Randomized Preconditioning

Haim Avron (IBM Research)
Based on joint work with Petar Maymounkov and Sivan Toledo

FOCS’12 RandNLA Workshop, October 2012
Outline

- Randomized Preconditioning - why is it a good idea?
- Fast linear regression for dense matrices
- A little on structured sparse (finite element) matrices
High accuracy is expected.

Most NLA methods are inexact (floating point computation), but high accuracy is expected (think $\epsilon = 10^{-8}$).

Potential failures are a problem.

Even a one-in-a-million probability is not enough: millions of linear systems are solved every day.

Highly optimized and constantly improved libraries. Very hard to beat them.

LAPACK is 20 years old, but last updated in April 2012.

This talk: how to address these issues for linear solvers.
Concrete example: \( \min_x \|Ax - b\|_2 \), \( A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m \)

```matlab
>> rand('state', 2378)
>> randn('state', 23984)
>> m = 20000; n = 100;
>> A = rand(m, n); b = rand(m, 1);
>> [U, S, V] = svd(A, 0);
>> S = diag(linspace(1, 10^6, 100));
>> A = U * S * V'; x = A \ b;
```
Concrete example: \( \min_x \|Ax - b\|_2, \ A \in \mathbb{R}^{m \times n}, \ b \in \mathbb{R}^m \)

Recall:

- Can be solved by \( A^T Ax = A^T b \) (not numerically stable)
- Alternatively: factor \( A = QR \) and solve \( Rx = Q^T b \).
- Both options require \( O(mn^2) \) operations (with classical matrix-matrix multiplication).

```matlab
>> rand('state', 2378)
>> randn('state', 23984)
>> m = 20000; n = 100;
>> A = rand(m, n); b = rand(m, 1);
>> [U, S, V] = svd(A, 0);
>> S = diag(linspace(1, 10^6, 100));
>> A = U * S * V'; x = A \ b;
```
Concrete example: $\min_x \|Ax - b\|_2$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$

```matlab
>> rand('state', 2378)
>> randn('state', 23984)
>> m = 20000; n = 100;
>> A = rand(m, n); b = rand(m, 1);
>> [U, S, V] = svd(A, 0);
>> S = diag(linspace(1, 10^6, 100));
>> A = U * S * V'; x = A \ b;
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
   1.0107
A random sampling solver

>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
   1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

\begin{verbatim}
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
\end{verbatim}
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
 1.0107
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
1.0107
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
A random sampling solver

>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
  1.0107

We’re getting something, but what exactly?
Analysis of the algorithm

\((m - \text{number of rows, } n - \text{number of columns})\)

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” A’s, \(\Omega(\varepsilon^{-1} \cdot n \log n)\) samples suffice for

\[
\|Ax_{\text{alg}} - b\|_2 \leq (1 + \varepsilon)\|Ax_{\text{opt}} - b\|_2
\]

with high probability.

1

2
Analysis of the algorithm

($m$ - number of rows, $n$ - number of columns)

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” A’s, $\Omega(\epsilon^{-1} \cdot n \log n)$ samples suffice for

$$\|Ax_{\text{alg}} - b\|_2 \leq (1 + \epsilon)\|Ax_{\text{opt}} - b\|_2$$

with high probability.

1. **Running time:** $O(\epsilon^{-1} \cdot n^3 \log n)$ (least squares of a $O(\epsilon^{-1} \cdot n \log n) \times n$ matrix).

2. \[ \]
Analysis of the algorithm

(m - number of rows, n - number of columns)

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” A’s, \( \Omega(\epsilon^{-1} \cdot n \log n) \) samples suffice for

\[
\|Ax_{\text{alg}} - b\|_2 \leq (1 + \epsilon)\|Ax_{\text{opt}} - b\|_2
\]

with high probability.

1. Running time: \( O(\epsilon^{-1} \cdot n^3 \log n) \) (least squares of a \( O(\epsilon^{-1} \cdot n \log n) \times n \) matrix).

2. Monte-Carlo style - deterministic running time, but only a probabilistic guarantee on the error.
Assessment

- Dependence on $m$ and $n$: if $m \gg n$ we gained something here.
- Dependence on $\epsilon$: $O(\epsilon^{-1})$
  - Application dependent, some allow not-too-small $\epsilon$.
  - But for NLA kernel, $\epsilon$ is constant but tiny (think: $\epsilon = 10^{-8}$).
  - $O(\epsilon^{-1})$ is exponential in number of correct significant digits.
- Monte-Carlo style:
  - Larger sample or repeated application will introduce a factor of $O(\ln(1/\delta))$ ($\delta$ is the failure probability).
  - Failure should be extremely rare (think: library) - introduces a not-too-small constant.
**Main idea:** At iteration $i$ find best solution (in some norm) in a rank $i$ subspace.

To allow quick update, the Krylov subspaces are used:

$$K_i(M, d) = \text{span} \{ d, M d, \ldots, M^{i-1} b \}.$$

Many Krylov methods exist: CG, MINRES, GMRES, CGLS, ...

For solving $\min \|Ax - b\|_2$ LSQR is recommended.

$$x_i = \arg \min_{\tilde{x} \in K_i(A^T A, A^T b)} \|A\tilde{x} - b\|_2.$$

Each iteration requires one multiply of $A$ by vector, one by $A^T$, and $O(1)$ dot products.
Well known convergence bound for LSQR: $O(\kappa(A) \cdot \log(\epsilon^{-1}))$ iterations.

$$\kappa(A) = \frac{\text{ratio between extreme singular values}}{\max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} \cdot \max_{y \neq 0} \frac{\|A^{-1}y\|_2}{\|y\|_2}}$$

(assuming $A$ is full rank)

Preconditioning:

$$\min_x \|Ax - b\|_2 \quad \rightarrow \quad \min_y \|AR^{-1}y - b\|_2$$

Might be a good idea if $\kappa(AR^{-1}) < \kappa(A)$. 
Fine balance:
- Gain: fewer iterations
- Lose: build preconditioner, iterations are more expensive

Preconditioner should
- be easy to solve.
- reduce condition number substantially.
Back to example: how to use preconditioning?

