Iterative Methods in Linear Algebra

Azzam Haidar

Innovative Computing Laboratory
Computer Science Department
The University of Tennessee

Wednesday April 18, 2018
Topics

Projection in Scientific Computing

Sparse matrices, parallel implementations

PDEs, Numerical solution, Tools, etc.

Iterative Methods
Part I
Discussion
Motivation for iterative methods

Part II
Stationary Iterative Methods

Part III
Iterative Refinement Methods

Part IV
Nonstationary Iterative Methods
Part I
Discussion
Discussion

Original matrix:

```matlab
>> load matrix.output
>> S = spconvert(matrix);
>> spy(S)
```

Reordered matrix:

```matlab
>> load A.data
>> S = spconvert(A);
>> spy(S)
```
Discussion

Original sub-matrix:
>> load matrix.output
>> S = spconvert(matrix);
>> spy(S(1:8660),1:8660))

Reordered sub-matrix:
>> load A.data
>> S = spconvert(A);
>> spy(S(8602:8700,8602:8700))
Discussion

Results on torc0:
Pentium III 933 MHz, 16 KB L1 and 256 KB L2 cache

- Original matrix: Mflop/s = 42.4568
- Reordered matrix: Mflop/s = 45.3954
- BCRS Mflop/s = 72.17374

Results on woodstock:
Intel Xeon 5160 Woodcrest 3.0GHz, L1 32 KB, L2 4 MB

- Original matrix: Mflop/s = 386.8
- Reordered matrix: Mflop/s = 387.6
- BCRS Mflop/s = 894.2
Iterative Methods
Motivation

So far we have discussed and have seen:

- Many engineering and physics simulations lead to sparse matrices
  - e.g. PDE based simulations and their discretizations based on
    - FEM/FVM
    - finite differences
    - Petrov-Galerkin type of conditions, etc.

- How to optimize performance on sparse matrix computations

- Some software how to solve sparse matrix problems (e.g. PETSc)

The question is:

**Can we solve sparse matrix problems faster than using direct sparse methods?**

- In certain cases **Yes:** using **iterative methods**
Sparse direct methods

- These are based on Gaussian elimination (LU)
- Performed in sparse format

Are there limitations of the approach?

- Yes, they have fill-ins which lead to
  - more memory requirements
  - more flops being performed
- Fill-ins can become prohibitively high
Consider LU for the matrix below
- a nonzero is represented by a *

1st step of LU factorization will introduce fill-ins
- marked by an F
Sparse direct methods

Fill-ins can be improved by reordering

- Remember: we talked about it in slightly different context (for speed and parallelization)
- Consider the following reordering

These were extreme cases

- but still, the problem exists
Sparse methods

Sparse direct vs dense methods

- Dense takes $O(n^2)$ storage, $O(n^3)$ flops, runs within peak performance
- Sparse direct can take $O(\#\text{nonz})$ storage, and $O(\#\text{nonz})$ flops, but these can also grow due to fill-ins and performance is bad
- with $\#\text{nonz} << n^2$ and 'proper' ordering it pays off to do sparse direct

Software (table from Julien Langou)

- [http://www.netlib.org/utk/people/JackDongarra/la-sw.html](http://www.netlib.org/utk/people/JackDongarra/la-sw.html)

<table>
<thead>
<tr>
<th>Package</th>
<th>Support</th>
<th>Type</th>
<th>Language</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>Complex</td>
<td>f77</td>
<td>c</td>
</tr>
<tr>
<td>DSCPACK</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HSL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MFACT</td>
<td>yes</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MUMPS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PSPASES</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SPARSE</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SPOOLES</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SuperLU</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TAUCS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UMFPACK</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Y12M</td>
<td>?</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sparse methods

What about Iterative Methods? Think for example

\[ x_{i+1} = x_i + P(b - Ax_i) \]

Pluses:

- Storage is only \( O(\#\text{nonz}) \) (for the matrix and a few working vectors)
- Can take only a few iterations to converge (e.g. \(< < n\))
  - for \( P = A^{-1} \) it takes 1 iteration (check)!
- In general easy to parallelize
Sparse methods

