Randomised algorithms for solving systems of linear equations

Per-Gunnar Martinsson
Dept. of Mathematics & Oden Inst. for Computational Sciences and Engineering
University of Texas at Austin

Students, postdocs, collaborators: Tracy Babb, Ke Chen, Robert van de Geijn, Abinand Gopal, Nathan Halko, Nathan Heavner, James Levitt, Yijun Liu, Gregorio Quintana-Ortí, Joel Tropp, Bowei Wu, Sergey Voronin, Anna Yesypenko, Patrick Young.

Slides: http://users.oden.utexas.edu/~pgm/main_talks.html

Research support by:
**Scope:** Let $\mathbf{A}$ be a given $m \times n$ matrix (real or complex), and let $\mathbf{b}$ be a given vector. The talk is about techniques for solving

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$ 

**Environments considered:** (Time permitting \ldots )

- $\mathbf{A}$ is square, nonsingular, and stored in RAM.
- $\mathbf{A}$ is square, nonsingular, and stored out of core. (Very large matrix.)
- $\mathbf{A}$ is rectangular, with $m \gg n$. (Very over determined system.)
- $\mathbf{A}$ is a graph Laplacian matrix. (Very large, and sparse).
- $\mathbf{A}$ is an $n \times n$ “kernel matrix”, in the sense that given some set of points $\{\mathbf{x}_i\}_{i=1}^n \subset \mathbb{R}^d$, the $ij$ entry of the matrix can be written as $k(\mathbf{x}_i, \mathbf{x}_j)$ for some kernel function $k$.
  
  ⭐ Scientific computing: High accuracy required.
  
  ⭐ Data analysis: $d$ is large ($d = 4$, or 10, or 100, or 1000, \ldots ).

**Techniques:** The recurring theme is *randomisation*.

- Randomized sampling. Typically used to build preconditioners.
- Randomized embeddings. Reduce effective dimension of intermediate steps.

**Prelude:** We introduce the ideas around *randomized embeddings* by reviewing randomized techniques for low rank approximation. (Recap of Tuesday talk.)
Randomised SVD:

**Objective:** Given an \( m \times n \) matrix \( A \) of approximate rank \( k \), compute a factorisation

\[
A \approx U \ D \ V^* 
\]

\[
\begin{array}{cccc}
m \times n & m \times k & k \times k & k \times n \\
\end{array}
\]

where \( U \) and \( V \) are orthonormal, and \( D \) is diagonal. (We assume \( k \ll \min(m, n) \).)
Randomised SVD:

**Objective:** Given an $m \times n$ matrix $A$ of approximate rank $k$, compute a factorisation

$$A \approx UDV^*$$

where $U$ and $V$ are orthonormal, and $D$ is diagonal. (We assume $k \ll \min(m, n)$.)

The only error we seek to control is

$$||A - UDV^*||.$$  

We do not aspire to approximate small singular values, to get high *relative* errors, etc.
Randomised SVD:

**Objective:** Given an $m \times n$ matrix $A$ of approximate rank $k$, compute a factorisation

$$A \approx U D V^*$$

$$m \times n \quad m \times k \quad k \times k \quad k \times n$$

where $U$ and $V$ are orthonormal, and $D$ is diagonal. (We assume $k \ll \min(m, n)$.)

**Stage A: Build an approximate basis for the range of $A$:** (Using randomisation!)

A.1 Draw an $n \times k$ Gaussian random matrix $R$.

$$R = \text{randn}(n,k)$$

A.2 Form the $m \times k$ sample matrix $Y = AR$.

$$Y = A * R$$

A.3 Form an $m \times k$ orthonormal matrix $Q$ such that $\text{ran}(Q) = \text{ran}(Y)$.

$$[Q, \sim] = \text{qr}(Y)$$

**Stage B: Restrict $A$ to the computed subspace and perform an exact factorisation:**

B.1 Form the $k \times n$ matrix $B = Q^* A$.

