1. Comments/Answers on the homework on Gram-Schmidt

2. Matrix-vector multiply
   - Quick recall on CSR-AIJ format
   - Using a stencil
   - SSS matrices
   - FMM method
   - Hamiltonian operator

3. Rate of convergence of stationary iterative methods
Comments/Answers on the homework on Gram-Schmidt
Matrix-vector multiply
The AIJ storage format

For example, the AIJ storage format is like:

\[
\begin{bmatrix}
1 & 1 & A(I, J) \\
1 & 2 \\
1 & 4 \\
2 & 1 \\
2 & 2 \\
2 & 5 \\
3 & 3 \\
4 & 4 \\
4 & 7 \\
\vdots & \vdots & \vdots
\end{bmatrix}
\]

It requires to store 3nnz numbers (float and integer).
The Compressed Sparse Row (CSR) storage format

For example, the CSR storage format is like:

\[
\begin{pmatrix}
1 & 1 & A(I,J) \\
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
4 & 4 & 4 & 4 \\
4 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
5 & 5 & 5 & 5 \\
7 & 3 & 3 & 3 \\
8 & 4 & 4 & 4 \\
7 & 7 & 7 & 7 \\
\vdots & \vdots & \vdots & \vdots \\
\end{pmatrix}
\]

It requires to store $2nnz + n$ numbers (float and integer).
Some storage formats

See in *Templates for the Solution of Linear Systems*

- AIJ Storage (AIJ)
- Compressed Row Storage (CRS)
- Compressed Column Storage (CCS)
- Block Compressed Row Storage (BCRS)
- Block Compressed Column Storage (BCCS)
- Compressed Diagonal Storage (CDS)
- Jagged Diagonal Storage (JDS)
- Skyline Storage (SKS)

Matrix-vector multiply
Matrix-vector multiply
Quick recall on CSR-AIJ format

The Compressed Sparse Row (CSR) matrix-vector product

<table>
<thead>
<tr>
<th>val</th>
<th>a11</th>
<th>a12</th>
<th>a14</th>
<th>a21</th>
<th>a23</th>
<th>a25</th>
<th>a33</th>
<th>a44</th>
<th>a47</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_ind</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>row_ptr</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On an Itanium2 machine (results from Haihang and Keith):

\[
\begin{array}{lll}
\text{MFlops} & \text{Machine Peak} & (\% \\
\text{GEMM (ATLAS)} & 3600.0 & (100.0) \\
\text{GEMV (ATLAS)} & 2188.1 & (60.8) \\
\text{CSR gcc-4.3.1} & 1222.8 & (33.4) \\
\text{CSR icc-8} & 26.6 & (0.7) \\
\end{array}
\]

```c
for(i=0; i<nrow; i++, y+=1)
    register double y0=*y;
    for(j=row_ptr[i]; j<row_ptr[i+1]; j++, col_idx++, a+=1)
        int kk=*col_idx;
        y0 += (*a) * x[kk];
    }
*y=y0;
```

...
Matrix-vector multiply
Quick recall on CSR-AIJ format

The Compressed Sparse Row (CSR) matrix-vector product

<table>
<thead>
<tr>
<th>val</th>
<th>a11</th>
<th>a12</th>
<th>a14</th>
<th>a21</th>
<th>a23</th>
<th>a25</th>
<th>a33</th>
<th>a44</th>
<th>a47</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_ind</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>row_ptr</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... for(i=0;i<nrow;i++,y+=1){
    register double y0=*y;
    for(j=row_ptr[i];j<row_ptr[i+1];j++,col_idx++,a+=1){
        int kk=*col_idx;
        y0 += (*a) * x[kk];
    }
    *y=y0;
}...

- Indirect addressing
- Low ratio between floating-point operations and memory accesses
- Low spatial and temporal locality for source vector
Solution to get better performance

In sequential
- blocking by explicit fill-in
- technique a la ATLAS

In parallel (minimizing the communication)
- ghostpoint
- solution partitioning (reordering, see Stanimire results)
Using ParMETIS (from Stanimire)

Using ParMETIS (or Chaos, or any graph partitioner package) enables us:

- load balance the
- minimize the interface
Sparse BLAS libraries

- A standard has been made, reference implementation exists [http://www.netlib.org/blas/blast-forum/](http://www.netlib.org/blas/blast-forum/)

- Optimized libraries can be found at [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>
<th>Dense</th>
<th>Sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>Complex</td>
<td>f77</td>
<td>c</td>
<td>c++</td>
</tr>
<tr>
<td>MTL</td>
<td>yes</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NETMAT</td>
<td>yes</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NIST S-BLAS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PSBLAS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SparseLib++</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>uBLAS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Using a pencil

Matrix-vector multiply

Using a stencil
Poisson equation in 2D

Let us consider the Poisson equation:

\[
\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = \Delta u = f, \quad \text{for } x = \left( \begin{array}{c} x_1 \\ x_2 \end{array} \right) \text{ in } \Omega
\]

where \( \Omega \) is a bounded, open domain in \( \mathbb{R}^2 \).

On the boundary of \( \Omega \), \( \partial \Omega \), we impose Dirichlet boundary condition

\[
u(x_1, x_2) = \phi(x_1, x_2) \text{ for } \left( \begin{array}{c} x_1 \\ x_2 \end{array} \right) \text{ in } \partial \Omega
\]

(\( \phi(x_1, x_2) \) and \( f(x_1, x_2) \) are known, the unknown is \( u(x_1, x_2) \).)
Difference Scheme in 1D for the Laplacean operator

We want to approximate the Laplacean \( \frac{d^2 u}{dx^2} \) in \( x \) knowing \( u(x), u(x + h), u(x - h) \):

\[
\begin{align*}
u(x + h) &= u(x) + h \frac{du}{dx} (x) + \frac{h^2}{2} \frac{d^2 u}{dx^2} (x) + O(h^3) \\
u(x - h) &= u(x) - h \frac{du}{dx} (x) + \frac{h^2}{2} \frac{d^2 u}{dx^2} (x) + O(h^3)
\end{align*}
\]

\[\Rightarrow \frac{d^2 u}{dx^2} (x) = \frac{1}{h^2} (u(x + h) + u(x - h) - 2u(x)) + O(h)\]
We want to approximate $\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$ in $(x_1, x_2)$ knowing $u(x_1, x_2), u(x_1 + h, x_2), u(x_1 - h, x_2), u(x_1, x_2 + h), u(x_1, x_2 - h)$ (the mesh is uniform)