What about **Iterative Methods**? Think for example

\[ x_{i+1} = x_i + P(b - Ax_i) \]

Minuses:

- Performance can be bad
  (as we saw previously)
- Convergence can be slow or even stagnate
  But can be improved with **preconditioning**
  - Think of \( P \) as a preconditioner, an operator/matrix \( P \approx A^{-1} \)
  - Optimal preconditioners (e.g. multigrid can be) lead to convergence in \( O(1) \) iterations
Part II

Stationary Iterative Methods
Stationary Iterative Methods

Can be expressed in the form

\[ x_{i+1} = Bx_i + c \]

where \( B \) and \( c \) do not depend on \( i \)

- older, simpler, easy to implement, but usually not as effective (as nonstationary)
- examples: Richardson, Jacobi, Gauss-Seidel, SOR, etc. (next)
Richardson Method

Richardson iteration

\[ x_{i+1} = x_i + (b - Ax_i) = (I - A)x_i + b \]  \hspace{1cm} (1)

i.e. \( B \) from the definition above is \( B = I - A \)

Denote \( e_i = x - x_i \) and rewrite (1)

\[
\begin{align*}
    x - x_{i+1} &= x - x_i - (Ax - Ax_i) \\
    e_{i+1} &= e_i - Ae_i \\
    &= (I - A)e_i \\
    ||e_{i+1}|| &\leq ||(I - A)e_i|| \leq ||I - A|| ||e_i|| \leq ||I - A||^2 ||e_{i-1}|| \\
    &\leq \cdots \leq ||I - A||^i ||e_0||
\end{align*}
\]

i.e. for convergence \( (e_i \to 0) \) we need

\[ ||I - A|| < 1 \]

for some norm \( || \cdot || \), e.g. when \( \rho(B) < 1 \).
Jacobi Method

\[ x_{i+1} = x_i + D^{-1}(b - Ax_i) = (I - D^{-1}A)x_i + D^{-1}b \]  \tag{2}

where \( D \) is the diagonal of \( A \) (assumed nonzero; \( B = I - D^{-1}A \) from the definition)

Denote \( e_i = x - x_i \) and rewrite (2)

\[
\begin{align*}
  x - x_{i+1} &= x - x_i - D^{-1}(Ax - Ax_i) \\
  e_{i+1} &= e_i - D^{-1}Ae_i \\
          &= (I - D^{-1}A)e_i
\end{align*}
\]

\[
\|e_{i+1}\| \leq \|(I - D^{-1}A)e_i\| \leq \|I - D^{-1}A\| \|e_i\| \leq \|I - D^{-1}A\|^2 \|e_{i-1}\| \\
\leq \cdots \leq \|I - D^{-1}A\|^\i \|e_0\|
\]

i.e. for convergence (\( e_i \to 0 \)) we need

\[
\|I - D^{-1}A\| < 1
\]

for some norm \( \| \cdot \| \), e.g. when \( \rho(I - D^{-1}A) < 1 \)
Gauss-Seidel Method

\[ x_{i+1} = (D - L)^{-1}(Ux_i + b) \]  \hspace{1cm} (3)

where \(-L\) is the lower and \(-U\) the upper triangular part of \(A\), and \(D\) is the diagonal.

Equivalently:

\[
\begin{align*}
    x_1^{(i+1)} &= (b_1 - a_{12}x_2^{(i)} - a_{13}x_3^{(i)} - a_{14}x_4^{(i)} - \ldots - a_{1n}x_n^{(i)}) / a_{11} \\
    x_2^{(i+1)} &= (b_2 - a_{21}x_1^{(i+1)} - a_{23}x_3^{(i)} - a_{24}x_4^{(i)} - \ldots - a_{2n}x_n^{(i)}) / a_{22} \\
    x_3^{(i+1)} &= (b_2 - a_{31}x_1^{(i+1)} - a_{32}x_2^{(i+1)} - a_{34}x_4^{(i)} - \ldots - a_{3n}x_n^{(i)}) / a_{22} \\
    &\vdots \\
    x_n^{(i+1)} &= (b_n - a_{n1}x_1^{(i+1)} - a_{n2}x_2^{(i+1)} - a_{n3}x_3^{(i+1)} - \ldots - a_{n,n-1}x_{n-1}^{(i+1)}) / a_{nn}
\end{align*}
\]
A comparison (from Julien Langou slides)

