1 Introduction
   - Motivation for sparse linear algebra
   - Brief overview of sparse direct methods
   - Motivation for iterative methods
   - Short pro/con Iterative/Direct

2 Sparse Matrix Storage and Matrix-Vector product

3 Iterative Methods for Linear Systems
   - Stationary Iterative Methods
   - Krylov Iterative Methods

4 Iterative Methods for Eigenproblems

5 Various stuff
Motivation for sparse linear algebra: examples with PDE’s
Poisson equation in 2D

Let us consider the Poisson equation:

\[
\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = \Delta u = f, \quad \text{for } x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \text{ in } \Omega
\]

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

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

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

(\( \phi(x_1, x_2) \) and \( f(x_1, x_2) \) are known, the unknown is \( u(x_1, x_2) \).)
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) + \mathcal{O}(h^2) \\
  u(x - h) &= u(x) - h \frac{du}{dx}(x) + \frac{h^2}{2} \frac{d^2 u}{dx^2}(x) + \mathcal{O}(h^2)
\end{align*}

$$\Rightarrow \frac{d^2 u}{dx^2}(x) = \frac{1}{h^2} (u(x + h) + u(x - h) - 2u(x)) + \mathcal{O}(1)$$
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)

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

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

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

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

and the Dirichlet boundary condition

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

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

\[
\begin{array}{c}
\times_0 \quad \times_1 \quad \cdots \quad \times_n \quad \times_{n+1} \\
h \quad h \quad \cdots \quad h \quad h
\end{array}
\]

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

\[
A = \frac{1}{h^2} \begin{pmatrix}
2 & -1 & & & \\
-1 & 2 & -1 & & \\
 & -1 & 2 & -1 & \\
& & \ddots & \ddots & \ddots \\
& & & -1 & 2 & -1 \\
& & & & -1 & 2
\end{pmatrix}
\]
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.

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

where

$$B = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \\ \vdots & \vdots & \ddots & \ddots \\ -1 & -1 & 4 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \\ \end{pmatrix}$$
Motivation for sparse linear algebra: examples with PDE’s

Introduction

Motivation for sparse linear algebra

Collection of Sparse matrices

- sparse matrices exist
- SPD matrices exist

( in real life application code!)

See also

- other equations: convection-diffusion, ...
- other discretizations: finite element method, finite volume method,…
- various matrix collections:
  - Matrix Market [http://math.nist.gov/MatrixMarket/]
  - University of Florida Sparse Matrix Collection [http://www.cise.ufl.edu/research/sparse/matrices/]
Brief overview of sparse direct methods
Definition for sparse matrices

What is a sparse matrix?

- a sparse matrix is a matrix that has sufficiently large number of zero elements that a specialized storage scheme is warranted
- all storage format that is not in LAPACK (no tridiag, banded, packed)
- see the definition of storage format given by the BLAS-T forum [http://www.netlib.org/blas/blast-forum/](http://www.netlib.org/blas/blast-forum/)
- ...

...
In Dense Linear Algebra, the Gaussian elimination of a $n$-by-$n$ matrix requires

1. requires $O(n^2)$ storage,
2. requires $O(n^3)$ flops,
3. and runs at a reasonable fraction of the machine peak performance.

What can we do when the matrix is sparse with $nnz\_per\_row$ elements per row ($nnz\_per\_row \ll n$)? Can we do better than a dense solve?

Answer: Yes but it is not that easy.

Storage in $O(nnz)$ is easy (see Storage).

Major difficulties:

1. fill-in occurs (the number of flops can increase dramatically),
2. performance are awful.
Apply the Gaussian elimination without pivoting on the matrix on the right. We are just interested in the symbolic LU factorization, the \( \bullet \) stands for any stored values in \( A \), gives the stored values of \( L \) and \( U \).

What do you observe?