\[
\begin{align*}
  u(x_1 + h, x_2) &= u(x_1, x_2) + h \frac{\partial u}{\partial x_1}(x_1, x_2) + \frac{h^2}{2} \frac{\partial^2 u}{\partial x_1^2}(x_1, x_2) + \mathcal{O}(h^3) \\
  u(x_1 - h, x_2) &= u(x_1, x_2) - h \frac{\partial u}{\partial x_1}(x_1, x_2) + \frac{h^2}{2} \frac{\partial^2 u}{\partial x_1^2}(x_1, x_2) + \mathcal{O}(h^3) \\
  u(x_1, x_2 + h) &= u(x_1, x_2) + h \frac{\partial u}{\partial x_2}(x_1, x_2) + \frac{h^2}{2} \frac{\partial^2 u}{\partial x_2^2}(x_1, x_2) + \mathcal{O}(h^3) \\
  u(x_1, x_2 - h) &= u(x_1, x_2) - h \frac{\partial u}{\partial x_2}(x_1, x_2) + \frac{h^2}{2} \frac{\partial^2 u}{\partial x_2^2}(x_1, x_2) + \mathcal{O}(h^3)
\end{align*}
\]

$\Rightarrow \Delta u(x_1, x_2) = \frac{1}{h^2} (u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)) + \mathcal{O}(h)$

(The five-point centered approximation for the Laplacean.)
Let us consider the one-dimensional equation:

\[ \frac{d^2 u}{dx^2}(x) = f(x), \quad \text{for } x = (0, 1) \]

and the Dirichlet boundary condition

\[ u(0) = u(1) = 0 \]

The interval \([0, 1]\) is discretized uniformly with \(n + 2\) points

\[ x_0 \quad x_1 \quad \cdots \quad x_{n-1} \quad x_n \]

\[ h \quad h \quad \cdots \quad h \]

At any point \(x_i\) in \(\Omega\) (the open domain), we are seeking for \(u_i\) an approximation of the exact solution \(u(x_i)\). Using the centered difference approximation

\[ -u_{i-1} + 2u_i - u_{i+1} = h^2 f_i, \]

and the fact that \(u_0 = u_{n+1} = 0\),

we obtain a linear system of the form

\[ Ax = b \]

where \(b = (f_i)_{i=1,n}\) and \(x = (u_i)_{i=1,n}\) and

\[
A = \frac{1}{h^2} \begin{pmatrix}
2 & -1 & 0 & \cdots & 0 \\
-1 & 2 & -1 & \cdots & 0 \\
0 & -1 & 2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & -1 & 2 & -1 \\
0 & 0 & \cdots & 0 & -1 & 2 & -1
\end{pmatrix}
\]
Let us consider the Poisson equation:
\[
\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = \Delta u = f, \quad \text{for } x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \text{ in } \Omega
\]
and the Dirichlet boundary condition
\[
u(x_1, x_2) = \phi(x_1, x_2) \text{ for } \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \text{ in } \partial \Omega
\]

The interval \([0, 1] \times [0, 1]\) is discretized uniformly with \((n + 2) \times (n + 2)\) points

\[
A = \frac{1}{h^2} \begin{pmatrix} B & -I & -1 & -I & -1 & \cdots & -I & -1 & -1 \\ -I & B & -1 & -I & -1 & \cdots & -I & -1 & -1 \\ -I & -I & B & -I & -1 & \cdots & -I & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ -I & -I & -I & B & -I & \cdots & -I & -1 & -1 \\ -I & -I & -I & -I & B & \cdots & -I & -1 & -1 \\ \end{pmatrix}
\]

where
\[
B = \frac{1}{h^2} \begin{pmatrix} 4 & -1 & -1 & \cdots & -1 \\ -1 & 4 & -1 & \cdots & -1 \\ -1 & -1 & 4 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & 4 \\ -1 & -1 & -1 & \cdots & -1 \\ \end{pmatrix}
\]
## Demo of pencil sparse matrix-vector product

- how to set the right-hand side? Physical interpretation of $A$ enable to check easily the results.
- tune various codes? you can do one mat-vec easily: is it possible to have a better one
- discussion with multiple right-hand sides % of the peak, is it better to have one RHS or several
- interfacing those code with some solvers
- what about parallel implementation?
example of interfacing with unsymmetric solver for the 1D operator

- GMRES (sparse iterative methods, need matrix-vector product)
- UMFPACK (sparse direct solver (AMD), need CSC format)
- LAPACK (dense direct solver, dense banded format)
Using a pencil
Matrix-vector multiply
Using a stencil

Fill-in example – 1D Laplacean – \( n = 64 \) – \( \text{nnz} = 288 \) – Symb. Cholesky

<table>
<thead>
<tr>
<th>Permutation</th>
<th>( \text{nnz in } R )</th>
<th>( \text{fill-in in } R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>974</td>
<td>3.81</td>
</tr>
<tr>
<td>Symmetric approximate minimum degree</td>
<td>644</td>
<td>2.24</td>
</tr>
<tr>
<td>Symmetric reverse Cuthill-McKee</td>
<td>792</td>
<td>2.75</td>
</tr>
</tbody>
</table>
Using a pencil
Matrix-vector multiply
SSS matrices

We consider the matrices $A$ and $B$ on the left, right-hand side $b$ is random, we want to solve the problem such that $\|b - Ax\| \leq 10^{-12} \|b\|$, this matrix is SPD.

- observe that $A$ is a matrix of rank one + a diagonal matrix, that $B$ is a matrix of rank 2 + a diagonal matrix
- write a clever way to store and apply the matrix-vector product by $A$
- show that the minimal polynomial of $A$ is of order 2, and the one of $B$ is of order 3
- choose a Krylov type iterative solver to solve your problem (most of the solvers are built-in functions in Matlab); solve it how many iterations does it need to converge? why (wait for Victor class, hint: look to the minimal polynomial)?
- Let us forget iterative methods: can you write a clever direct method to solve the problem? (hint: Sherman-Morrison)

$$A = \begin{pmatrix}
  n & 1 & 1 & 1 & \ldots & 1 \\
  1 & n & 1 & 1 & \ldots & 1 \\
  1 & 1 & n & 1 & \ldots & 1 \\
  1 & 1 & 1 & n & \ldots & 1 \\
  1 & 1 & 1 & 1 & \ldots & n
\end{pmatrix}$$

$$B = \begin{pmatrix}
  10 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\
  1 & 10 & 1 & 1 & 2 & 2 & 2 & 2 \\
  1 & 1 & 10 & 1 & 2 & 2 & 2 & 2 \\
  1 & 1 & 1 & 10 & 2 & 2 & 2 & 2 \\
  2 & 2 & 2 & 2 & 10 & 1 & 1 & 1 \\
  2 & 2 & 2 & 2 & 1 & 10 & 1 & 1 \\
  2 & 2 & 2 & 2 & 1 & 1 & 10 & 1 \\
  2 & 2 & 2 & 2 & 1 & 1 & 1 & 10
\end{pmatrix}$$