Gauss-Seidel method

\[ A = \begin{pmatrix}
10 & 1 & 2 & 0 & 1 \\
0 & 12 & 1 & 3 & 1 \\
1 & 2 & 9 & 1 & 0 \\
0 & 3 & 1 & 10 & 0 \\
1 & 2 & 0 & 0 & 15
\end{pmatrix} \quad b = \begin{pmatrix}
23 \\
44 \\
36 \\
49 \\
80
\end{pmatrix} \]

\[ \|b - Ax^{(i)}\|_2 / \|b\|_2, \text{ function of } i, \text{ the number of iterations} \]

<table>
<thead>
<tr>
<th>$x^{(0)}$</th>
<th>$x^{(1)}$</th>
<th>$x^{(2)}$</th>
<th>$x^{(3)}$</th>
<th>$x^{(4)}$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>2.3000</td>
<td>0.8783</td>
<td>0.9790</td>
<td>0.9978</td>
<td>1.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>3.6667</td>
<td>2.1548</td>
<td>2.0107</td>
<td>2.0005</td>
<td>2.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>2.9296</td>
<td>3.0339</td>
<td>3.0055</td>
<td>3.0006</td>
<td>3.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>3.5070</td>
<td>3.9502</td>
<td>3.9962</td>
<td>3.9998</td>
<td>4.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>4.6911</td>
<td>4.9875</td>
<td>5.0000</td>
<td>5.0001</td>
<td>5.0000</td>
</tr>
</tbody>
</table>
A comparison (from Julien Langou slides)

Jacobi Method:

\[
\begin{align*}
    x^{(i+1)}_1 &= (b_1 - a_{12}x_2^{(i)} - a_{13}x_3^{(i)} - a_{14}x_4^{(i)} - \ldots - a_{1n}x_n^{(i)})/a_{11} \\
    x^{(i+1)}_2 &= (b_2 - a_{21}x_1^{(i)} - a_{23}x_3^{(i)} - a_{24}x_4^{(i)} - \ldots - a_{2n}x_n^{(i)})/a_{22} \\
    x^{(i+1)}_3 &= (b_3 - a_{31}x_1^{(i)} - a_{32}x_2^{(i)} - a_{34}x_4^{(i)} - \ldots - a_{3n}x_n^{(i)})/a_{22} \\
    &\vdots \\
    x^{(i+1)}_n &= (b_n - a_{n1}x_1^{(i)} - a_{n2}x_2^{(i)} - a_{n3}x_3^{(i)} - \ldots - a_{n,n-1}x_{n-1}^{(i)})/a_{nn}
\end{align*}
\]

Gauss-Seidel method

\[
\begin{align*}
    x^{(i+1)}_1 &= (b_1 - a_{12}x_2^{(i)} - a_{13}x_3^{(i)} - a_{14}x_4^{(i)} - \ldots - a_{1n}x_n^{(i)})/a_{11} \\
    x^{(i+1)}_2 &= (b_2 - a_{21}x_1^{(i+1)} - a_{23}x_3^{(i)} - a_{24}x_4^{(i)} - \ldots - a_{2n}x_n^{(i)})/a_{22} \\
    x^{(i+1)}_3 &= (b_3 - a_{31}x_1^{(i+1)} - a_{32}x_2^{(i+1)} - a_{34}x_4^{(i)} - \ldots - a_{3n}x_n^{(i)})/a_{22} \\
    &\vdots \\
    x^{(i+1)}_n &= (b_n - a_{n1}x_1^{(i+1)} - a_{n2}x_2^{(i+1)} - a_{n3}x_3^{(i+1)} - \ldots - a_{n,n-1}x_{n-1}^{(i+1)})/a_{nn}
\end{align*}
\]

Gauss-Seidel is the method with better numerical properties (less iterations to convergence). Which is the method with better efficiency in term of implementation in sequential or parallel computer? Why?