$$B = Q' * A$$

B.2 Form SVD of the matrix $B$: $B = \hat{U}D\hat{V}^*$. 

$$[\hat{U}, \text{Sigma}, \hat{V}] = \text{svd}(B,'\text{econ'})$$

B.3 Form the matrix $U = Q \hat{U}$.

$$U = Q * \hat{U}$$

The objective of Stage A is to compute an ON-basis that approximately spans the column space of $A$. The matrix $Q$ holds these basis vectors and $A \approx QQ^*A$. 
**Randomised SVD:**

**Objective:** Given an \( m \times n \) matrix \( A \) of approximate rank \( k \), compute a factorisation:

\[
A \approx U D V^* \\
m \times n \quad m \times k \quad k \times k \quad k \times n
\]

where \( U \) and \( V \) are orthonormal, and \( D \) is diagonal. (We assume \( k \ll \min(m,n) \).)

**Stage A: Build an approximate basis for the range of \( A \): (Using randomisation!)**

A.1 Draw an \( n \times k \) Gaussian random matrix \( R \).

\[
R = \text{randn}(n,k)
\]

A.2 Form the \( m \times k \) sample matrix \( Y = AR \).

\[
Y = A \ast R
\]

A.3 Form an \( m \times k \) orthonormal matrix \( Q \) such that \( \text{ran}(Q) = \text{ran}(Y) \).

\[
[Q, \sim] = \text{qr}(Y)
\]

**Stage B: Restrict \( A \) to the computed subspace and perform an exact factorisation:**

B.1 Form the \( k \times n \) matrix \( B = Q^* A \).

\[
B = Q' \ast A
\]

B.2 Form SVD of the matrix \( B \): \( B = \hat{U}D\hat{V}^* \).

\[
[\hat{U}, \text{Sigma}, V] = \text{svd}(B,'econ')
\]

B.3 Form the matrix \( U = Q \hat{U} \).

\[
U = Q \ast \hat{U}
\]

The objective of Stage A is to compute an ON-basis that approximately spans the column space of \( A \). The matrix \( Q \) holds these basis vectors and \( A \approx QQ^* A \).

Stage B is exact:

\[
\| A - Q \hat{U} D V^* \| = \| A - U D V^* \|.
\]
**Randomised SVD:**

**Objective:** Given an $m \times n$ matrix $A$ of approximate rank $k$, compute a factorisation

$$ A \approx U D V^* $$

where $U$ and $V$ are orthonormal, and $D$ is diagonal. (We assume $k \ll \min(m, n)$.)

**Stage A: Build an approximate basis for the range of $A$: (Using randomisation!)**

A.1 Draw an $n \times k$ Gaussian random matrix $R$.

$$ R = \text{randn}(n,k) $$

A.2 Form the $m \times k$ sample matrix $Y = AR$.

$$ Y = A \ast R $$

A.3 Form an $m \times k$ orthonormal matrix $Q$ such that $\text{ran}(Q) = \text{ran}(Y)$.

$$ [Q, \sim] = \text{qr}(Y) $$

**Stage B: Restrict $A$ to the computed subspace and perform an exact factorisation:**

B.1 Form the $k \times n$ matrix $B = Q^* A$.

$$ B = Q' \ast A $$

B.2 Form SVD of the matrix $B$: $B = \hat{U} D V^*$.

$$ [\text{Uhat}, \Sigma, V] = \text{svd}(B,'econ') $$

B.3 Form the matrix $U = Q \ast \text{Uhat}$.

$$ U = Q \ast \text{Uhat} $$

Distortions in the randomised projections are fine, since all we need is a subspace that captures “the essential” part of the range. Pollution from unwanted singular modes is harmless, as long as we capture the dominant ones. By drawing $p$ extra samples (for, say, $p = 5$ or $p = 10$), we make the risk of missing anything important essentially zero.
**Randomised SVD:**