```matlab
>> sampled_rows = randsample(m, ceil(10 * n * log(n)), true);
>> A1 = A(sampled_rows, :); b1 = b(sampled_rows);
>> x1 = A1 \ b1;
>> norm(A * x1 - b) / norm(A * x - b)
ans =
    1.0107
```
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

```matlab
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 17 to a solution with relative residual 0.5
```
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[
\begin{bmatrix}
Q & R
\end{bmatrix}
= \begin{bmatrix}
\end{bmatrix}
\]

\[
\text{>> [Q, R] = qr(A1, 0);}
\]
\[
\text{>> x1 = lsqr(A, b, eps, 100, R);}
\]
\[
\text{lsqr converged at iteration 17 to a solution with}
\]
\[
\text{relative residual 0.5}
\]
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 17 to a solution with relative residual 0.5
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[ x = \text{LSQR}(A, b, \quad ) \]

>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 17 to a solution with relative residual 0.5
Random sampling preconditioning

Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[ x = \text{LSQR}(A, b, \quad ) \]

\[
\begin{align*}
\text{>> } & [Q, R] = \text{qr}(A_1, 0); \\
\text{>> } & x_1 = \text{lsqr}(A, b, \text{eps}, 100, R); \\
\text{lsqr converged at iteration 17 to a solution with} \\
\text{relative residual 0.5}
\end{align*}
\]
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

