 := b - Ax_0$.
2.  Choose $r_0^*$ such that $(r_0, r_0^*) \neq 0$;
    Set $p_0 := r_0$, $p_0^* := r_0^*$
3.  For $j = 0, 1, \ldots$, until convergence Do:,
4.      $\alpha_j := (r_j, r_j^*)/(Ap_j, p_j^*)$
5.      $x_{j+1} := x_j + \alpha_j p_j$
6.      $r_{j+1} := r_j - \alpha_j A p_j$
7.      $r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
8.      $\beta_j := (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)$
9.      $p_{j+1} := r_{j+1} + \beta_j p_j$
10.     $p_{j+1}^* := r_{j+1}^* + \beta_j p_j^*$
11. EndDo
```

Quasi-Minimal Residual Algorithm

- **Recall relation from the lanczos algorithm:** $AV_m = V_{m+1}\bar{T}_m$ **with** $\bar{T}_m$ **=** $(m+1) \times m$ **tridiagonal matrix** $\bar{T}_m = \begin{pmatrix} T_m \\ \delta_{m+1} e_m^T \end{pmatrix}$.

- **Let** $v_1 \equiv \beta r_0$ **and** $x = x_0 + V_m y$**. Residual norm** $\|b - Ax\|_2$ **equals**

$$\|r_0 - AV_m y\|_2 = \|\beta v_1 - V_{m+1}\bar{T}_m y\|_2 = \|V_{m+1}\left(\beta e_1 - \bar{T}_m y\right)\|_2$$

- **Column-vectors of** V_{m+1} **are not** $\perp$ **(**$\neq$ **GMRES).**

- **But: reasonable idea to minimize the function** $J(y) \equiv \|\beta e_1 - \bar{T}_m y\|_2$

- **Quasi-Minimal Residual Algorithm (Freund, 1990).**

Transpose-Free Variants

➤ **BCG and QMR require a matrix-by-vector product with A and A^T at each step. The products with A^T do not contribute directly to x_m.** ➤ **They allow to determine the scalars (α_j and β_j in BCG).**

➤ **QUESTION: is it possible to bypass the use of A^T?**

➤ **Motivation: in nonlinear equations, A is often not available explicitly but via the Frechet derivative:**

$$J(u_k)v = \frac{F(u_k + \epsilon v) - F(u_k)}{\epsilon}.$$

Conjugate Gradient Squared

* Clever variant of BCG which avoids using A^T [Sonneveld, 1984].

In BCG:

$$r_i = \rho_i(A)r_0$$

where ρ_i = polynomial of degree i.

In CGS:

$$r_i = \rho_i^2(A)r_0$$

➤ Define :

$$r_j = \phi_j(A)r_0,$$

$$p_j = \pi_j(A)r_0,$$

$$r_j^* = \phi_j(A^T)r_0^*,$$

$$p_j^* = \pi_j(A^T)r_0^*$$

Scalar α_j in BCG is given by

$$\alpha_j = \frac{(\phi_j(A)r_0, \phi_j(A^T)r_0^*)}{(A\pi_j(A)r_0, \pi_j(A^T)r_0^*)} = \frac{(\phi_j^2(A)r_0, r_0^*)}{(A\pi_j^2(A)r_0, r_0^*)}$$

- **Possible to get a recursion for the $\phi_j^2(A)r_0$ and $\pi_j^2(A)r_0$?**

$$\begin{aligned} \phi_{j+1}(t) &= \phi_j(t) - \alpha_j t \pi_j(t), \\ \pi_{j+1}(t) &= \phi_{j+1}(t) + \beta_j \pi_j(t) \end{aligned}$$

- **Square these equalities**

$$\begin{aligned} \phi_{j+1}^2(t) &= \phi_j^2(t) - 2\alpha_j t \pi_j(t)\phi_j(t) + \alpha_j^2 t^2 \pi_j^2(t), \\ \pi_{j+1}^2(t) &= \phi_{j+1}^2(t) + 2\beta_j \phi_{j+1}(t)\pi_j(t) + \beta_j^2 \pi_j(t)^2. \end{aligned}$$

- **Problem: ...**

.. Cross terms

Solution: Let $\phi_{j+1}(t)\pi_j(t)$, be a third member of the recurrence. For $\pi_j(t)\phi_j(t)$, note:

$$\phi_j(t)\pi_j(t) = \phi_j(t)\left(\phi_j(t) + \beta_{j-1}\pi_{j-1}(t)\right)$$

$$= \phi_j^2(t) + \beta_{j-1}\phi_j(t)\pi_{j-1}(t).$$

Result:

$$\begin{aligned} \phi_{j+1}^2 &= \phi_j^2 - \alpha_j t \left(2\phi_j^2 + 2\beta_{j-1}\phi_j\pi_{j-1} - \alpha_j t\, \pi_j^2\right) \\ \phi_{j+1}\pi_j &= \phi_j^2 + \beta_{j-1}\phi_j\pi_{j-1} - \alpha_j t\, \pi_j^2 \\ \pi_{j+1}^2 &= \phi_{j+1}^2 + 2\beta_j\phi_{j+1}\pi_j + \beta_j^2\pi_j^2. \end{aligned}$$

Define:

$$r_j = \phi_j^2(A)r_0, \quad p_j = \pi_j^2(A)r_0, \quad q_j = \phi_{j+1}(A)\pi_j(A)r_0$$

Recurrences become:

$$
\begin{aligned}
r_{j+1} &= r_j - \alpha_j A\,(2r_j + 2\beta_{j-1}q_{j-1} - \alpha_j A\,p_j)\,, \\
q_j &= r_j + \beta_{j-1}q_{j-1} - \alpha_j A\,p_j, \\
p_{j+1} &= r_{j+1} + 2\beta_j q_j + \beta_j^2 p_j.
\end{aligned}
$$

Define auxiliary vector $d_j = 2r_j + 2\beta_{j-1}q_{j-1} - \alpha_j A p_j$

➤ Sequence of operations to compute the approximate solution, starting with $r_0 := b - Ax_0$, $p_0 := r_0$, $q_0 := 0$, $\beta_0 := 0$.