1: for \( k = 1 : n - 1 \) do
2: \[ a_{k+1:n,k} = a_{k+1:n,k} / a_{kk} \]
3: for \( i = k + 1 : n \) do
4: for \( j = k + 1 : n \) do
5: \[ a_{ij} = a_{ij} - a_{i,k} a_{kj} \]
6: end for
7: end for
8: end for
Apply the Gaussian elimination without pivoting on the matrix on the right.

nnz(A) = 18 nnz(LU) =

Algorithm LU – \((j = 1, i = 1)\)
Fill-in example

Apply the Gaussian elimination without pivoting on the matrix on the right.

\[ \text{nnz}(A) = 18 \quad \text{nnz}(LU) = \]

Algorithm LU – \((j = 1, i = 2)\)

```plaintext
1: for \(k = 1 : n - 1\) do
2: \(a_{k+1:n,k} = a_{k+1:n,k} / a_{kk}\)
3: for \(i = k + 1 : n\) do
4: \(\text{for } j = k + 1 : n\) do
5: \(a_{ij} = a_{ij} - a_{i,k} a_{kj}\)
6: \(\text{end for}\)
7: \(\text{end for}\)
8: \(\text{end for}\)
```
Apply the Gaussian elimination without pivoting on the matrix on the right.

nnz(A) = 18 nnz(LU) =

Algorithm LU – $(j = 2, i = \cdot)$

1: for $k = 1 : n - 1$ do
2:   $a_{k+1:n,k} = a_{k+1:n,k} / a_{kk}$
3: for $i = k + 1 : n$ do
4:   for $j = k + 1 : n$ do
5:     $a_{ij} = a_{ij} - a_{i,k} a_{kj}$
6:   end for
7: end for
8: end for
Fill-in example

Apply the Gaussian elimination without pivoting on the matrix on the right.

\[ \text{nnz}(A) = 18 \quad \text{nnz}(LU) = \]

1: for \( k = 1 : n - 1 \) do
2: \[ a_{k+1:n,k} = a_{k+1:n,k} / a_{kk} \]
3: for \( i = k + 1 : n \) do
4: for \( j = k + 1 : n \) do
5: \[ a_{ij} = a_{ij} - a_{i,k} a_{kj} \]
6: end for
7: end for
8: end for

Algorithm LU \((j = 4, i = \cdot)\)
Apply the Gaussian elimination without pivoting on the matrix on the right.

\[ \text{nnz}(A) = 18 \quad \text{nnz}(LU) = 24 \]
Fill-in example

This matrix is certainly the worst example for fill-in. What is the cure?
Fill-in example

This matrix is certainly the worst example for fill-in.

Apply the reordering columns and row:

\[ [10, 9, 8, \ldots, 1] \]

Full fill-in (without permutation) against no fill-in (with permutation).
In dense linear algebra, what is the goal of the permutation in the Gauss method (see Jack class)? Note as well that we do not care about the values of the entries during our previous analysis (just $\cdot$s and $\cdot$). What do you think about this? (two remarks)

A priori permutation are done for stability purpose not for minimizing the fill-in. Therefore we see that sparse direct method have to do a trade-off between stability and small fill-in. Note as well that the symbolic factorization is extremely fast (just 0 and 1).

Finally this trade-off does not exist in the Cholesky factorization for SPD matrices since pivoting is not neccessary for stability (providing the matrix is enough 'SPD').
Brief overview of sparse direct methods

Introduction

Brief overview of sparse direct methods

Fill-in example – symmetrized SHERMAN5 – $n = 3,312$ – nnz=20,793 – Symb. Cholesky

<table>
<thead>
<tr>
<th>Permutation</th>
<th>nnz in $R$</th>
<th>fill-in in $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>1,188,956</td>
<td>114.4</td>
</tr>
<tr>
<td>Symmetric approximate minimum degree</td>
<td>157,630</td>
<td>15.2</td>
</tr>
<tr>
<td>Symmetric reverse Cuthill-McKee</td>
<td>188,658</td>
<td>18.1</td>
</tr>
</tbody>
</table>

permutation

original

symamd

symrcm

$A$

$R$
Fill-in example – BCSPWR10 – $n = 5,300$ – nnz=21,842 – SPD – Cholesky factorization

<table>
<thead>
<tr>
<th>Permutation</th>
<th>nnz in $R$</th>
<th>fill-in in $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>51,312</td>
<td>4.7</td>
</tr>
<tr>
<td>Symmetric approximate minimum degree</td>
<td>50,722</td>
<td>4.6</td>
</tr>
<tr>
<td>Symmetric reverse Cuthill-McKee</td>
<td>292,952</td>
<td>26.8</td>
</tr>
</tbody>
</table>
Efficiency problem in the sparse operation

See Section on Sparse Matrix-vector product, the same problems arise and the same cures apply.
### Software available