**NB:**
the conclusion is that iterative methods can also be used efficiently on dense operator provided we manage to exploit an interesting structure. this is related to FFT and FFM methods that arise for semi separable operator (boundary element method in electromagnetism, companion matrix, ...
Electromagnetism application

\[ Z_{j, \ell} = - \int_\Gamma \int_\Gamma ikZ_0 \frac{e^{ik|y-x|}}{4\pi|y-x|} \left( \Psi_j(x) \cdot \Psi_\ell(y) - \frac{1}{k^2} \text{div}_\Gamma \Psi_j(x) \text{div}_\Gamma \Psi_\ell(y) \right) , \]

\[ \Rightarrow \text{Z is} \]

- complex,
- symmetric (in this case),
- dense,
- large

and the problem has multiple right-hand side.
Using a pencil
Matrix-vector multiply
FMM method
Using a pencil
Matrix-vector multiply
FMM method
Using a pencil
Matrix-vector multiply
FMM method
Using a pencil
Matrix-vector multiply
FMM method
Using a pencil
Matrix-vector multiply
FMM method
Using a pencil
Matrix-vector multiply
FMM method
The Hamiltonian operator coming from the Schrödinger equation (material science) is of the form:

\[ H = -\frac{1}{2} \nabla^2 + V \]

where

1. \(-\frac{1}{2} \nabla^2\) is diagonal in the Fourier space
2. \(V\) is (often) diagonal in the real space

Some codes operate in the Fourier space, when \(Hx\) is needed:

1. \(y \leftarrow \text{InvFFT3D}(x)\)
2. \(y \leftarrow V(y)\)
3. \(x \leftarrow \text{FFT3D}(y)\)
4. \(x \leftarrow -\frac{1}{2} \nabla^2 x + x\)
Rate of convergence of stationary iterative methods
Suppose you want to solve the system of linear equations

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + \ldots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + \ldots + a_{2n}x_n &= b_2 \\
    a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + \ldots + a_{3n}x_n &= b_3 \\
    \vdots \\
    a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + a_{n4}x_4 + \ldots + a_{nn}x_n &= b_n
\end{align*}
\]

then if an initial guess \(x^{(0)}\) is given, we can compute \(^1 x^{(1)}:\)

\[
\begin{align*}
    x_1^{(1)} &= (b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)} - \ldots - a_{1n}x_n^{(0)}) / a_{11} \\
    x_2^{(1)} &= (b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)} - \ldots - a_{2n}x_n^{(0)}) / a_{22} \\
    x_3^{(1)} &= (b_2 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)} - \ldots - a_{3n}x_n^{(0)}) / a_{22} \\
    \vdots \\
    x_n^{(1)} &= (b_n - a_{n1}x_1^{(0)} - a_{n2}x_2^{(0)} - a_{n3}x_3^{(0)} - \ldots - a_{n,n-1}x_{n-1}^{(0)}) / a_{nn}
\end{align*}
\]

\(^1\text{we assume } a_{ii} \neq 0\)
Rate of convergence of stationary iterative methods

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

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

<table>
<thead>
<tr>
<th>( x^{(0)} )</th>
<th>( x^{(1)} )</th>
<th>( x^{(2)} )</th>
<th>( x^{(3)} )</th>
<th>( x^{(4)} )</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>2.3000</td>
<td>0.6000</td>
<td>0.9330</td>
<td>0.9893</td>
<td>1.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>3.6667</td>
<td>1.6639</td>
<td>1.9380</td>
<td>1.9895</td>
<td>2.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>4.0000</td>
<td>2.3852</td>
<td>2.9107</td>
<td>2.9865</td>
<td>3.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>4.9000</td>
<td>3.4000</td>
<td>3.9133</td>
<td>3.9871</td>
<td>4.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>5.3333</td>
<td>4.6911</td>
<td>4.9572</td>
<td>4.9938</td>
<td>5.0000</td>
</tr>
</tbody>
</table>
At each step, we are performing:

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

The number of Flops needed per iteration is \(2n^2\) Flops

The number of Flops needed to converge is 
\(nb\_iter \times 2n^2 = nb\_iter \times Flops_{\text{matvec}}\) Ideally \(nb\_iter\) is independent of the size of the problem which makes the method scale as \(O(n^2)\). (In practice, this is not always the case.)
A more compact formula for Jacobi’s method

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

\[
x^{(i+1)} = D^{-1}(b - Ax^{(i)}) + x^{(i)} = D^{-1}r^{(i)} + x^{(i)}
\]

Note that we just need to know the diagonal of \( A \) and then access the matrix-vector product.
Richardson method

Assume we know $M^{-1}$ a good approximate of the inverse of $A$. Then we can compute

$$x_1 = M^{-1}b$$

as a first approximation of the exact solution $x$. The residual $r = b - Ax_1$ is not zero so we are looking for a correction $e$ to $x_1$ such that $x_2 = x_1 + e$ is a better approximation. The goal is

$$Ax_2 \sim b.$$ 

which means

$$A(x_1 + e) \sim b \Rightarrow Ae \sim (b - Ax_1) = r_1$$

Naturally we will take

$$e = M^{-1}r_1$$
Richardson method

The algorithm is then

1: initialize $x$
2: $r = b - Ax$
3: check if $x$, $r$ ok
4: for $k = 1, \ldots$ do
5: \hspace{1em} $e = M^{-1}r$
6: \hspace{1em} $x = x + e$
7: \hspace{1em} $r = b - Ax$
8: \hspace{1em} check if $x$, $r$ ok
9: end for

Richardson: $x^{(i+1)} = r^{(i)} + x^{(i)}$
Jacobi: $x^{(i+1)} = D^{-1}r^{(i)} + x^{(i)}$

1. To be accurate Richardson’s method does not have a preconditioner
2. If $M = D$ then Richardson is Jacobi
3. Richardson is used in iterative refinement
Gauss-Seidel method

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

**Richardson:**

\[x^{(i+1)} = r^{(i)} + x^{(i)}\]

**Jacobi:**

\[x^{(i+1)} = D^{-1}r^{(i)} + x^{(i)}\]

**Gauss-Seidel:**

\[x^{(i+1)} = (D + L)^{-1}r^{(i)} + x^{(i)}\]

1. \(D\) is the diagonal of \(A\)
2. \(E\) is the upper part of \(A\)
3. \(F\) is the lower part of \(A\)
4. \(A = D + E + F\)
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>\ldots</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>
Jacobi Method:

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

Gauss-Seidel method

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

Gauss-Seidel is the method with the best numerical properties (less iteration to convergence) What is the method with the best 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 \(x_k^{(i+1)}\). Parallelisation is impossible.
Richardson, Gauss-Seidel and Jacobi methods:

- Richardson, Gauss-Seidel and Jacobi methods are part of the **stationary iterative methods** (these are the methods that just kept one vector from one iteration to the others \((x^{(i)})\)).