1. $\alpha_j = (r_j, r_0^*)/(Ap_j, r_0^*)$
2. $d_j = 2r_j + 2\beta_{j-1}q_{j-1} - \alpha_j Ap_j$
3. $q_j = r_j + \beta_{j-1}q_{j-1} - \alpha_j Ap_j$
4. $x_{j+1} = x_j + \alpha_j d_j$
5. $r_{j+1} = r_j - \alpha_j Ad_j$
6. $\beta_j = (r_{j+1}, r_0^*)/(r_j, r_0^*)$
7. $p_{j+1} = r_{j+1} + \beta_j(2q_j + \beta_j p_j)$.

- **one more auxiliary vector, $u_j = r_j + \beta_{j-1} q_{j-1}$. So**

$$\begin{aligned} d_j &= u_j + q_j, \\ q_j &= u_j - \alpha_j A p_j, \\ p_{j+1} &= u_{j+1} + \beta_j (q_j + \beta_j p_j), \end{aligned}$$

- **vector d_j is no longer needed.**

ALGORITHM : 7 . Conjugate Gradient Squared

```
1. Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary.
2. Set $p_0 := u_0 := r_0$.
3. For $j = 0, 1, 2 \ldots,$ until convergence Do:
4.     $\alpha_j = (r_j, r_0^*)/(Ap_j, r_0^*)$
5.     $q_j = u_j - \alpha_j Ap_j$
6.     $x_{j+1} = x_j + \alpha_j(u_j + q_j)$
7.     $r_{j+1} = r_j - \alpha_j A(u_j + q_j)$
8.     $\beta_j = (r_{j+1}, r_0^*)/(r_j, r_0^*)$
9.     $u_{j+1} = r_{j+1} + \beta_j q_j$
10.    $p_{j+1} = u_{j+1} + \beta_j(q_j + \beta_j p_j)$
11. EndDo
```

➤ **Note: no matrix-by-vector products with A^T but two matrix-by-vector products with A, at each step.**

Vector:	$\longleftrightarrow$	**Polynomial in BCG :**
q_i	$\longleftrightarrow$	$\bar{r}_i(t)\bar{p}_{i-1}(t)$
u_i	$\longleftrightarrow$	$\bar{p}_i^2(t)$
r_i	$\longleftrightarrow$	$\bar{r}_i^2(t)$

where $\bar{r}_i(t)$ = residual polynomial at step i for BCG, .i.e., $r_i = \bar{r}_i(A)r_0$, and $\bar{p}_i(t)$ = conjugate direction polynomial at step i, i.e., $p_i = \bar{p}_i(A)r_0$.

BCGSTAB (van der Vorst, 1992)

➤ In CGS: residual polynomial of BCG is squared. ➤ bad behavior in case of irregular convergence.

➤ Bi-Conjugate Gradient Stabilized (BCGSTAB) = a variation of CGS which avoids this difficulty. ➤ Derivation similar to CGS.

➤ Residuals in BCGSTAB are of the form,

$$r'_j = \psi_j(A)\phi_j(A)r_0$$

in which, $\phi_j(t)$ = BCG residual polynomial, and ..

➤ .. $\psi_j(t)$ = a new polynomial defined recursively as

$$\psi_{j+1}(t) = (1 - \omega_j t)\psi_j(t)$$

ω_i chosen to ‘smooth’ convergence [steepest descent step]

ALGORITHM : 8 . BCGSTAB

```
1.  Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary;
2.  $p_0 := r_0$.
3.  For $j = 0, 1, \ldots,$ until convergence Do:
4.      $\alpha_j := (r_j, r_0^*)/(Ap_j, r_0^*)$
5.      $s_j := r_j - \alpha_j Ap_j$
6.      $\omega_j := (As_j, s_j)/(As_j, As_j)$
7.      $x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
8.      $r_{j+1} := s_j - \omega_j As_j$
9.      $\beta_j := \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$
10.     $p_{j+1} := r_{j+1} + \beta_j(p_j - \omega_j Ap_j)$
11. EndDo
```

PRECONDITIONING

Preconditioning – Basic principles

Basic idea is to use the Krylov subspace method on a modified system such as

$$\boxed{M^{-1}Ax = M^{-1}b.}$$

- The matrix $M^{-1}A$ need not be formed explicitly; only need to solve $Mw = v$ whenever needed.
- Consequence: fundamental requirement is that it should be easy to compute $M^{-1}v$ for an arbitrary vector v.

Left, Right, and Split preconditioning

Left preconditioning: $M^{-1}Ax = M^{-1}b$

Right preconditioning: $AM^{-1}u = b$, **with** $x = M^{-1}u$

Split preconditioning: $M_L^{-1}AM_R^{-1}u = M_L^{-1}b$, **with** $x = M_R^{-1}u$

[Assume M is factored: $M = M_L M_R$.]

Preconditioned CG (PCG)

- Assume: A and M are both SPD.
- Applying CG directly to $M^{-1}Ax = M^{-1}b$ or $AM^{-1}u = b$ won't work because coefficient matrices are not symmetric.
- Alternative: when $M = LL^T$ use split preconditioner option
- Second alternative: Observe that $M^{-1}A$ is self-adjoint wrt M inner product:

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M^{-1}Ay)_M$$

Preconditioned CG (PCG)

ALGORITHM : 9. *Preconditioned Conjugate Gradient*

```
1. Compute $r_0 := b - Ax_0$, $z_0 = M^{-1}r_0$, and $p_0 := z_0$
2. For $j = 0, 1, \ldots$, until convergence Do:
3.     $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4.     $x_{j+1} := x_j + \alpha_j p_j$
5.     $r_{j+1} := r_j - \alpha_j Ap_j$
6.     $z_{j+1} := M^{-1}r_{j+1}$
7.     $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8.     $p_{j+1} := z_{j+1} + \beta_j p_j$
9. EndDo
```

Note $M^{-1}A$ is also self-adjoint with respect to $(.,.)_A$:

$$(M^{-1}Ax, y)_A = (AM^{-1}Ax, y) = (x, AM^{-1}Ay) = (x, M^{-1}Ay)_A$$

- Can obtain a similar algorithm
- Assume that M = Cholesky product $M = LL^T$.