[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></td>
<td></td>
<td>Real</td>
<td>Complex</td>
</tr>
<tr>
<td>DSCPACK</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></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>X</td>
<td></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>X</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></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>X</td>
<td></td>
</tr>
</tbody>
</table>
To conclude this section, when the matrix is reasonably sparse \((\text{nnz\_per\_row} \ll n)\) then sparse direct iterative methods offer a viable alternative to dense direct methods in term of

- storage
- flops
- speed

(and definitely not in term of performance rate)
Motivation for iterative methods: Jacobi Method
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\(^1\) \(x^{(1)}\):

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

\(^1\)we assume \(a_{ii} \neq 0\)
Motivation for iterative methods: Jacobi Method

Introduction

Motivation for 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>...</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>
Motivation for iterative methods: Jacobi Method

Introduction

Motivation for iterative methods

At each step, we are performing:

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

The number of Flops needed per iteration is $2n^2$ Flops

The number of Flops needed to converge is $nb\_iter \times 2n^2 = nb\_iter \times Flops_{matvec}$ 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

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

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

Note that we just need to know the diagonal of \( A \) and then access the matrix-vector product
Motivation for iterative methods: Jacobi Method

Gauss-Seidel method

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

Introduction

Motivation for iterative methods

Jacobi Method:

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

Gauss-Seidel method

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

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 \( x_k^{(i+1)} \) implies the knowledge of \( x_k^{(i+1)} \). Parallelisation is impossible.
Gauss-Seidel and Jacobi methods:

- 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*.
Introduction

Motivation for iterative methods

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
  - among iterative methods, trade-off between time per iteration and efficiency of an iteration

First conclusion that holds for any stationary iterative method

+ extremely robust to errasure/error during the computation

But what about convergence???? Is there any theorem?

see Section Stationary Iterative Methods
Motivation for iterative methods: an argumentation (H. van der Vorst)
discretization of a second order PDE in 3D, finite element discretization over an irregular grid with about $m$ unknowns per dimension

**Sparse Direct method (roughly what happens)**
- $n = m^3$
- bandwidth $= m^2 = n^{2/3}$
- Flops for LU $\sim n^{7/3}$

**Sparse Iterative Methods (roughly what happens)**
- we want to use the formula: $\# \text{ Flops} = (\# \text{ Flops/iteration}) \times \# \text{ of iteration}$
- let assume that $(\# \text{ Flops/iteration}) = fn$
- error per reduction step of CG is $\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$ (where $\kappa = ||A||_2 ||A^{-1}||_2$)
- and $\kappa \sim m^2$
- thus you can guess the number of iteration ...
- $\# \text{ Flops for CG} \sim fn^{4/3} \log \epsilon$

(caution: this study does not speak about performance just $\# \text{ Flops}$)

Short pro/con Iterative/Direct
The AIJ storage format

For example, the AIJ storage format is like:

\[
\begin{pmatrix}
\bullet & \bullet & \bullet & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \bullet & \bullet & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{pmatrix}
\]

\[
\begin{array}{ccc}
1 & 1 & \bullet \\
1 & 2 & \bullet \\
1 & 4 & \bullet \\
2 & 1 & \bullet \\
2 & 2 & \bullet \\
2 & 5 & \bullet \\
3 & 3 & \bullet \\
4 & 4 & \bullet \\
4 & 7 & \bullet \\
\vdots & \vdots & \vdots \\
\end{array}
\]

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

For example, the CSR storage format is like:

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

It requires to store \(2\text{nnz} + n\) numbers (float and integer).
The ITPACK (or Purdue) storage format

\[
\begin{pmatrix}
1 & 3 & 6 & . & . & . & . & . & . \\
2 & 1 & . & 9 & . & . & . & . & . \\
. & . & 2 & . & . & . & . & . & . \\
. & . & 8 & . & 4 & . & . & . & . \\
. & 4 & . & 1 & . & . & . & . & . \\
. & . & . & 3 & . & . & . & . & . \\
. & 2 & . & . & . & 7 & . & . & . \\
. & . & . & . & . & 6 & . & . & . \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 3 & 6 \\
2 & 1 & 9 \\
2 & 0 & 0 \\
8 & 4 & 0 \\
4 & 1 & 0 \\
3 & 0 & 0 \\
2 & 7 & 0 \\
6 & 0 & 0 \\
2 & 0 & 0 \\
3 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 2 & 4 \\
1 & 2 & 5 \\
3 & 0 & 0 \\
4 & 7 & 0 \\
2 & 5 & 0 \\
6 & 0 & 0 \\
2 & 7 & 0 \\
8 & 0 & 0 \\
0 & 0 & 0 \\
4 & 10 & 0
\end{pmatrix}
\]