**Input:** An $m \times n$ matrix $A$, a target rank $k$, and an over-sampling parameter $p$ (say $p = 5$).

**Output:** Rank-$(k + p)$ factors $U$, $D$, and $V$ in an approximate SVD $A \approx UDV^*$. 

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Draw an $n \times (k + p)$ random matrix $R$.</td>
</tr>
<tr>
<td>2.</td>
<td>Form the $m \times (k + p)$ sample matrix $Y = AR$.</td>
</tr>
<tr>
<td>3.</td>
<td>Compute an ON matrix $Q$ s.t. $Y = QQ^*Y$.</td>
</tr>
<tr>
<td>4.</td>
<td>Form the small matrix $B = Q^* A$.</td>
</tr>
<tr>
<td>5.</td>
<td>Factor the small matrix $B = \hat{U}DV^*$.</td>
</tr>
<tr>
<td>6.</td>
<td>Form $U = Q\hat{U}$.</td>
</tr>
</tbody>
</table>

- It is simple to adapt the scheme to the situation where the tolerance is given, and the rank has to be determined adaptively.
- The RSVD is in many environments far faster than classical deterministic techniques. The primary reason is that it requires less communication → the computational primitive is the matrix-matrix multiplication. The method is particularly effective for GPUs, data stored out-of-core, distributed computing, etc.
- A unique advantage of the RSVD is that it can be modified to operate on streaming data that cannot be stored at all. “Single-view.”
- Accuracy of the basic scheme is good when the singular values decay reasonably fast. When they do not, the scheme can be combined with Krylov-type ideas: Taking one or two steps of subspace iteration vastly improves the accuracy. For instance, use the sampling matrix $Y = AA^*AG$ instead of $Y = AG$. 


Randomised SVD:

**Input:** An $m \times n$ matrix $A$, a target rank $k$, and an over-sampling parameter $p$ (say $p = 5$).

**Output:** Rank-$(k + p)$ factors $U$, $D$, and $V$ in an approximate SVD $A \approx UDV^*$. 

| (1) Draw an $n \times (k + p)$ random matrix $R$. | (4) Form the small matrix $B = Q^* A$. |
| (2) Form the $m \times (k + p)$ sample matrix $Y = AR$. | (5) Factor the small matrix $B = \hat{U}DV^*$. |
| (3) Compute an ON matrix $Q$ s.t. $Y = QQ^*Y$. | (6) Form $U = Q\hat{U}$. |

- It is simple to adapt the scheme to the situation where the tolerance is given, and the rank has to be determined adaptively.

- The RSVD is in many environments far faster than classical deterministic techniques. The primary reason is that it requires less communication → the computational primitive is the matrix-matrix multiplication. The method is particularly effective for GPUs, data stored out-of-core, distributed computing, etc.

- A unique advantage of the RSVD is that it can be modified to operate on streaming data that cannot be stored at all. “Single-view”.

- Accuracy of the basic scheme is good when the singular values decay reasonably fast. When they do not, the scheme can be combined with Krylov-type ideas: Taking one or two steps of subspace iteration vastly improves the accuracy. For instance, use the sampling matrix $Y = AA^*AG$ instead of $Y = AG$. 


Randomised SVD:

![Graph showing spectral norm error and accuracy in rank k approximation for different methods]

The plot shows the errors from the randomised range finder. To be precise, we plot

\[ e_k = \|A - P_k A\|, \]

where \( P_k \) is the orthogonal projection onto the first \( k \) columns of

\[ Y = (AA^*)^q AG, \]

and where \( G \) is a Gaussian random matrix. (Recall that \( P_k A = U_k D_k V_k^* \).) The matrix \( A \) is an approximation to a scattering operator for a Helmholtz problem.
The plot shows the errors from the randomised range finder. To be precise, we plot

\[ e_k = \| A - P_k A \|, \]

where \( P_k \) is the orthogonal projection onto the first \( k \) columns of

\[ Y = (AA^*)^q AG, \]

and where \( G \) is a Gaussian random matrix. (Recall that \( P_k A = U_k D_k V_k^* \).)

The matrix \( A \) now has singular values that decay slowly.
Randomised SVD: The same plot as before, but now showing 100 instantiations.