```matlab
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
lsqr converged at iteration 17 to a solution with relative residual 0.5
```
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[
x = \text{LSQR}(A, b, )
\]

\[
>> [Q, R] = \text{qr}(A1, 0);
>> x1 = \text{lsqr}(A, b, \text{eps}, 100, R);
\]

lsqr converged at iteration 17 to a solution with relative residual 0.5
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[
x = \text{LSQR}(A, b, R)
\]

\[
\begin{align*}
\text{>> } [Q, R] &= \text{qr}(A1, 0); \\
\text{>> } x1 &= \text{lsqr}(A, b, \text{eps}, 100, R); \\
\text{lsqr converged at iteration 17 to a solution with relative residual 0.5}
\end{align*}
\]
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[ x = \text{LSQR}(A, b, R) \]

\[
\begin{align*}
\text{>> } & [Q, R] = \text{qr}(A1, 0); \\
\text{>> } & x_1 = \text{lsqr}(A, b, \text{eps}, 100, R); \\
\text{lsqr} & \text{ converged at iteration 17 to a solution with} \\
\text{relative residual } & 0.5
\end{align*}
\]
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[
x = \text{LSQR}(A, b, \quad)
\]

\[
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
\]

lsqr converged at iteration 17 to a solution with
relative residual 0.5
Random sampling preconditioning
Rokhlin and Tygert (2008) PNAS; Avron, Maymounkov and Toledo (2010) SISC

\[ \mathbf{x} = \text{LSQR}(\mathbf{A}, \mathbf{b}, \mathbf{R}) \]

\[
\begin{align*}
\gg [Q, R] &= \text{qr}(A1, 0); \\
\gg x1 &= \text{lsqr}(A, b, \text{eps}, 100, R); \\
\text{lsqlr converged at iteration 17 to a solution with} \\
\text{relative residual 0.5}
\end{align*}
\]

A very accurate solution
Analysis of the algorithm

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” A’s, $\Omega(n \log n)$ samples suffice for

$$\kappa(AR^{-1}) = O(1)$$

with high probability, where $R$ is the $R$ factor of the sampled matrix.

1

2
Analysis of the algorithm

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” $A$’s, $\Omega(n \log n)$ samples suffice for

$$\kappa(AR^{-1}) = O(1)$$

with high probability, where $R$ is the $R$-factor of the sampled matrix.

1. Running time: $O(n^3 \log n + mn \log \epsilon^{-1})$ with high probability.
   1. Factorization of an $O(n \log n) \times n$ matrix
   2. $O(\log \epsilon^{-1})$ iterations of preconditioned LSQR

2. Deterministic guarantee on the error, but probabilistic running time.
Analysis of the algorithm

Using carefully chosen non-uniform sampling probabilities (leverage scores), or uniform sampling on “well-behaved” A’s, $\Omega(n \log n)$ samples suffice for

$$\kappa(AR^{-1}) = O(1)$$

with high probability, where $R$ is the $R$-factor of the sampled matrix.

1. **Running time:** $O(n^3 \log n + mn \log \epsilon^{-1})$ with high probability.
   - Factorization of an $O(n \log n) \times n$ matrix
   - $O(\log \epsilon^{-1})$ iterations of preconditioned LSQR

2. **Las Vegas style** - Deterministic guarantee on the error, but probabilistic running time.
### Why Preconditioning?

<table>
<thead>
<tr>
<th></th>
<th>Sample &amp; Solve</th>
<th>Sample &amp; Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running time</strong></td>
<td>$O(\epsilon^{-1} \cdot n^3 \log n)$</td>
<td>$O(n^3 \log(n) + mn \log \epsilon^{-1})$</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Probabilistic error (Monte Carlo style)</td>
<td>Probabilistic run-time (Las Vegas style)</td>
</tr>
</tbody>
</table>
**Ill-behaved matrices**