\[\text{val} = \begin{pmatrix} 1 & 3 & 6 \\ 2 & 1 & 9 \\ 2 & 0 & 0 \\ 8 & 4 & 0 \\ 4 & 1 & 0 \\ 3 & 0 & 0 \\ 2 & 7 & 0 \\ 6 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 1 & 0 \end{pmatrix} \quad \text{col\_ind} = \begin{pmatrix} 1 & 2 & 4 \\ 1 & 2 & 5 \\ 3 & 0 & 0 \\ 4 & 7 & 0 \\ 2 & 5 & 0 \\ 6 & 0 & 0 \\ 2 & 7 & 0 \\ 8 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 10 & 0 \end{pmatrix}\]
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)

The Compressed Sparse Row (CSR) matrix-vector product

<table>
<thead>
<tr>
<th>val</th>
<th>(a_{11})</th>
<th>(a_{12})</th>
<th>(a_{14})</th>
<th>(a_{21})</th>
<th>(a_{23})</th>
<th>(a_{25})</th>
<th>(a_{33})</th>
<th>(a_{44})</th>
<th>(a_{47})</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 Itaniumum2 machine (results from Haihang and Keith):

<table>
<thead>
<tr>
<th></th>
<th>MFlops</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Peak</td>
<td>3600.0</td>
<td>(100.0)</td>
</tr>
<tr>
<td>GEMM (ATLAS)</td>
<td>2188.1</td>
<td>(60.8)</td>
</tr>
<tr>
<td>GEMV (ATLAS)</td>
<td>1222.8</td>
<td>(33.4)</td>
</tr>
<tr>
<td>CSR gcc-4.3.1</td>
<td>26.6</td>
<td>(0.7)</td>
</tr>
<tr>
<td>CSR icc-8</td>
<td>80.0</td>
<td>(2.2)</td>
</tr>
</tbody>
</table>

```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;
}
... 
```
The Compressed Sparse Row (CSR) matrix-vector product

\[
\begin{array}{cccccccccc}
& a_{11} & a_{12} & a_{14} & a_{21} & a_{23} & a_{25} & a_{33} & a_{44} & a_{47} \\
\text{val} & & & & & & & & & \\
\text{col\_ind} & 1 & 2 & 4 & 1 & 2 & 5 & 3 & 4 & 7 \\
\text{row\_ptr} & 1 & 4 & 7 & 8 \\
\end{array}
\]

... 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

<table>
<thead>
<tr>
<th>In sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>- blocking by explicit fill-in</td>
</tr>
<tr>
<td>- technique ala ATLAS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In parallel (minimizing the communication)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ghostpoint</td>
</tr>
<tr>
<td>- solution partitioning (reordering, see Stanimire results)</td>
</tr>
</tbody>
</table>
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>MTL</td>
<td>yes</td>
<td>X</td>
<td>f77</td>
<td>x</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NETMAT</td>
<td>yes</td>
<td>X</td>
<td>c</td>
<td>x</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NIST S-BLAS</td>
<td>yes</td>
<td>X</td>
<td>c++</td>
<td>x</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PSBLAS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>SparseLib++</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>uBLAS</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Stationary Iterative Methods
Stationary Iterative Methods

Iterative Methods for Linear Systems

Stationary Iterative Methods

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

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 $\rho(D^{-1} A) < 1$, then the Jacobi method converges linearly.

link with the diagonal dominancy of $A$
Krylov iterative methods
The Krylov $\mathcal{K}(A, r_0, i)$ space is the space

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

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

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

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

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 \( R_i \).
GCR: Generalized Conjugate Residuals

find an approximate solution $x_{i+1}$

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

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

which is equivalent to impose the Petrov-Galerkin condition

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

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 \( C'_i \)

\[ C_i = C'_i \times H, \text{ where } H \text{ upper triangular and } C_i^T C_i = I \]

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

\[ AR_i = C'_i \]

\[ AU_i = C_i \]