- Those methods are not that efficient in practice (they do not converge very well), the example in this section are somehow *special*.
First conclusion that holds for any iterative method

+ to be considered in particular if good initial guess
+ the solve will not be exact but approximate
- no easy way to deal with multiple right-hand sides

First conclusion that holds for any stationary iterative method

+ extremely robust to errasure/error during the computation
We have seen that Jacobi method can be written:

\[
x^{(i+1)} = D^{-1}(b - Ax^{(i)}) + x^{(i)} = D^{-1}b + (I - D^{-1}A)x^{(i)}
\]
We have seen that Jacobi method can be written: Note that, if $x^{(*)}$ is the exact solution, we also have

$$x^{(*)} = D^{-1}b + (I - D^{-1}A)x^{(*)}$$

thus

$$(x^{(i+1)} - x^{(*)}) = (I - D^{-1}A)(x^{(i)} - x^{(*)}) = \ldots = (I - D^{-1}A)^i(x^{(0)} - x^{(*)})$$

naming $e^{(i)} = x^{(i)} - x^{(*)}$, the error at step $i$:

$$e^{(i+1)} = (I - D^{-1}A)^i e^{(0)}$$

or

$$r^{(i+1)} = (I - AD^{-1})^i r^{(0)}$$

**Theorem:** If $\rho(I - D^{-1}A) < 1$, then the Jacobi method converges. Convergence is asymptotically linear.

link with the diagonal dominancy of $A$
Rate of convergence of stationary iterative methods

Rate of convergence of stationary iterative methods

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

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

Jacobi \quad \rho(I - D^{-1}A) = 0.324

Gauss-Seidel \quad \rho(I - (D + L)^{-1}A) = 0.062
Nonsymmetric Linear System
Krylov iterative methods
The Krylov $\mathcal{K}(A, r_0, n)$ space is the space

$$\mathcal{K}(A, r_0, i) = (r_0, Ar_0, A^2r_0, \ldots, A^i r_0).$$

Krylov iterative methods are projection methods that seeks at each step $i$ an approximate solution of $Ax = b$ in the Krylov space $\mathcal{K}(A, r_0, i)$.

The basic steps are

- expand the space (this is done by a matrix-vector product)
- find an approximate solution in this space according to a given rule

Depending on how you do those two steps you are define your methods.
why seeking the solution in a Krylov space?

- generalization of stationary methods with $D^{-1}$ a preconditioner
  \[ x^{(i+1)} = D^{-1} r^{(i)} + x^{(i)} \]

- why Krylov space? $A^{-1}$ is a polynomial of degree less than $n$ in $A$
GCR: Generalized Conjugate Residuals

The first Krylov method for nonsymmetric system was GCR. It is still have advantage that makes it a solution to consider in some cases.

expand the space (this is done by a matrix-vector product)

Note that

\[ \mathcal{K}(A, r_0, i) = (r_0, Ar_0, A^2r_0, A^3r_0, \ldots, A^i r_0) \]

\[ = (r_0, Ar_0, Ar_1, Ar_2, \ldots, Ar_{i-1}) = R^{(i)} \]

The expansion in GCR at step \( i \) is done via: \( c_i' \leftarrow Ar_i \). The basis \( C_i' \) is such that

\[ A(r_0, r_1, \ldots, r_i) = (c_0', c_1', \ldots, c_i') \quad \text{or} \quad AR_i = C_i' \]

\( C_i' \) is a basis of \( A\mathcal{K}(A, r_0, i) \) The search space will be \( R_i \).
find an approximate solution $x_{i+1}$

A natural choice is to look for $x_{i+1}$ that

$$\min_{x \in \mathcal{K}(A, r_0, i)} (\|b - Ax\|_2)$$

which is equivalent to impose the Petrov-Galerkin condition

$$r_{i+1} \perp A\mathcal{K}(A, r_0, i)$$

which means

$$r_{i+1} \perp C_{i+1}$$
GCR: Generalized Conjugate Residuals

Normalization

\[ r_{i+1} \perp C_{i+1} \]

A convenient way to go is to consider at each step \( i \), \( C_i \) the orthonormal basis of \( C'_i \)

\[ C_i = C'_i \times H, \text{ where } H \text{ upper triangular and } C'^T_i C_i = I \]

A second basis \( U_i \) is stored to keep the relation

\[ AR_i = C'_i \]

\[ AU_i = C_i \]

(by imposing \( U_i = C'_i H \))
1: Choose a convergence threshold $\epsilon$
2: Choose an initial guess $x_0$
3: $r_0 = b - Ax_0; \beta = \|r_0\|_2$
4: for $k = 0, 2, \ldots$ do
5: \ $u_{i+1} = r_i$
6: \ $c^\prime_{i+1} = Au_i$
7: \ orthonormalize $c^\prime_{i+1}$ to obtain $c_{i+1}$ against $c_1, \ldots, c_i$
8: \ same operation on $u_{i+1}$ with $u_1, \ldots, u_i$ (to maintain the relation $AC_{i+1} = U_{i+1}$)
9: \ $\alpha = c^\top_{i+1} r_i$
10: \ $r_{i+1} = r_{i+1} - c_{i+1} \alpha$ (orthogonal proj of $r_i$ on $c_{i+1}$)
11: \ $x_{i+1} = x_{i+1} + u_{i+1} \alpha$ (maintain the relation $r = b - Ax$)
12: end for
GMRES: Generalized minimum Residual

1: Choose a convergence threshold \( \epsilon \)
2: Choose an initial guess \( x_0 \)
3: \( r_0 = b - Ax_0; \beta = \|r_0\|_2 \)
4: \( v_1 = r_0 / \|r_0\|_2; \)
5: for \( k = 1, 2, \ldots \) do
6: \( z = Av_k \)
7: for \( i = 1 \) to \( k \) do
8: \( h_{i,k} = v_i^T z \)
9: \( z = z - h_{i,k} v_i \)
10: end for
11: \( h_{k+1,k} = \|z\|_2 \)
12: \( v_{k+1} = z / h_{k+1,k} \)
13: Solve the least-squares problem \( \min \| \beta e_1 - \tilde{H}_k y_k \|_2 \) for \( y_k \)
14: if \( \| \tilde{r}_k \|_2 = \| \beta e_1 - \tilde{H}_k y_k \|_2 \leq \epsilon \| A \|_2 \| x_k \|_2 \) then
15: set \( x_k = x_0 + V_k y_k \)
16: exit
17: end if
18: end for