```matlab
>> A(1:end-1, end) = 0;
>> A(end, end) = 1;
>> A1 = A(sampled_rows, :);
>> [Q, R] = qr(A1, 0);
>> x1 = lsqr(A, b, eps, 100, R);
Warning: Matrix is singular to working precision.
> In sparfun/private/iterapp at 33
  In lsqr at 217 In overview at 35
lsqr stopped at iteration 0 without converging to the desired tolerance 2.2e-016 because the system involving the preconditioner was ill conditioned.
The iterate returned (number 0) has relative residual 1
```
Non-uniform sampling

$U$ - columns are an orthonormal basis of range($A$).

$\tau_r = \|U_{(r)}\|^2_2$ - leverage score of row $r$. Measure of $r$’s importance.

- Very useful in analyzing of randomized matrix algorithms
  (Mahoney & Drineas (2009) PNAS)

Carefully chosen probabilities: $p_r = \tau_r/n$.

Problem: computing them exactly is expensive ($O(mn^2)$)
... but fast approximation algorithms exists
(Drineas et al. (2012) ICML)
Coherence: $\mu(A) = \max_r \tau_r$

"incoherent" $n/m \leq \mu(A) \leq 1$ "coherent"

$\Omega(\mu(A)m\log(\mu(A)m))$ suffice for $\kappa = O(1)$ with high probability.
(Avron & Toledo (2011), Ipsen & Wentworth (2012) for more explicit bounds.)

$\mu(A) = n/m \rightarrow \Theta(n \log n)$ samples for $\kappa \leq 3$
$\mu(A) = 1 \rightarrow \Theta(m \log m)$ (all rows) samples for $\kappa \leq 3$
Reducing Coherence By “Row Mixing”

\[ B = F \ast D \ast A \]

- \( F \) is a Discrete Cosine Transform (DCT) matrix
  - Fast matrix-vector multiplication.
  - Small entries
- \( D \) is random diagonal with \( \pm 1 \) on diagonal.

\( B \) has low coherence (with high probability), but the same spectrum as \( A \).

Combined with sampling, this is “Fast J-L Transform”, Ailon & Chazelle (2006) SICOMP

Similar construction (with Walsh-Hadamard instead of DCT) was first used by Drineas et al. (2011) Numer. Math.
Algorithm

Mix: \[ A \rightarrow B = DCT \ast D_{\text{RAND}} \ast A \]

Sample: \[ B \rightarrow B_1 = SB \]

Factor: \[ B_1 \rightarrow R \text{ from } B_1 = QR \]

Solve: LSQR using \( R \) as a preconditioner

\[
\begin{align*}
&\text{>> } B1 = B(\text{sampled}_\text{rows}, :) ; \\
&\text{>> } [Q, R] = \text{qr}(B1, 0) ; \\
&\text{>> } x1 = \text{lsqr}(A, b, \text{eps}, 100, R) ; \\
&\text{lsqr converged at iteration 15 to a solution with} \\
&\text{relative residual 1}
\end{align*}
\]

Theorem:

\[
O \left( mn \log(m) + \left( \sqrt{n} + \sqrt{8 \log(m)} \right)^2 n^2 \log(n) + mn \log \epsilon^{-1} \right)
\]

with high probability

(using Boutsidis & Gittens (2012))
Algorithm

Mix: \( A \longrightarrow B = DCT \times D_{\text{RAND}} \times A \)

Sample: \( B \longrightarrow B_1 = SB \)

Factor: \( B_1 \longrightarrow R \) from \( B_1 = QR \)

Solve: LSQR using \( R \) as a preconditioner

\[
\begin{align*}
\text{>> } B1 &= B(\text{sampled\_rows},:) ; \\
\text{>> } [Q, R] &= \text{qr}(B1, 0) ; \\
\text{>> } x1 &= \text{lsqr}(A, b, \text{eps}, 100, R) ;
\end{align*}
\]

lsqr converged at iteration 15 to a solution with relative residual 1

Theorem:

\[
O \left( mn \log(m) + \left( \sqrt{n} + \sqrt{8 \log(m)} \right)^2 n^2 \log(n) + mn \log \epsilon^{-1} \right)
\]

with high probability

(\text{using Boutsidis & Gittens (2012)})
Blendenpik - a fast and accurate least squares solver

\[
\text{LAPACK time / Blendenpik time}
\]

\[
m \text{-by-} (m / 40) \text{ well-conditioned matrices}
\]
Some more experiments...

- m-by-(m / 40) well-conditioned incoherent matrices
- Row mixing
- Sample+QR
- LSQR

- Incoherent
- Semi-coherent
- Coherent
- LAPACK

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>120000x750</td>