(by imposing \( U_i = C'_i H \))
GCR: Generalized Conjugate Residuals

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: $c_{i+1}' = Au_i$
7: orthonormalize $c_{i+1}'$ 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: $\alpha = c_{i+1}'r_i$
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 $i = 1 \text{ to } k$ 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 $\min \|\beta e_1 - \bar{H}_k y\|_2$ for $y_k$
14: if $\|\tilde{r}_k\|_2 = \|\beta e_1 - \bar{H}_k y_k\|_2 \leq \epsilon \|A\|_2 \|x_k\|_2$ 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

Iterative Methods for Linear Systems

Krylov Iterative Methods

\[
A = \begin{pmatrix}
10 & 1 & 2 & 0 & 1 \\
0 & 12 & 1 & 3 & 1 \\
1 & 2 & 9 & 1 & 0 \\
0 & 3 & 1 & 10 & 0 \\
1 & 2 & 0 & 0 & 15 \\
\end{pmatrix}
\]

\[
b = \begin{pmatrix}
23 \\
44 \\
36 \\
49 \\
80 \\
\end{pmatrix}
\]

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

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></td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complex</td>
<td>f77</td>
<td>c</td>
</tr>
<tr>
<td>(B/H)LZPACK</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HYPRE</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>QMRPACK</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LASO</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>P_ARPACK</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PLANSO</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SLEPC</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SPAM</td>
<td>yes</td>
<td>X</td>
<td>90</td>
<td>X</td>
</tr>
<tr>
<td>TRLAN</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Various stuff
How to choose a sparse iterative solver

1. choose a method adapted to your problem (see the flowchart in the Templates)
2. check the difficulty to interface it with your code (and other software you might to use), three model
   1. PETSc-default, your library is in PETSc style
   2. the library calls given routines for the Kernel operations
   3. reverse communication (CERFACS, ARPACK)
3. check for nice interoperability feature with other package (for example PETSc has a lots of interface, HYPRE as well)
4. check the quality of the software
5. check the compatibility of language
Building Blocks for Iterative Methods

- Most iterative methods made abstraction of the data and manipulate it use with very few abstract routines like:
  - AXPY $y \leftarrow \alpha x + y$
  - DOT $\alpha \leftarrow \langle x, y \rangle = x^T y$
  - MATVEC $y \leftarrow Ax$
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: \hspace{1em} $z = Av_k$
7: for $i = 1$ to $k$ do
8: \hspace{2em} $h_{i,k} = v_i^T z$
9: \hspace{2em} $z = z - h_{i,k} v_i$
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 $\min \|\beta e_1 - H_k y\|_2$ for $y_k$
14: if $\|r_k\|_2 = \|\beta e_1 - H_k y_k\|_2 \leq \epsilon \|A\|_2 \|x_k\|_2$ then
15: \hspace{1em} Set $x_k = x_0 + V_k y_k$
16: end if
17: end for
18: end for
call drive_zgmres(n,n,m,lwork,work,
   &       irc,icnt1,cnt1,info,rinfo)
revcom = irc(1)
colx = irc(2)
coly = irc(3)
colz = irc(4)
nbscal = irc(5)

* if (revcom.eq.matvec) then
* perform the matrix vector product
* work(colz) <-- A * work(colx)
call zgemv('N',n,n,ONE,a,lda,work(colx),1,
   &       ZERO,work(colz),1)
goto 10
* else if (revcom.eq.precondLeft) then
* perform the left preconditioning
* work(colz) <-- M^{-1} * work(colx)
call zcopy(n,work(colx),1,work(colz),1)
goto 10
* else if (revcom.eq.precondRight) then
* perform the right preconditioning
call zcopy(n,work(colx),1,work(colz),1)
goto 10
* else if (revcom.eq.dotProd) then
* perform the scalar product
* work(colz) <-- work(colx) work(coly)
* call zgemv('C',n,nbscal,ONE,work(colx),n,
   &       work(coly),1,ZERO,work(colz),1)
goto 10
endif
1 Yousef Saad. *Iterative Methods for Sparse Linear Systems.*

2 Yousef Saad. *Iterative Methods for Large Eigenvalue Problems.*

3 Jack Dongarra. *Jack’s list of freely available software for linear algebra on the web.*
   - [http://www.netlib.org/utk/people/JackDongarra/la-sw.html](http://www.netlib.org/utk/people/JackDongarra/la-sw.html)