Then, another possibility: Split preconditioning option, which applies CG to the system

$$L^{-1}AL^{-T}u = L^{-1}b, \text{ with } x = L^T u$$

- Notation: $\hat{A} = L^{-1}AL^{-T}$. All quantities related to the preconditioned system are indicated by $\hat{}$.

ALGORITHM : 10 . *CG with Split Preconditioner*

```
1. Compute $r_0 := b - Ax_0$; $\hat{r}_0 = L^{-1}r_0$; and $p_0 := L^{-T}\hat{r}_0$.
2. For $j = 0, 1, \ldots,$ until convergence Do:
3.     $\alpha_j := (\hat{r}_j, \hat{r}_j)/(Ap_j, p_j)$
4.     $x_{j+1} := x_j + \alpha_j p_j$
5.     $\hat{r}_{j+1} := \hat{r}_j - \alpha_j L^{-1}Ap_j$
6.     $\beta_j := (\hat{r}_{j+1}, \hat{r}_{j+1})/(\hat{r}_j, \hat{r}_j)$
7.     $p_{j+1} := L^{-T}\hat{r}_{j+1} + \beta_j p_j$
8. EndDo
```

➤ **The x_j's produced by the above algorithm and PCG are identical (if same initial guess is used).**

Flexible accelerators

Question: What can we do in case M is defined only approximately? i.e., if it can vary from one step to the other.?

Applications:

➤ Iterative techniques as preconditioners: Block-SOR, SSOR, Multi-grid, etc..

➤ Chaotic relaxation type preconditioners (e.g., in a parallel computing environment)

➤ Mixing Preconditioners – mixing coarse mesh / fine mesh preconditioners.

ALGORITHM : 11 . *GMRES – No preconditioning*

1. Start: *Choose* x_0 *and a dimension* m *of the Krylov subspaces.*

2. Arnoldi process:

- *Compute* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$ *and* $v_1 = r_0/\beta$.
- *For* $j = 1, ..., m$ *do*
  - *Compute* $w := Av_j$
  - *for* $i = 1, \ldots, j$, *do* $\left\{ \begin{array}{l} h_{i,j} := (w, v_i) \\ w := w - h_{i,j}v_i \end{array} \right\}$;
  - $h_{j+1,1} = \|w\|_2$; $v_{j+1} = \frac{w}{h_{j+1,1}}$
- *Define* $V_m := [v_1, ..., v_m]$ *and* $\bar{H}_m = \{h_{i,j}\}$.

3. Form the approximate solution: *Compute* $\boxed{x_m = x_0 + V_m y_m}$ *where* $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ *and* $e_1 = [1, 0, \ldots, 0]^T$.

4. Restart: *If satisfied stop, else set* $x_0 \leftarrow x_m$ *and goto 2.*

ALGORITHM : 12 ▪ *GMRES – (right) Preconditioning*

1. Start: *Choose* x_0 *and a dimension* m

2. Arnoldi process:

- *Compute* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$ *and* $v_1 = r_0/\beta$.
- *For* $j = 1, ..., m$ *do*
  - *Compute* $z_j := M^{-1}v_j$
  - *Compute* $w := Az_j$
  - *for* $i = 1, \ldots, j$, *do* : $\left\{ \begin{array}{l} h_{i,j} := (w, v_i) \\ w := w - h_{i,j}v_i \end{array} \right\}$
  - $h_{j+1,1} = \|w\|_2$; $v_{j+1} = w/h_{j+1,1}$
- *Define* $V_m := [v_1, ..., v_m]$ *and* $\bar{H}_m = \{h_{i,j}\}$.

3. Form the approximate solution: $\boxed{x_m = x_0 + M^{-1}V_m y_m}$ *where* $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ *and* $e_1 = [1, 0, \ldots, 0]^T$.

4. Restart: *If satisfied stop, else set* $x_0 \leftarrow x_m$ *and goto 2.*

ALGORITHM : 13 ▪ *GMRES – variable preconditioner*

```
1. Start: Choose $x_0$ and a dimension $m$ of the Krylov subspaces.
2. Arnoldi process:
    • Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$ and $v_1 = r_0/\beta$.
    • For $j = 1, ..., m$ do
        – Compute $z_j := M_j^{-1} v_j$ ; Compute $w := Az_j$;
        – for $i = 1, \ldots, j$, do: $\left\{ \begin{array}{l} h_{i,j} := (w, v_i) \\ w := w - h_{i,j} v_i \end{array} \right\}$;
        – $h_{j+1,1} = \|w\|_2$; $v_{j+1} = w/h_{j+1,1}$
    • Define $Z_m := [z_1, ...., z_m]$ and $\bar{H}_m = \{h_{i,j}\}$.
3. Form the approximate solution: Compute $\boxed{x_m = x_0 + Z_m y_m}$ where
   $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ and $e_1 = [1, 0, \ldots, 0]^T$.
4. Restart: If satisfied stop, else set $x_0 \leftarrow x_m$ and goto 2.
```

Properties

- x_m minimizes $b - Ax_m$ over $\text{Span}\{Z_m\}$.
- If $Az_j = v_j$ (i.e., if preconditioning is 'exact' at step j) then approximation x_j is exact.
- If M_j is constant then method is $\equiv$ to Right-Preconditioned GMRES.

Additional Costs:

- Arithmetic: none.
- Memory: Must save the additional set of vectors $\{z_j\}_{j=1,\ldots m}$

Advantage: Flexibility

Standard preconditioners

- Simplest preconditioner: M = Diag(A) ➤ poor convergence.
- Next to simplest: SSOR $M = (D - \omega E)D^{-1}(D - \omega F)$
- Still simple but often more efficient: ILU(0).
- ILU(p) – ILU with level of fill p – more complex.
- Class of ILU preconditioners with threshold
- Class of approximate inverse preconditioners
- Class of Multilevel ILU preconditioners: Multigrid, Algebraic Multi-grid, M-level ILU, ..

An observation. Introduction to Preconditioning

➤ Take a look back at basic relaxation methods: Jacobi, Gauss-Seidel, SOR, SSOR, ...