In Gauss-Seidel, the computation of \(x_{k+1}^{(i+1)}\) implies the knowledge of \(x_k^{(i+1)}\). Parallelization is impossible.
Convergence can be very slow
Consider a modified **Richardson method**:

\[ x_{i+1} = x_i + \tau (b - Ax_i) \]  \hspace{1cm} (4)

Convergence is **linear**, similarly to Richardson we get

\[ ||e_{i+1}|| \leq ||I - \tau A||^i ||e_0|| \]

but can be very slow (if \( ||I - \tau A|| \) is close to 1), e.g. let

- \( A \) be symmetric and positive definite (SPD)
- the matrix norm in \( ||I - \tau A|| \) is induced by \( || \cdot ||_2 \)

Then the best choice for \( \tau \) (\( \tau = \frac{2}{\lambda_1 + \lambda_N} \)) would give

\[ ||I - \tau A|| = \frac{k(A) - 1}{k(A) + 1} \]

where \( k(A) = \frac{\lambda_N}{\lambda_1} \) is the condition number of \( A \).
Convergence

Note:

- The rate of convergence depend on the condition number $k(A)$
- Even for the best $\tau$ the rate of convergence

$$\frac{k(A) - 1}{k(A) + 1}$$

is slow (i.e. close to 1) for large $k(A)$

- We will see convergence of nonstationary methods also depend on $k(A)$ but is better, e.g. compare with CG

$$\frac{\sqrt{k(A)} - 1}{\sqrt{k(A)} + 1}$$
Part III

Iterative Refinement Methods
Mixed-Precision Solvers
Dense Linear Algebra (DLA) is needed in a wide variety of science and engineering applications:

- **Linear systems:** Solve \( Ax = b \)
  - Computational electromagnetics, material science, applications using boundary integral equations, airflow past wings, fluid flow around ship and other offshore constructions, and many more
- **Least squares:** Find \( x \) to minimize \( || Ax - b || \)
  - Computational statistics (e.g., linear least squares or ordinary least squares), econometrics, control theory, signal processing, curve fitting, and many more
- **Eigenproblems:** Solve \( Ax = \lambda x \)
  - Computational chemistry, quantum mechanics, material science, face recognition, PCA, data-mining, marketing, Google Page Rank, spectral clustering, vibrational analysis, compression, and many more
- **SVD:** \( A = U \Sigma V^* \) (\( Au = \sigma v \) and \( A^*v = \sigma u \))
  - Information retrieval, web search, signal processing, big data analytics, low rank matrix approximation, total least squares minimization, pseudo-inverse, and many more
- **Many variations depending on structure of \( A \)**
  - \( A \) can be symmetric, positive definite, tridiagonal, Hessenberg, banded, sparse with dense blocks, etc.
- **DLA is crucial to the development of sparse solvers**

Provided in MAGMA 2.3

http://icl.cs.utk.edu/magma
https://bitbucket.org/icl/magma
Leveraging Half Precision in HPC on V100

solving linear system $Ax = b$

**LU factorization**

- LU factorization is used to solve a linear system $Ax=b$

\[
\begin{align*}
A \times x &= b \\
LUx &= b \\
Ly &= b \\
\text{then} \quad Ux &= y
\end{align*}
\]
Leveraging Half Precision in HPC on V100 solving linear system $Ax = b$

For $s = 0, nb, .. N$

1. panel factorize
2. update trailing matrix

LU factorization requires $O(n^3)$
most of the operations are spent in GEMM
Leveraging Half Precision in HPC on V100

Motivation

Study of the Matrix Matrix multiplication kernel on Nvidia V100

- `dgemm` achieve about 6.4 Tflop/s
- `sgemm` achieve about 14 Tflop/s
- `hgemm` achieve about 27 Tflop/s
- Tensor cores `gemm` reach about 85 Tflop/s

Matrix matrix multiplication GEMM

\[ C = \alpha A + \beta B + C \]
Leveraging Half Precision in HPC on V100

Motivation

LU factorization is used to solve a linear system \( Ax = b \)

\[
\begin{align*}
LUx &= b \\
Ly &= b \\
\text{then} & \\
Ux &= y
\end{align*}
\]
Use Mixed Precision algorithms