The darker lines show the mean errors across the 100 experiments.
**Randomised SVD:** The same plot as before, but now showing 100 instantiations.

*The darker lines show the mean errors across the 100 experiments.*
**Randomised SVD:**

| Input: An $m \times n$ matrix $A$, a target rank $k$, and an over-sampling parameter $p$ (say $p = 5$). |
| Output: Rank-$(k + p)$ factors $U$, $D$, and $V$ in an approximate SVD $A \approx UDV^*$. |

1. Draw an $n \times (k + p)$ random matrix $R$.  
2. Form the $m \times (k + p)$ sample matrix $Y = AR$.  
3. Compute an ON matrix $Q$ s.t. $Y = QQ^*Y$.  
4. Form the small matrix $B = Q^* A$.  
5. Factor the small matrix $B = \hat{U}DV^*$.  
6. Form $U = Q\hat{U}$.

We can reduce the flop count from $O(mnk)$ to $O(mn\log k)$ by using a so called “fast Johnson-Lindenstrauss” transform. A popular choice is the **subsampled random Fourier Transform (SRFT)** which can be applied using a variation of the FFT. Many other options: sub-sampled Hadamard transform, chains of Givens rotations, sparse projections, ...
**Randomised SVD:**

**Input:** An $m \times n$ matrix $A$, a target rank $k$, and an over-sampling parameter $p$ (say $p = 5$).

**Output:** Rank-$(k + p)$ factors $U$, $D$, and $V$ in an approximate SVD $A \approx UDV^*$. 

| (1) Draw an $n \times (k + p)$ random matrix $R$. | (4) Form the small matrix $B = Q^* A$. |
| (2) Form the $m \times (k + p)$ sample matrix $Y = AR$. | (5) Factor the small matrix $B = \hat{UDV}^*$. |
| (3) Compute an ON matrix $Q$ s.t. $Y = QQ^*Y$. | (6) Form $U = Q\hat{U}$. |

We can reduce the flop count from $O(mnk)$ to $O(mn\log k)$ by using a so-called “fast Johnson-Lindenstrauss” transform. A popular choice is the *subsampled random Fourier Transform (SRFT)* which can be applied using a variation of the FFT. Many other options: sub-sampled Hadamard transform, chains of Givens rotations, sparse projections, ... 

**Example:** The SRFT takes the form

$$R = D F S.$$  

$n \times k \quad n \times n \quad n \times n \quad n \times k$

- $D$ is a diagonal matrix whose entries are i.i.d. random variables drawn from a uniform distribution on the unit circle in $\mathbb{C}$.
- $F$ is the discrete Fourier transform, $F_{pq} = \frac{1}{\sqrt{n}} e^{-2\pi i (p-1)(q-1)/n}$.
- $S$ is a matrix whose entries are all zeros except for a single, randomly placed 1 in each column. (So the action of $S$ is to draw $k$ columns at random from $DF$.)
### Randomised SVD:

**Input:** An $m \times n$ matrix $A$, a target rank $k$, and an over-sampling parameter $p$ (say $p = 5$).

**Output:** Rank-$(k + p)$ factors $U$, $D$, and $V$ in an approximate SVD $A \approx UDV^*$. 

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Draw an $n \times (k + p)$ random matrix $R$.</td>
</tr>
<tr>
<td>(2)</td>
<td>Form the $m \times (k + p)$ sample matrix $Y = AR$.</td>
</tr>
<tr>
<td>(3)</td>
<td>Compute an ON matrix $Q$ s.t. $Y = QQ^*Y$.</td>
</tr>
<tr>
<td>(4)</td>
<td>Form the small matrix $B = Q^* A$.</td>
</tr>
<tr>
<td>(5)</td>
<td>Factor the small matrix $B = \hat{U}D\hat{V}^*$.</td>
</tr>
<tr>
<td>(6)</td>
<td>Form $U = Q\hat{U}$.</td>
</tr>
</tbody>
</table>

We can reduce the flop count from $O(mnk)$ to $O(mn\log k)$ by using a so called “fast Johnson-Lindenstrauss” transform. A popular choice is the *subsampled random Fourier Transform (SRFT)* which can be applied using a variation of the FFT. Many other options: sub-sampled Hadamard transform, chains of Givens rotations, sparse projections, ... 