➤ These are iterations of the form $x^{(k+1)} = Mx^{(k)} + f$ where M is of the form $M = I - P^{-1}A$. For example for SSOR,

$$P_{SSOR} = (D - \omega E)D^{-1}(D - \omega F)$$

- **SSOR attempts to solve the equivalent system**

$$P^{-1}Ax = P^{-1}b$$

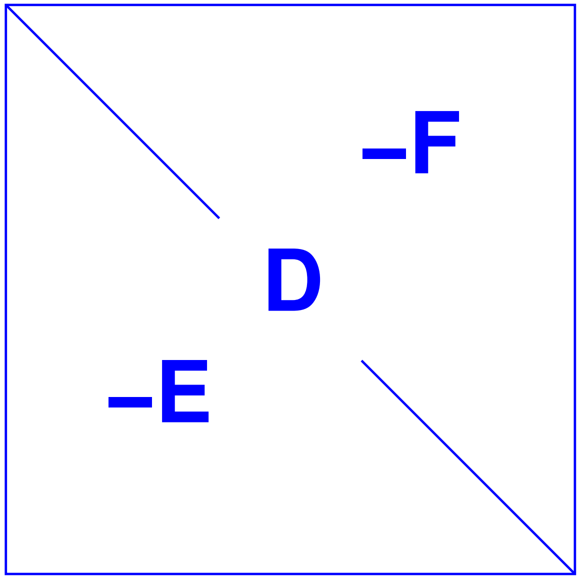
where $P \equiv P_{SSOR}$ by the fixed point iteration

$$x^{(k+1)} = \underbrace{(I - P^{-1}A)}_{M} x^{(k)} + P^{-1}b \quad \textbf{instead of} \quad x^{(k+1)} = (I - A)x^{(k)} + b$$

<u>In other words:</u>

Relaxation Scheme $\Longleftrightarrow$ **Preconditioned Fixed Point Iteration**

The SOR/SSOR preconditioner



- **SOR preconditioning**

$$M_{SOR} = (D - \omega E)$$

- **SSOR preconditioning**

$$M_{SSOR} = (D - \omega E)D^{-1}(D - \omega F)$$

- $M_{SSOR} = LU$, $L =$ **lower unit matrix,** $U =$ **upper triangular. One solve with** $M_{SSOR} \approx$ **same cost as a MAT-VEC.**

- k-step SOR (resp. SSOR) preconditioning:

$$\boxed{k \text{ steps of SOR (resp. SSOR)}}$$

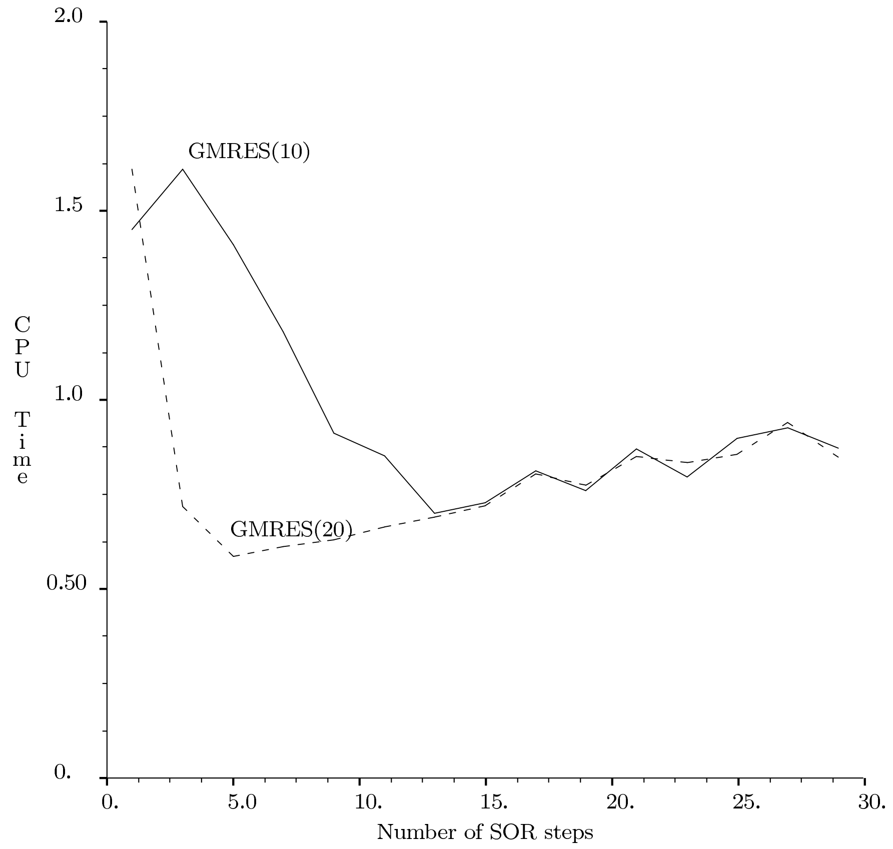
- Questions: Best ω? For preconditioning can take $\omega = 1$

$$\boxed{M = (D - E)D^{-1}(D - F)}$$

Observe: $M = LU + R$ with $R = ED^{-1}F$.

- Best k? $k = 1$ is rarely the best. Substantial difference in performance.