<td>120</td>
</tr>
<tr>
<td>90000x1000</td>
<td>90</td>
</tr>
<tr>
<td>60000x1500</td>
<td>60</td>
</tr>
<tr>
<td>40000x2250</td>
<td>40</td>
</tr>
<tr>
<td>30000x3000</td>
<td>30</td>
</tr>
</tbody>
</table>
Sparse Matrices

If $A$ is sparse we want to use that.

- Input sparsity methods - construct a subspace embedding that uses sparsity in $A$. Proceed with dense methods.
  - Meng, Saunders & Mahoney (2011) (LSRN)
  - Clarkson & Woodruff (2012) - Talk at 5pm.

Especially useful for unstructured sparse matrices, not so much for structured matrices.

- Structure utilizing methods - use the combinatorial structure to approximate leverages.
Laplacian Matrices

- Leverages in the edge-incidence matrix are effective resistances (Drineas and Mahoney (2010))

- This observation can be utilized for fast solvers.

- More on that on the next talk by Ioannis Koutis.
\[ A = \sum_{e \in E} A_e \] is called finite-element matrix

\( A_e \) are called the element matrices

\( A_e \) are small PSD dense matrices embedded in big sparse matrix

Think of: an hypergraph with a small matrix associated with every hyperedge (rows/columns correspond to vertices)
“Finite Element Matrices”

Such structure arises in finite element discretization of PDEs (hence the name)

But it also arises in other applications, e.g. approximate multicommodity flow (Kelner, Miller and Peng (2012) STOC)
The effective stiffness matrix shows how the entire structure responds to a load that is placed on one element. Algebraically, for element $e$

- $\bar{K}$ - reorder the columns so its vertices are last
- Partition:
  $$\bar{K} = \begin{pmatrix} \bar{K}_{11} & \bar{K}_{12} \\ \bar{K}_{12}^T & \bar{K}_{22} \end{pmatrix}$$

- $S_e = \bar{K}_{22} - \bar{K}_{12}^T \bar{K}_{11}^{-1} \bar{K}_{12}$ (Schur complement)

This is a formal way to say: eliminate from $K$ all columns not associated with $e$. 

Definition

The leverage of an element $e$ is

$$\tau_e = \lambda_{\text{max}}(K_e, S_e)$$

Theorem

Let $\tilde{\tau}_e \geq \tau_e$ and $\tilde{\tau} = \sum_{e \in E} \tilde{\tau}_e$. Given $K = \sum_{e \in E} K_e$ sample $O(\tilde{\tau} \log \tilde{\tau})$ elements to form $H \subseteq E$, where element $e$ is sampled with probability relative to $\tilde{\tau}_e$. With high probability

$$\kappa(K, \sum_{e \in H} \tilde{\tau}_e^{-1} K_e) = O(1).$$
Approximating the leverages

The leverages can be approximated locally:

- For an element $e$ find the set of elements with a radius $r$ from $e$ in the dual graph.
- Compute the leverage only in this sub-model.
- This guarantees: $\tilde{\tau}_e \geq \tau_e$. 

\[\frac{1}{r} \sum_{i \in S} \frac{1}{d_{i,e}} \geq \frac{1}{r} \sum_{i \in S} \frac{1}{d_{i,e}} \]
Still work in progress...

Two more issues need to be solved for an efficient solver:

1. Number of elements: in typical FEM number of elements is only $O(n)$.
2. We really want to minimize number of non-zeros in factor, not number of elements.

Both issues were already solved for the special case of Laplacians (next talk!)
Summary

Efficient and reliable in applied numerical linear algebra means:
- poly-log running time dependence on the accuracy
- Deterministic error bounds (probabilistic running time is OK).

Achieving these is not easy for randomized algorithms.
For solving linear equations it can be achieved by preconditioning
- General strategy: combine randomization with older techniques.

Dense matrices: can beat the state-of-the-art (LAPACK) in practice.
Sparse matrices: possible for certain classes (SDD, FEM), but still work in progress.
Thank You! Questions?