- GMRES is equivalent to GCR (mathematically) but requires twice as less storage vectors (twice as less vector operations as well),
- GMRES does not allow variable preconditioners
- Convergence in \( n \) steps guaranteed (in fact convergence in the degree of the minimal polynomial of \( A \) guaranteed), like GCR
- Good behavior versus floating point arithmetic (stable). This statement depends however on the orthogonalization schemes used
- Least squares problem solves via Givens rotation
Krylov iterative methods
Nonymmetric Linear System

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

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

GMRES converges in at most $n$ steps......
That’s a bit too much: it means to store $n$ vectors !!

In practice this is real bottleneck, need to apply restart of the algorithm (i.e after $m$ step, you stop the method, compute the residual and restart from the approximate solution as an initial guess).

GMRES with restart
- stores just $m$ vectors
- have a short orthogonalization scheme
- but converges more slowly than GMRES ! can even stagnates

Cure:
GMRES-DR, short term recurrences (see Victor)
Symmetric Positive Definite Linear system
Symmetric Indefinite Linear System
Symmetric Indefinite Linear System

You can use a short term recurrence in this case, the problem is that the inner product with respect to $A$ is not definite and so those methods can in practice fails.

<table>
<thead>
<tr>
<th>Method</th>
<th>Matrix</th>
<th>Preconditioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symm-QMR</td>
<td>Indefinite</td>
<td>Indefinite</td>
</tr>
<tr>
<td>MINRES</td>
<td>Indefinite</td>
<td>Definite</td>
</tr>
<tr>
<td>LSQR (SYMMLQ)</td>
<td>Indefinite</td>
<td>Definite</td>
</tr>
<tr>
<td>CG</td>
<td>SPD</td>
<td>SPD</td>
</tr>
</tbody>
</table>
Multiple right-hand Sides
Symmetric Eigenvalue Problem

Symmetric Eigenvalue Problem

\[ H\psi = \psi E \]

Find a few (4-10) of the interior eigenvalues of a matrix \( H \) the closest to a given point \( E_{\text{ref}} \)

- \( H \) is complex, Hermitian (eigenvalues are real, negative and positive)
- \( H \) is accessed via a matrix-free matrix-vector product
- Multiple eigenvalues to be expected (say max degeneracy is 4)
Methods

1. Iterative method family

2. Spectral Transformation

3. Preconditioners

4. Kernel Operations
Methods

1. Iterative method family
   - Lanczos
   - Jacobi-Davidson
   - Conjugate Gradient

2. Spectral Transformation
   -
   -
   -

3. Preconditioners
   -
   -

4. Kernel Operations
   -
   -
Methods

1. Iterative method family
   - Lanczos
   - Jacobi-Davidson
   - Conjugate Gradient

2. Spectral Transformation
   - Folded Spectrum \((H - E_{\text{ref}})^2\)
   - Shift-and-Invert \((H - E_{\text{ref}})^{-1}\)

3. Preconditioners

4. Kernel Operations
Methods

1. Iterative method family
   - Lanczos
   - Jacobi-Davidson
   - Conjugate Gradient

2. Spectral Transformation
   - Folded Spectrum \((H - E_{\text{ref}})^2\)
   - Shift-and-Invert \((H - E_{\text{ref}})^{-1}\)

3. Preconditioners
   - diagonal preconditioner

4. Kernel Operations

\[ H = -\frac{1}{2} \nabla^2 + V \]

if Folded Spectrum \((H - E_{\text{ref}})^2\)

\[ p_i = \frac{E_k^2}{\left(\frac{1}{2} q_i^2 + V_0 - E_{\text{ref}}\right)^2 + E_k^2}, \]

where

- \(q_i\) is the diagonal term of the Laplacian,
- \(V_0\) is the average potential and \(E_k\) is the average kinetic energy of the wave function \(\psi_i\).
Methods

1. Iterative method family
   - Lanczos
   - Jacobi-Davidson
   - Conjugate Gradient

2. Spectral Transformation
   - Folded Spectrum \((H - E_{\text{ref}})^2\)
   - Shift-and-Invert \((H - E_{\text{ref}})^{-1}\)

3. Preconditioners
   - diagonal preconditioner

4. Kernel Operations
   - matrix-vector product based on 3D-FFTs

\[ H = -\frac{1}{2} \nabla^2 + V \]

The code is in the Fourier space, when \(Hx\) is needed:

1. \(y \leftarrow \text{InvFFT}3D(x)\)
2. \(y \leftarrow V(y)\)
3. \(x \leftarrow \text{FFT}3D(y)\)
4. \(x \leftarrow -\frac{1}{2} \nabla^2 x + x\)
Quantum Dots of 78 Cadmium atoms and 69 Selenium atoms
Matrix 29,423; 16 processors (IBM-SP3); Eref = -4.8 eV; 7
eigencouples requested; tol= 10^{-4}

<table>
<thead>
<tr>
<th>Method</th>
<th># it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG on Folded Spectrum</td>
<td>5,100</td>
<td>90s</td>
</tr>
<tr>
<td>Lanczos</td>
<td>500</td>
<td>43s</td>
</tr>
<tr>
<td>IRA(60)</td>
<td>1,059</td>
<td>27s</td>
</tr>
<tr>
<td>IRA(60) + Shift-and-Invert</td>
<td>50+5,000</td>
<td>213s</td>
</tr>
<tr>
<td>IRA(60) + Folded Spectrum</td>
<td>532</td>
<td>49s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Folded</th>
<th>Lanczos</th>
<th>IRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.874</td>
<td>-3.198</td>
<td>-3.198</td>
<td></td>
</tr>
<tr>
<td>-1.911</td>
<td>-6.594</td>
<td>-6.594</td>
<td></td>
</tr>
<tr>
<td>-2.537</td>
<td>-6.594</td>
<td>-6.594</td>
<td></td>
</tr>
<tr>
<td>-3.198</td>
<td>-6.632</td>
<td>-6.632</td>
<td></td>
</tr>
<tr>
<td>-3.198</td>
<td>-6.632</td>
<td>-6.632</td>
<td></td>
</tr>
<tr>
<td>-6.594</td>
<td>-6.691</td>
<td>-6.691</td>
<td></td>
</tr>
<tr>
<td>-6.762</td>
<td>-6.784</td>
<td>-6.820</td>
<td></td>
</tr>
</tbody>
</table>
Quantum Dots of 78 Cadmium atoms and 69 Selenium atoms  
Matrix 29,423; 16 processors (IBM-SP3); $E_{\text{ref}} = -4.8 \text{ eV}$; 7 eigencouples requested; tol$= 10^{-4}$