Iteration times versus k for SOR(k) preconditioned GMRES



ILU(0) and IC(0) preconditioners

- **Notation:** $NZ(X) = \{(i,j) \mid X_{i,j} \neq 0\}$
- **Formal definition of ILU(0):**

$$\begin{array}{l} A = LU + R \\ NZ(L) \bigcup NZ(U) = NZ(A) \\ r_{ij} = 0 \text{ for } (i,j) \in NZ(A) \end{array}$$

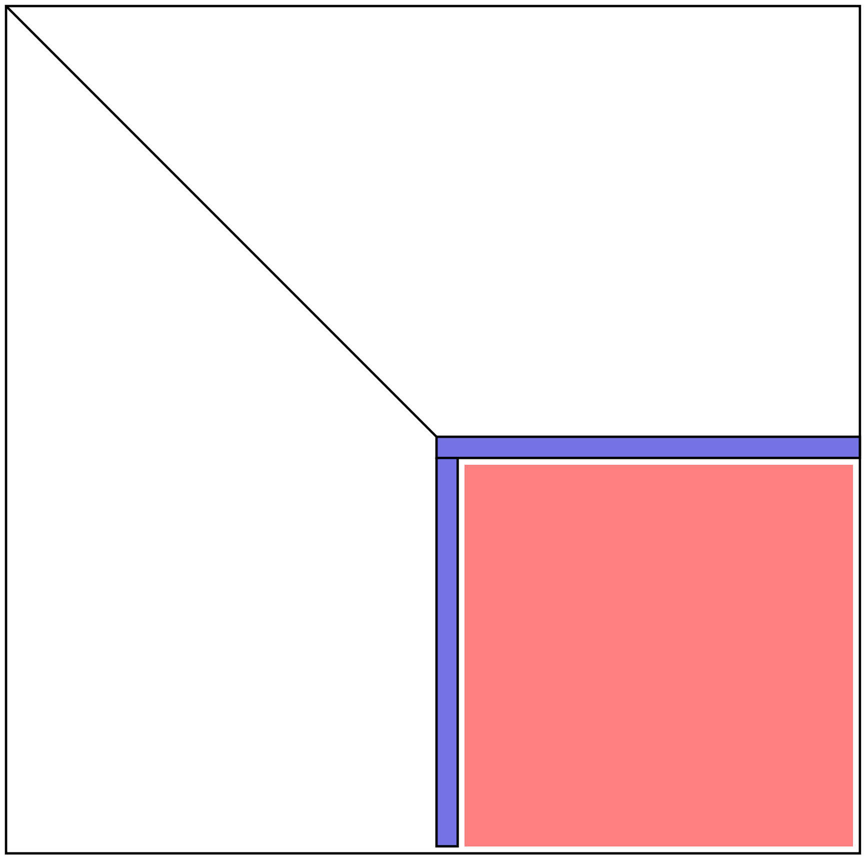
- **This does not define $ILU(0)$ in a unique way.**

Constructive definition: Compute the LU factorization of A but drop any fill-in in L and U outside of Struct(A).

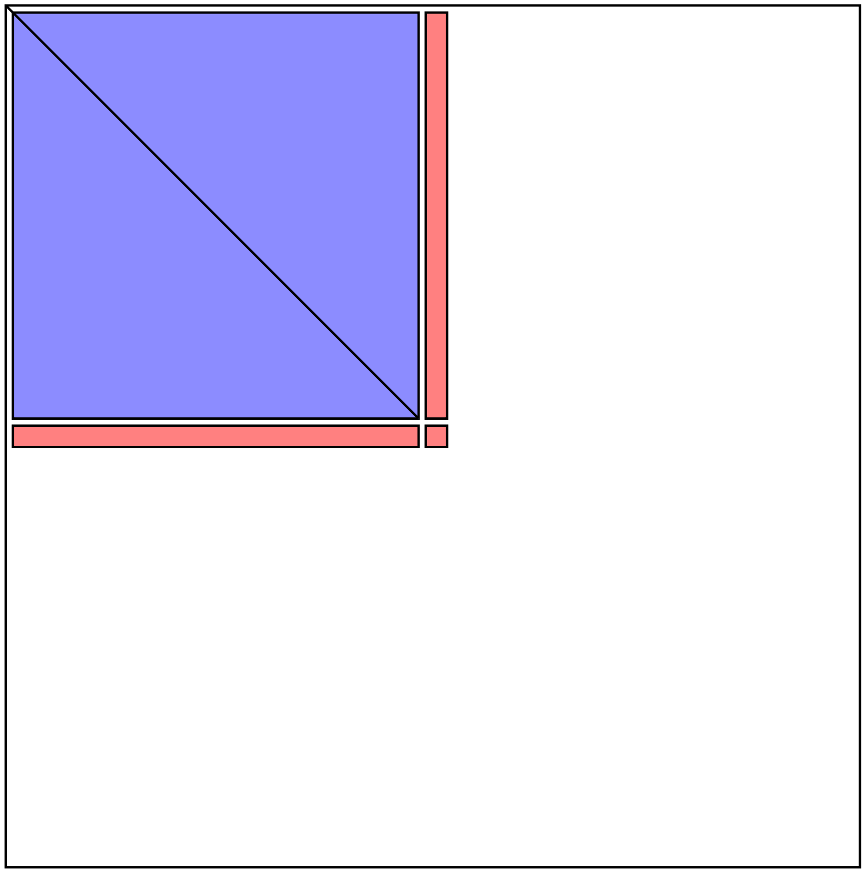