**Idea:** use lower precision to compute the expensive flops ($LU \ O(n^3)$) and then iteratively refine the solution in order to achieve the FP64 arithmetic

- Achieve higher performance $\rightarrow$ faster time to solution
- Reduce power consumption by decreasing the execution time $\rightarrow$ **Energy Savings !!!**
Leveraging Half Precision in HPC on V100
Iterative Refinement

**Idea:** use lower precision to compute the expensive flops ($LU O(n^3)$) and then iteratively refine the solution in order to achieve the FP64 arithmetic.

Iterative refinement for dense systems, $Ax = b$, can work this way.

$L U = lu(A)$
$x = U\backslash(L\backslash b)$
$r = b - Ax$

**WHILE** $|| r ||$ not small enough

1. find a correction “$z$” to adjust $x$ that satisfy $Az = r$
   solving $Az = r$ could be done by either:
   - $z = U\backslash(L\backslash r)$ **Classical Iterative Refinement**
   - GMRes preconditioned by the LU to solve $Az = r$ **Iterative Refinement using GMRes**
2. $x = x + z$
3. $r = b - Ax$

**END**

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP $fl$ pt results when using DP $fl$ pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.
Leveraging FP16 in HPC on V100: Numerical Behavior

Convergence history for Classic Iterative Refinement
\[ n = 10240, \kappa_\infty(A) = 1.2e+04 \]

- Convergence history of the iterative refinement solver to achieve FP64 solution accuracy.
- Left graph shows the classical iterative refinement method.
- Right graph shows the iterative refinement using GMRes.
- Problem generated with an arithmetic distribution of the singular values \( \sigma_i = 1 - \frac{(i-1)}{n-1}(1 - \frac{1}{\text{cond}}) \) and positive eigenvalues.

The \textbf{FP32\textarrow{}64} algorithm converge as expected and is able to achieve the FP64 solution accuracy in about 3-5 iterations.

The \textbf{FP16\textarrow{}64} algorithm requires more iterations than FP32\textarrow{}64 because of the lower precision factorization.

The \textbf{FP16\textarrow{}64 (Tensor Cores)} outperforms the FP16\textarrow{}64 because the accumulation during the FP16-TC GEMM (Schur update) use FP32-bit.
Leveraging Half Precision in HPC on V100

Performance Behavior

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - \left( \frac{i-1}{n-1} \right)(1 - \frac{1}{\text{cond}})$ and positive eigenvalues.

Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- solving $Ax = b$ using FP64 LU
- solving $Ax = b$ using FP32 LU and iterative refinement to achieve FP64 accuracy
- solving $Ax = b$ using FP16 LU and iterative refinement to achieve FP64 accuracy
- solving $Ax = b$ using FP16 Tensor Cores LU and iterative refinement to achieve FP64 accuracy
Leveraging Half Precision in HPC on V100 Performance Behavior

Problem generated with an arithmetic distribution of the singular values \( \sigma_i = 1 - \left( \frac{i-1}{n-1} \right) \left( 1 - \frac{1}{\text{cond}} \right) \) and positive eigenvalues.

Flops = \( 2n^3/(3 \text{ time}) \), meaning twice higher is twice faster

- solving \( Ax = b \) using FP64 LU
- solving \( Ax = b \) using FP32 LU and iterative refinement to achieve FP64 accuracy
- solving \( Ax = b \) using FP16 LU and iterative refinement to achieve FP64 accuracy
- solving \( Ax = b \) using FP16 Tensor Cores LU and iterative refinement to achieve FP64 accuracy
Leveraging FP16 in HPC on V100: Numerical Behavior

- Convergence history of the iterative refinement solver to achieve FP64 solution accuracy.
- Left graph shows the classical iterative refinement method.
- Right graph shows the iterative refinement using GMRes.
- Problem generated with a clustered distribution of the singular values $\sigma = [1, \cdots, 1, \frac{1}{\text{cond}}]$.