<table>
<thead>
<tr>
<th></th>
<th># it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG on Folded Spectrum</td>
<td>5,100</td>
<td>90s</td>
</tr>
<tr>
<td>Lanczos</td>
<td>500</td>
<td>43s</td>
</tr>
<tr>
<td>IRA(60)</td>
<td>1,059</td>
<td>27s</td>
</tr>
<tr>
<td>IRA(60) + Shift-and-Invert</td>
<td>50 + 5,000</td>
<td>213s</td>
</tr>
<tr>
<td>IRA(60) + Folded Spectrum</td>
<td>532</td>
<td>49s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Folded</th>
<th>Lanczos</th>
<th>IRA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1.874</td>
<td>-1.911</td>
<td>-2.537</td>
</tr>
<tr>
<td></td>
<td>-3.198</td>
<td>-3.198</td>
<td>-3.198</td>
</tr>
<tr>
<td></td>
<td>-6.594</td>
<td>-6.594</td>
<td>-6.594</td>
</tr>
<tr>
<td></td>
<td>-6.594</td>
<td>-6.594</td>
<td>-6.594</td>
</tr>
<tr>
<td></td>
<td>-6.632</td>
<td>-6.632</td>
<td>-6.632</td>
</tr>
<tr>
<td></td>
<td>-6.632</td>
<td>-6.632</td>
<td>-6.632</td>
</tr>
<tr>
<td></td>
<td>-6.691</td>
<td>-6.691</td>
<td>-6.691</td>
</tr>
<tr>
<td></td>
<td>-6.762</td>
<td>-6.784</td>
<td>-6.820</td>
</tr>
</tbody>
</table>

Lack of robustness of Lanczos in our current implementation did not try: block Lanczos, consecutive runs, ...
Nonlinear Conjugate Gradient Methods for eigenvalue problems
Nonlinear Conjugate Gradient Methods for eigenvalue problems

The minimum eigenvalue $\lambda_{\text{min}}$ minimizes the Rayleigh quotient function

$$f(x) = \frac{x^H H x}{x^H x}$$

where the gradient is

$$\nabla f(x) = H x - x \left( \frac{x^H H x}{x^H x} \right) = r(x)$$
Nonlinear Conjugate Gradient Methods for eigenvalue problems

$$f(x) = \frac{x^H H x}{x^H x} \quad \nabla f(x) = Hx - x \frac{x^H H x}{x^H x} = r(x)$$

1: Given $x_0$
2: Evaluate $f_0 = f(x_0)$, $\nabla f(x_0)$;
3: Set $p_0 = -\nabla f_0$, $k \rightarrow 0$;
4: while $\nabla f_k \neq 0$ do
5: \hspace{1em} Compute $\alpha_k$ and set $x_{k+1} = x_k + \alpha_k p_k$;
6: \hspace{1em} Evaluate $\nabla f_{k+1}$
7: \hspace{2em} $\beta_{k+1} \leftarrow -\frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_{k}^T \nabla f_{k}}$;
8: \hspace{1em} $p_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1} p_k$;
9: \hspace{1em} $k \leftarrow k + 1$;
10: end while

copied from Nocedal and Wright Numerical Optimization
Nonlinear Conjugate Gradient Methods for eigenvalue problems

\[ f(x) = \frac{x^H H x}{x^H x} \quad \nabla f(x) = H x - x \frac{x^H H x}{x^H x} = r(x) \]

1: Given \( x_0 \)
2: Evaluate \( f_0 = f(x_0), \nabla f(x_0) \);
3: Set \( p_0 = -\nabla f_0, k \rightarrow 0; \)
4: while \( \nabla f_k \neq 0 \) do
5: Compute \( \alpha_k \) and set \( x_{k+1} = x_k + \alpha_k p_k \);
6: Evaluate \( \nabla f_{k+1} \)
7: \( \beta_{k+1} \leftarrow -\frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k} \);
8: \( p_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1} p_k \);
9: \( k \leftarrow k + 1; \)
10: end while

copied from Nocedal and Wright Numerical Optimization
Find $\theta$ such that

$$x_{k+1} = x_k \cos \theta + p_k \sin \theta$$

minimizes the Rayleigh quotient function.
Find $\theta$ such that

$$x_{k+1} = x_k \cos \theta + p_k \sin \theta$$

minimizes the Rayleigh quotient function.

$$\theta_{\text{min}} = \frac{1}{2} \tan^{-1} \left( - \frac{\left( \frac{\partial f}{\partial \theta} \right)_{\theta=0}}{\frac{1}{2} \left( \frac{\partial^2 f}{\partial^2 \theta} \right)_{\theta=0}} \right)$$

$$\frac{\partial f}{\partial \theta} = 2 \text{ Re}(p_k^H Hx_k)$$

$$\frac{\partial^2 f}{\partial^2 \theta} = 2p_k^H Hp_k - x^H Hx$$
Nonlinear Conjugate Gradient Methods for eigenvalue problems

```plaintext
1  do i = 1, niter
4    ax = A X
6    λ = X \cdot ax
7    if (||ax − λ X||_2 < tol) exit
8    r_{j+1} = (I - X X^T) ax
9    β = \frac{(r_{j+1} - r_j) \cdot Pr_{j+1}}{r_j \cdot Pr_j}
10    d_{j+1} = -P r_{j+1} + β d_j
11    d_{j+1} = (I - XX^T)d_{j+1}
12    γ = \frac{||d_{j+1}||_2^{-1}}{2 γ^2 \cdot \lambda(m)}
13    θ = 0.5 \left| \text{atan} \frac{2 γ \cdot d_{j+1} \cdot ax}{λ(m) - γ^2 \cdot d_{j+1} \cdot A d_{j+1}} \right|
14    X = \cos(θ) X + \sin(θ) γ d_{j+1}
15    ax = \cos(θ) ax + \sin(θ) γ A d_{j+1}
19  enddo
+ Polak-Ribière method
+ Preconditioner
just for the minimum eigenvalue eigenvalue
```
Nonlinear Conjugate Gradient Methods for eigenvalue problems