- **ILU factorizations are often based on i, k, j version of GE.**

What is the IKJ version of GE?

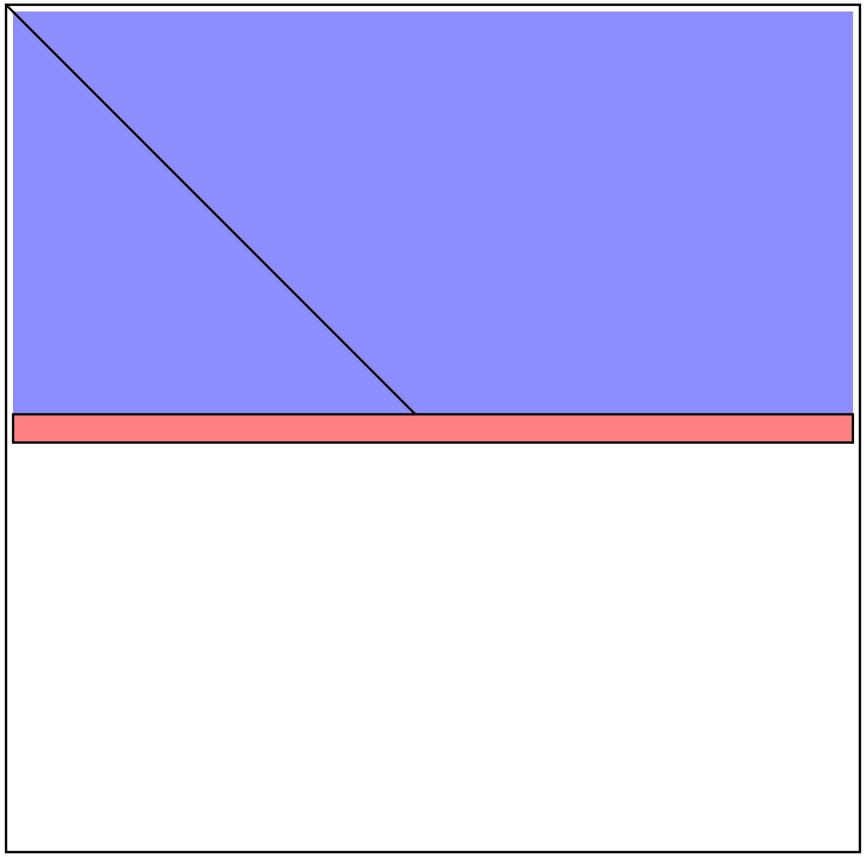
Different computational patterns for gaussian elimination



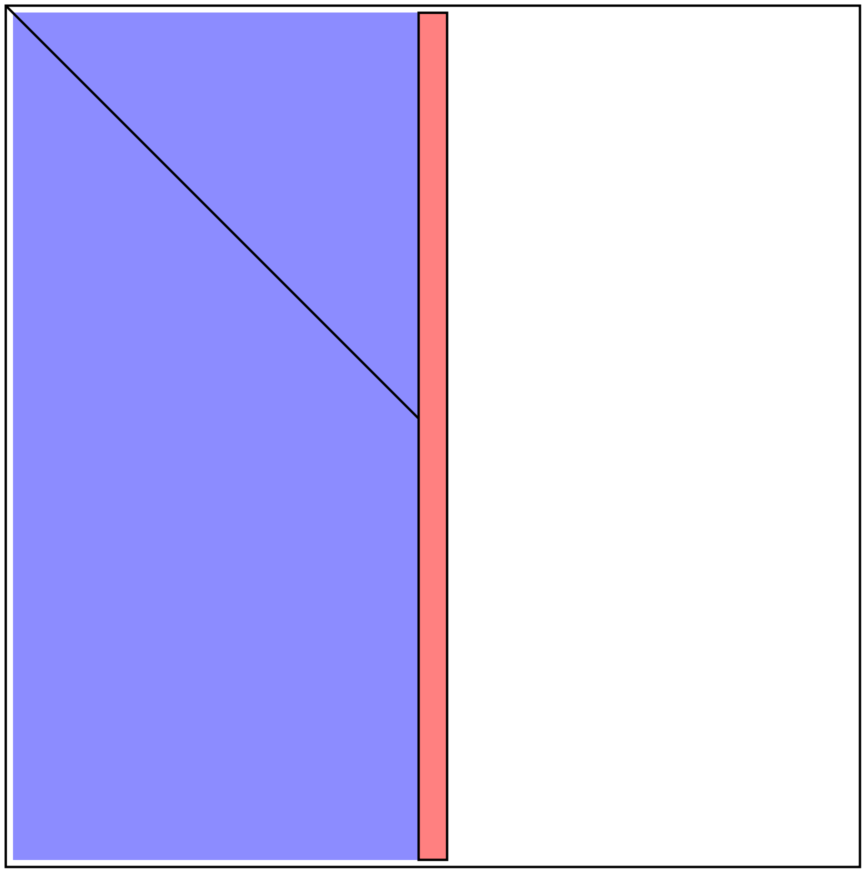
KJI,KJI



IJK



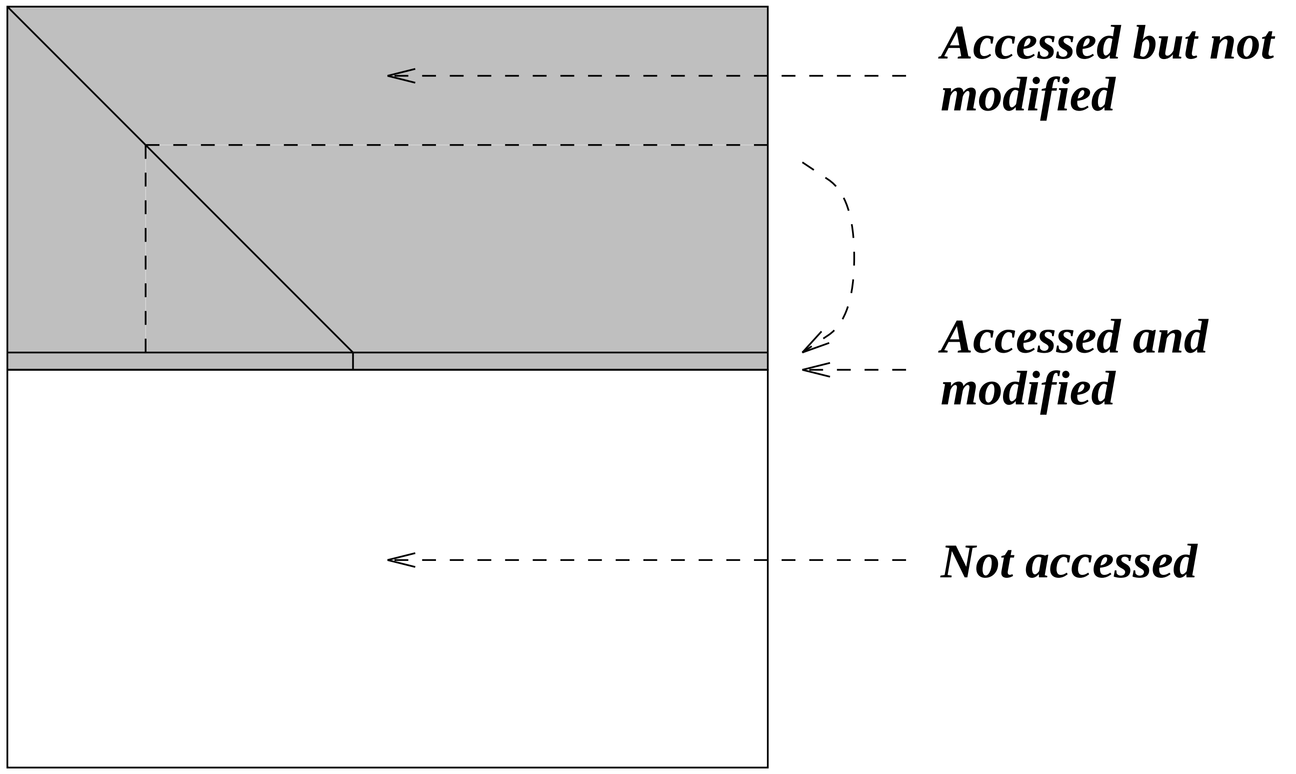
IKJ



JKI

ALGORITHM : 14 . Gaussian Elimination – IKJ Variant

```
1. For $i = 2, \ldots, n$ Do:
2.     For $k = 1, \ldots, i-1$ Do:
3.         $a_{ik} := a_{ik}/a_{kk}$
4.         For $j = k+1, \ldots, n$ Do:
5.             $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
6.         EndDo
7.     EndDo
8. EndDo