- The FP32->64 algorithm converges as expected and is able to achieve the FP64 solution accuracy in about 3-5 iterations.
- The FP16->64 algorithm requires too many iterations and might be a performance bottleneck.
- The FP16->64 (Tensor Cores) outperforms the FP16->64 because the accumulation during the FP16-TC GEMM (Schur update) use FP32-bit.
Leveraging Half Precision in HPC on V100
Performance Behavior

Problem generated with an arithmetic distribution of the singular values $\sigma = [1, \cdots, 1, \frac{1}{\text{cond}}]$

- solving $Ax = b$ using FP64 LU
- solving $Ax = b$ using FP32 LU and iterative refinement to achieve FP64 accuracy
- solving $Ax = b$ using FP16 LU and iterative refinement to achieve FP64 accuracy
- solving $Ax = b$ using FP16 Tensor Cores LU and iterative refinement to achieve FP64 accuracy

Floors = $2n^3/(3 \text{ time})$ meaning twice higher is twice faster
Part IV

Nonstationary Iterative Methods
Nonstationary Iterative Methods

The methods involve information that changes at every iteration
- e.g. inner-products with residuals or other vectors, etc.

The methods are
- newer, harder to understand, but more effective
- in general based on the idea of orthogonal vectors and subspace projections
- examples: Krylov iterative methods
  - CG/PCG, GMRES, CGNE, QMR, BiCG, etc.
Krylov Iterative Methods
Krylov subspaces: these are the spaces

\[ K_i(A, r) = \text{span}\{r, Ar, A^2r, \ldots, A^{i-1}r\} \]

Krylov iterative methods find approximation \( x_i \) to \( x \) where

\[ Ax = b, \]

as a minimization on \( K_i(A, r) \).

We have seen how this can be done for example by projection, i.e. by the Petrov-Galerkin conditions:

\[ (Ax_i, \phi) = (b, \phi) \quad \text{for} \quad \forall \phi \in K_i(A, r) \]
Krylov Iterative Methods

In general we

- expend the Krylov subspace by a matrix-vector product, and
- do a minimization/projection in it.

Various methods result by specific choices of expansion and minimization/projection.
An example: The **Conjugate Gradient Method (CG)**

- The method is for SPD matrices
- There is a way at iteration $i$ to construct new 'search direction' $p_i$ s.t.

$$\text{span}\{p_0, p_1, \ldots, p_i\} \equiv K_{i+1}(A, r_0) \text{ and } (A p_i, p_j) = 0 \text{ for } i \neq j.$$ 

**Note:** $A$ is SPD $\Rightarrow (A p_i, p_j) \equiv (p_i, p_j)_A$ can be used as an inner product, i.e. $p_0, \ldots, p_i$ is an $(\cdot, \cdot)_A$ orthogonal basis for $K_{i+1}(A, r_0)$

$\Rightarrow$ we can easily find $x_{i+1} \approx x$ as

$$x_{i+1} = x_0 + \alpha_0 p_0 + \cdots + \alpha_i p_i \quad \text{s.t.}$$

$$(A x_{i+1}, p_j) = (b, p_j) \text{ for } j = 0, \ldots, i$$

Namely, because of the $(\cdot, \cdot)_A$ orthogonality of $p_0, \ldots, p_i$ at iteration $i + 1$ we have to find only $\alpha_i$

$$(A x_{i+1}, p_j) = (A(x_i + \alpha_i p_i), p_i) = (b, p_i), \quad \Rightarrow \quad \alpha_i = \frac{(r_i, p_i)}{(A p_i, p_i)}$$

**Note:** $x_i$ above actually can be replaced by any $x_0 + v, v \in K_i(A, r_0)$ (Why?)
Learning Goals

A brief introduction to iterative methods

- Motivation for iterative methods and links to previous lectures, namely
  - PDE discretization and sparse matrices
  - Optimized implementations

- Stationary iterative methods

- Nonstationary iterative methods and links to building blocks that we have already covered
  - Projection/Minimization
  - Orthogonalization

- Krylov methods; an example with CG; to see more examples ... (next lecture)