\begin{Verbatim}
1    do i = 1, niter
2       do m = 1, blockSize
3       orthonormalize X(m) to X(1:m − 1)
4       ax = A X(m)
5       do j = 1, nline
6          \lambda(m) = X(m) \cdot ax
7          if (||ax − \lambda(m) X(m)||_2 < tol .or. j == nline) exit
8          r_{j+1} = (I − X X^T) ax
9          \beta = \frac{(r_{j+1}−r_j) \cdot \text{pr}_{j+1}}{r_j \cdot \text{pr}_j}
10         d_{j+1} = −P r_{j+1} + \beta d_j
11         d_{j+1} = (I − XX^T)d_{j+1}
12         \gamma = ||d_{j+1}||_2^{-1}
13         \theta = 0.5 |\text{atan} \frac{2 γ d_{j+1} \cdot ax}{\lambda(m) − γ^2 d_{j+1} \cdot A d_{j+1}} |
14         X(m) = \cos(\theta) X(m) + \sin(\theta) \gamma d_{j+1}
15         ax = \cos(\theta) ax + \sin(\theta) \gamma A d_{j+1}
16       enddo
17    enddo
18  [X, \lambda] = Rayleigh−Ritz on span\{X\}
\end{Verbatim}
LOBPCG algorithm

1: Start with a random \( X_0 \)
2: \( R_0 = AX_0 - X_0 \Theta_0 \)
3: \textbf{for} \( k = 0, n_{\text{max}} \) \textbf{do}
4: \quad \text{check if needed deflate in the vector } R_k \text{ and } P_k
5: \quad \text{check for convergence}
6: \quad H_k = \text{precond}(R_k)
7: \quad (Y_{k+1}, \Theta_{k+1}) = \text{Raleigh-Ritz}([X_k, H_k, P_k])
8: \quad X_{k+1} = [X_k, H_k, P_k]Y_{k+1}
9: \quad P_{k+1} = [0, H_k, P_k]Y_{k+1}
10: \quad R_{k+1} = AX_{k+1} - X_{k+1} \Theta_{k+1}
11: \textbf{end for}

(formulation inspired from Arbenz et al 2005)
LOBPCG algorithm

Start with a random $X_0$

$R_0 = AX_0 - X_0\Theta_0$

for $k = 0, n_{\text{max}}$ do

check if needed deflate in the vector $R_k$ and $P_k$

check for convergence

$H_k = \text{precond}(R_k)$

$(Y_{k+1}, \Theta_{k+1}) = \text{Raleigh-Ritz}([X_k, H_k, P_k])$

$X_{k+1} = [X_k, H_k, P_k]Y_{k+1}$

$P_{k+1} = [0, H_k, P_k]Y_{k+1}$

$R_{k+1} = AX_{k+1} - X_{k+1}\Theta_{k+1}$

end for

Lots of trick in a practical implementations: no orthogonalization of $X$, $R$ and $P$; restarting; deflation
Nonlinear Conjugate Gradient Methods for eigenvalue problems

1. \( \text{do } i = 1, \text{niter} \)
2. \( \text{do } m = 1, \text{blockSize} \)
3. \( \text{orthonormalize } X(m) \text{ to } X(1:m-1) \)
4. \( \text{ax} = A \cdot X(m) \)
5. \( \text{do } j = 1, \text{nline} \)
6. \( \lambda(m) = X(m) \cdot \text{ax} \)
7. \( \text{if } (||\text{ax} - \lambda(m) \cdot X(m)||_2 < \text{tol} \text{ or } j == \text{nline}) \text{ exit} \)
8. \( r_{j+1} = (I - XX^T) \cdot \text{ax} \)
9. \( \beta = \frac{(r_{j+1} - r_j) \cdot Pr_{j+1}}{r_j \cdot Pr_j} \)
10. \( d_{j+1} = -P \cdot r_{j+1} + \beta \cdot d_j \)
11. \( d_{j+1} = (I - XX^T) \cdot d_{j+1} \)
12. \( \gamma = ||d_{j+1}||^{-1}_2 \)
13. \( \theta = 0.5 \cdot |\text{atan} \frac{2 \gamma \cdot d_{j+1} \cdot \text{ax}}{\lambda(m) - \gamma^2 \cdot d_{j+1}^T \cdot A \cdot d_{j+1}}| \)
14. \( X(m) = \cos(\theta) \cdot X(m) + \sin(\theta) \cdot \gamma \cdot d_{j+1} \)
15. \( \text{ax} = \cos(\theta) \cdot \text{ax} + \sin(\theta) \cdot \gamma \cdot A \cdot d_{j+1} \)
16. \( \text{enddo} \)
17. \( \text{enddo} \)
18. \([X, \ \lambda] = \text{Rayleigh-Ritz on span}\{X\}\)
Nonlinear Conjugate Gradient Methods for eigenvalue problems

```plaintext
1  do i = 1, niter
2    do m = 1, blockSize
3      orthonormalize X(m) to X(1:m−1)
4      ax = A X(m)
5      do j = 1, nline
6        λ(m) = X(m) · ax
7        if (||ax − λ(m) X(m)||_2 < tol .or. j == nline) exit
8          r_{j+1} = (I − X X^T) ax
9          β = \frac{(r_{j+1}−r_j)·pr_{j+1}}{r_j·pr_j}
10         d_{j+1} = −P r_{j+1} + β d_j
11        d_{j+1} = (I − XX^T)d_{j+1}
12        γ = ||d_{j+1}||_2^{-1}
13        θ = 0.5 |\text{atan} \frac{2 γ d_{j+1}·ax}{λ(m)−γ^2 d_{j+1}·A d_{j+1}}|
14        X(m) = \cos(θ) X(m) + \sin(θ) γ d_{j+1}
15        ax = \cos(θ) ax + \sin(θ) γ A d_{j+1}
16      enddo
17  enddo
18  [X, λ] = Rayleigh−Ritz on span\{X\}
```
variants

PCG

1    do i = 1,niter
2        [X] = CG_inner(X)
18       [X, \lambda] = Rayleigh–Ritz on span\{X\}
19        enddo
variants

**PCG**

```
1     do i = 1, niter
2         [X] = CG_inner(X)
18       [X, λ] = Rayleigh – Ritz on span{X}
19     enddo
```

**PCG-XR**

```
1     do i = 1, niter
2         [X] = CG_inner(X)
18       [X, λ] = Rayleigh – Ritz on span{X,R}
19     enddo
```

**PCG-XRP**

```
1     do i = 1, niter
2         [X] = CG_inner(X)
18       [X, λ] = Rayleigh – Ritz on span{X,R,X_{−1}}
19     enddo
```
Insertion of the folded spectrum in LOBPCG