```


Accessed but not modified
Accessed and modified
Not accessed

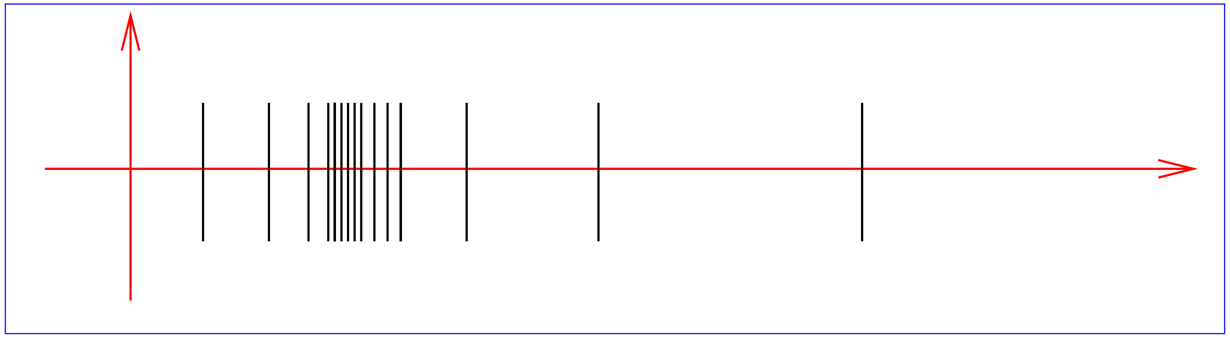
ILU(0) – zero-fill ILU

ALGORITHM : 15 . *ILU(0)*

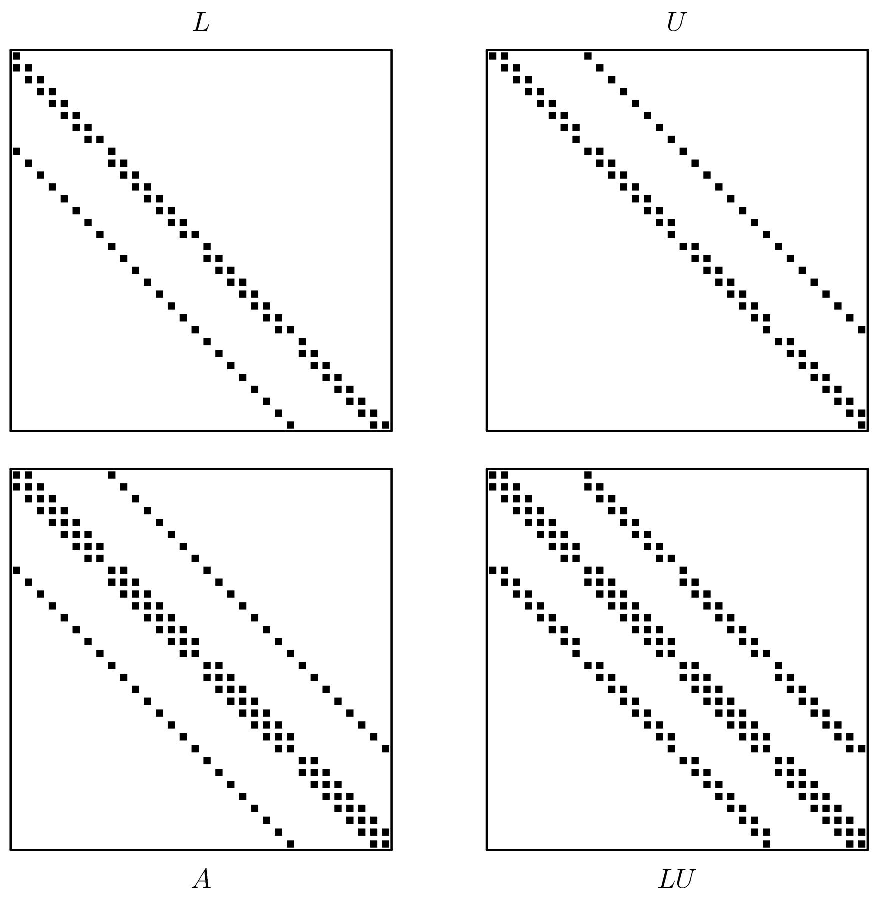
```
For $i = 1, \ldots, N$ Do:
    For $k = 1, \ldots, i-1$ and if $(i,k) \in NZ(A)$ Do:
        Compute $a_{ik} := a_{ik}/a_{kj}$
        For $j = k+1, \ldots$ and if $(i,j) \in NZ(A)$, Do:
        compute $a_{ij} := a_{ij} - a_{ik}a_{k,j}$.
    EndFor
EndFor
```

➤ When A is SPD then the ILU factorization = Incomplete Cholesky factorization – IC(0). Meijerink and Van der Vorst [1977].