Notation: \( A = (H - E_{ref}^2) \)
Instead of having 6 blocks of vectors

\[
[X, R, P] \quad \text{and} \quad [AX, AP, AR]
\]

we added three more

\[
[HX, HR, HP]
\]

to be able to:

- control the convergence of the residuals on \( H \) (not \( A \))
- deflate accordingly to \( H \) (not \( A \))
Storage and vector operations (dot and axpy) comparisons

- PCG: \( m + 4 \)
- PCG-RX: \( 2m + 4 \)
- LOBPCG: \( 9m \)
Numerical experiments
5-point operator

On a 5-point operator with the coefficients $a = 8$ (diagonal) and $b = -1 - i$ (for the connections with the 4 closest neighbors on a regular 2D mesh). Parameter: Linux Intel Pentium IV with Intel Fortran compiler; 10 smallest eigenstates; matrix size is 20,000; $\text{tol} = 10^{-8}$.

<table>
<thead>
<tr>
<th>Method</th>
<th>PCG</th>
<th>LOBPCG ²</th>
<th>PCG-XR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>time (s)</strong></td>
<td>37.1</td>
<td>61.7</td>
<td>20.2</td>
</tr>
<tr>
<td><strong>matvecs</strong></td>
<td>3,555</td>
<td>1,679</td>
<td>1,760</td>
</tr>
<tr>
<td><strong>dotprds</strong></td>
<td>68,245</td>
<td>137,400</td>
<td>37,248</td>
</tr>
<tr>
<td><strong>axpys</strong></td>
<td>66,340</td>
<td>158,261</td>
<td>36,608</td>
</tr>
<tr>
<td><strong>copys</strong></td>
<td>6,190</td>
<td>9,976</td>
<td>3,560</td>
</tr>
</tbody>
</table>

Comparison of the PCG, PCG-XR and LOBPCG methods in finding 10 eigenstates on a problem of size $20,000 \times 20,000$.

²http://www-math.cudenver.edu/~aknyazev/software/CG/latest/lobpcg.m (revision 4.10 written in Matlab, with some slight modifications)
Comparison of

1. LOBPCG algorithm
2. PCG
3. PCG-XR

with Folded Spectrum
with diagonal preconditioner
with matrix-vector product via FFTs ⇒ No BLOCKING

With

1. IBM-SP seaborg at NERSC in Lawrence Berkeley National Laboratory.
2. $mx = 10$ interior eigenvalues around $E_{ref} = -4.8\text{eV}$
3. $tol = 10^{-6}$
4. 16 processors (except for the smallest case (20Cd,19Se) which is run on 8 processors)
A notable fact is that all the solvers find the same 10 eigenvalues with the correct accuracy for all the runs in less than 30 minutes. Therefore they are all robust.
Symmetric Eigenvalue Problem

**(83Cd, 81Se)** \( n = 34,143 \)

<table>
<thead>
<tr>
<th>Method</th>
<th>Precond</th>
<th># matvec</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(200)</td>
<td>no</td>
<td>7,176</td>
<td>1274 s</td>
</tr>
<tr>
<td>FS-LOBPCG</td>
<td>no</td>
<td>17,810</td>
<td>341 s</td>
</tr>
</tbody>
</table>

**Table:** Comparison of FS-PCG and FS-LOBPCG with and without preconditioner to find \( m_x = 10 \) eigenvalues of the quantum dots (83Cd,81Se)
Table: Comparison of FS-PCG and FS-LOBPCG with and without preconditioner to find $m_x = 10$ eigenvalues of the quantum dots $(83\text{Cd}, 81\text{Se})$
Comparison of FS-PCG, FS-PCG-XR and FS-LOBPCG methods in finding 10 eigenstates around the gap of quantum dots of increasing size.

<table>
<thead>
<tr>
<th>(20Cd, 19Se) n = 11,331</th>
<th># matvec</th>
<th># outer it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(50)</td>
<td>4898</td>
<td>(8)</td>
<td>50.4s</td>
</tr>
<tr>
<td>FS-PCG-XR(50)</td>
<td>4740</td>
<td>(6)</td>
<td>49.1s</td>
</tr>
<tr>
<td>FS-LOBPCG</td>
<td>4576</td>
<td></td>
<td>52.0s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(83Cd, 81Se) n = 34,143</th>
<th># matvec</th>
<th># outer it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(200)</td>
<td>15096</td>
<td>(11)</td>
<td>264 s</td>
</tr>
<tr>
<td>FS-PCG-XR(200)</td>
<td>12174</td>
<td>(5)</td>
<td>209 s</td>
</tr>
<tr>
<td>FS-LOBPCG</td>
<td>10688</td>
<td></td>
<td>210 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(232Cd, 235Se) n = 75,645</th>
<th># matvec</th>
<th># outer it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(200)</td>
<td>15754</td>
<td>(8)</td>
<td>513 s</td>
</tr>
<tr>
<td>FS-PCG-XR(200)</td>
<td>15716</td>
<td>(6)</td>
<td>508 s</td>
</tr>
<tr>
<td>FS-LOBPCG</td>
<td>11864</td>
<td></td>
<td>458 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(534Cd, 527Se) n = 141,625</th>
<th># matvec</th>
<th># outer it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(500)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS-PCG-XR(500)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS-LOBPCG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(232Cd, 235Se) n = 75,645

<table>
<thead>
<tr>
<th>Method</th>
<th># matvec</th>
<th># outer it</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS-PCG(100)</td>
<td>17062</td>
<td>(15)</td>
<td>577s</td>
</tr>
<tr>
<td>FS-PCG(200)</td>
<td>15716</td>
<td>(6)</td>
<td>508s</td>
</tr>
<tr>
<td>FS-PCG(300)</td>
<td>15990</td>
<td>(4)</td>
<td>517s</td>
</tr>
<tr>
<td>FS-PCG($10^{-1}$)</td>
<td>15076</td>
<td>(6)</td>
<td>497s</td>
</tr>
</tbody>
</table>

The problem of finding the best inner length for FS-PCG can be avoided by fixing a tolerance as stopping criterion in the inner loop.
Conclusions

www.cs.utk.edu/~langou/articles/nonlinearCG-for-nano.pdf

- not fair with LOBPCG (no blocking, no efficient preconditioner)
- still LOBPCG is a good candidate, but basically all the methods (PCG, PCG-XR, LOBPCG) are in the same order of time
- regarding the number of vector stored FS-PCG is really impressive \((m + 4)\)
Nonsymmetric Eigenvalue problem