Typical eigenvalue distribution of preconditioned matrix

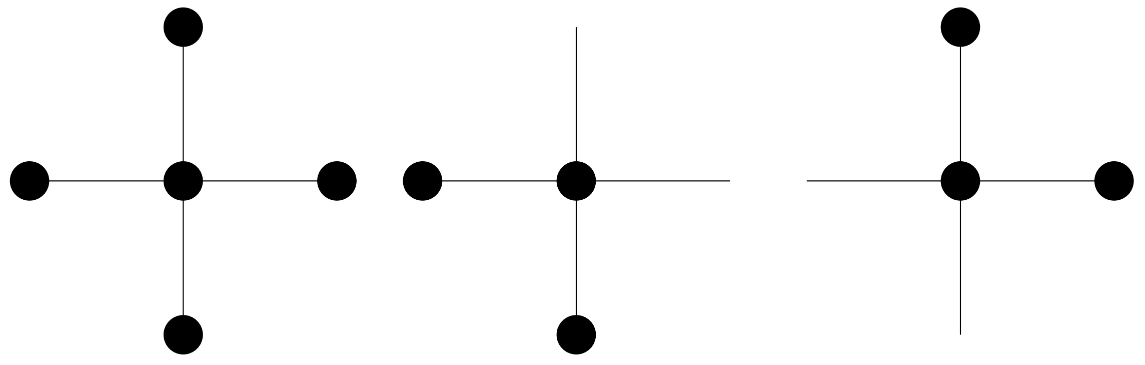


Pattern of ILU(0) for 5-point matrix

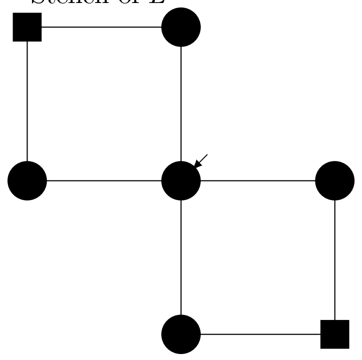


Stencils and ILU factorization

Stencils of A and the L and U parts of A:



Stencil of A Stencil of L Stencil of U



■ Fill-ins

Higher order ILU factorization

- Higher accuracy incomplete Cholesky: for regularly structured problems, IC(p) allows p additional diagonals in L.
- Can be generalized to irregular sparse matrices using the notion of level of fill-in [Watts III, 1979]

- Initially $Lev_{ij} = \begin{cases} 0 & \text{for } a_{ij} \neq 0 \\ \infty & \text{for } a_{ij} == 0 \end{cases}$
- At a given step i of Gaussian elimination:

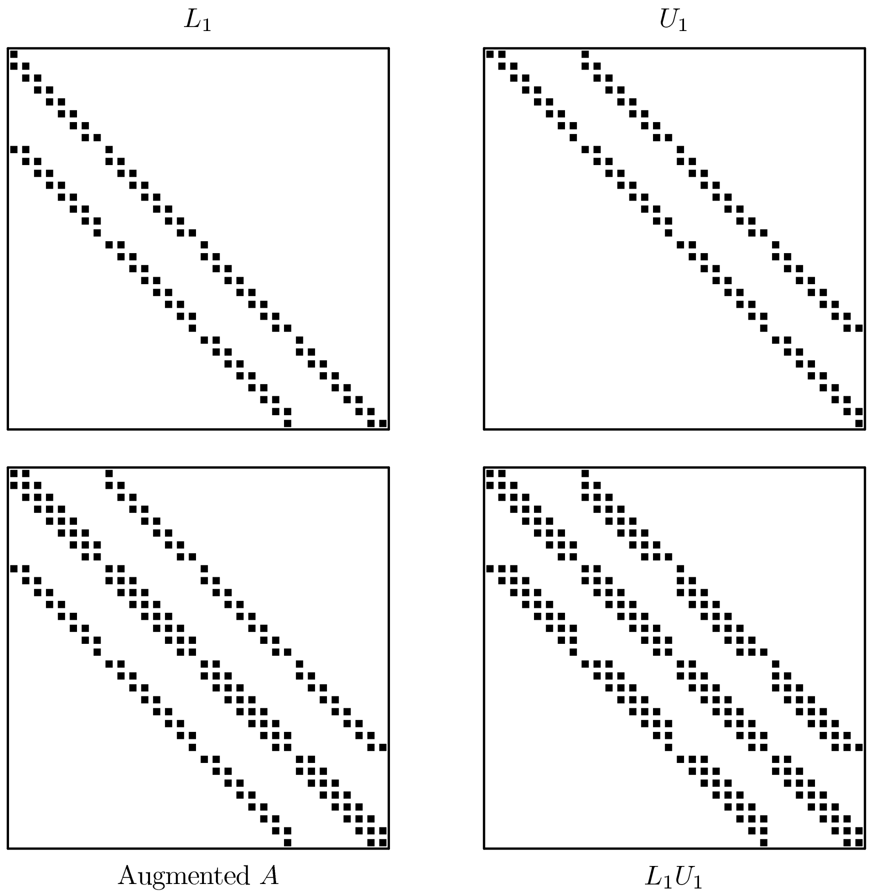
$$\boxed{Lev_{kj} = \min\{Lev_{kj}; Lev_{ki} + Lev_{ij} + 1\}}$$

➤ ILU(p) Strategy = drop anything with level of fill-in exceeding p.

* Increasing level of fill-in usually results in more accurate ILU and...

* ...typically in fewer steps and fewer arithmetic operations.

ILU(1)



ALGORITHM : 16. *ILU(p)*

```
For $i = 2, N$ Do
    For each $k = 1, \ldots, i-1$ and if $a_{ij} \neq 0$ do
        Compute $a_{ik} := a_{ik}/a_{jj}$
        Compute $a_{i,*} := a_{i,*} - a_{ik}a_{k,*}$.
        Update the levels of $a_{i,*}$
        Replace any element in row $i$ with $lev(a_{ij}) > p$ by zero.
    EndFor
EndFor
```

➤ The algorithm can be split into a symbolic and a numerical phase. Level-of-fill ➤ in Symbolic phase

ILU with threshold – generic algorithms

ILU(p) factorizations are based on structure only and not numerical values ➤ potential problems for non M-matrices.

➤ One remedy: ILU with threshold – (generic name ILUT.)

Two broad approaches:

First approach [derived from direct solvers]: use any (direct) sparse solver and incorporate a dropping strategy. [Munksgaard (?), Osterby & Zlatev, Sameh & Zlatev[90], D. Young, & al. (Boeing) etc...]

Second approach : [derived from ‘iterative solvers’ viewpoint]

1. use a (row or colum) version of the (i, k, j) version of GE;
2. apply a drop strategy for the elment l_{ik} as it is computed;
3. perform the linear combinations to get a_{i*}. Use full row expansion of a_{i*};
4. apply a drop strategy to fill-ins.

ILU with threshold: ILUT(k, ϵ)

- Do the i, k, j version of Gaussian Elimination (GE).
- During each i-th step in GE, discard any pivot or fill-in whose value is below $\epsilon \|row_i(A)\|$.
- Once the i-th row of $L + U$, (L-part + U-part) is computed retain only the k largest elements in both parts.

➤ Advantages: controlled fill-in. Smaller memory overhead.

➤ Easy to implement –

➤ Can be made quite inexpensive.