- The algorithm must be modified a bit beside replacing the random matrix.
- The SRFT leads to large speed-ups for moderate matrix sizes.
  - For instance, for $m = n = 4000$, and $k \sim 10^2$, we observe about $\times 5$ speedup.
- In practice, accuracy is similar to what you get from Gaussian random matrices.
- Theory is still quite weak.

**References:** Ailon and Chazelle (2006); Liberty, Rokhlin, Tygert, and Woolfe (2006). Halko, Martinsson, Tropp (2011). Much subsequent work ...
Linear solvers

Given an $m \times n$ matrix $A$ (real or complex), we consider the task of solving

$$Ax = b.$$ 

Focus is on the case where $A$ is of size $n \times n$ and non-singular.

The techniques we describe can be organized as follows:

- $O(n^3)$ methods for general coefficient matrices.
- Faster than $O(n^3)$ methods for general coefficient matrices?
- Linear complexity methods for “special” coefficient matrices.

Observation: When $A$ is well-conditioned, iterative methods converge rapidly.

Worst case complexity for solving $Ax = b$ to precision $\epsilon$ is then

$$\log\left(\frac{1}{\epsilon}\right) \times \text{Cost of matrix-vector multiplication}.$$ 

For a dense matrix, this of course works out to $O(n^2 \log(1/\epsilon))$.

The challenge concerns matrices that are ill-conditioned.

Or, to be more precise, whose spectra are not clustered.
Linear solvers

Given an \( m \times n \) matrix \( A \) (real or complex), we consider the task of solving

\[
Ax = b.
\]

Focus is on the case where \( A \) is of size \( n \times n \) and non-singular.

The techniques we describe can be organized as follows:

- \( O(n^3) \) methods for general coefficient matrices.
- Faster than \( O(n^3) \) methods for general coefficient matrices?
- Linear complexity methods for “special” coefficient matrices.

**Observation:** When \( A \) is well-conditioned, iterative methods converge rapidly.

Worst case complexity for solving \( Ax = b \) to precision \( \varepsilon \) is then

\[
\log \left( \frac{1}{\varepsilon} \right) \times \text{Cost of matrix-vector multiplication}.
\]

For a dense matrix, this of course works out to \( O(n^2 \log(1/\varepsilon)) \).

The challenge concerns matrices that are ill-conditioned.

Or, to be more precise, whose spectra are not clustered.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Suppose $A$ is a dense ill-conditioned matrix of moderate size. In such a case, it is natural to look to $O(n^3)$ methods that compute a full factorisation of the matrix.

Standard options (all with complexity $O(n^3)$) include:

- Unpivoted QR (QR)
- Partially pivoted LU
- Column pivoted QR (CPQR)
- Fully pivoted LU
- SVD

Not always stable.
Always stable.
Fast.
Slow.
Suppose $A$ is a dense ill-conditioned matrix of moderate size. In such a case, it is natural to look to $O(n^3)$ methods that compute a full factorisation of the matrix.

Standard options (all with complexity $O(n^3)$) include:

<table>
<thead>
<tr>
<th></th>
<th>Unpivoted QR (QR)</th>
<th>Partially pivoted LU</th>
<th>Column pivoted QR (CPQR)</th>
<th>Fully pivoted LU</th>
<th>SVD</th>
</tr>
</thead>
</table>

The “robust” factorisations to the right all depend on algorithms that proceed through a sequence of rank-one updates to the matrix. This makes them slow when executed on modern hardware (even on a single core).
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Suppose $A$ is a dense ill-conditioned matrix of moderate size. In such a case, it is natural to look to $O(n^3)$ methods that compute a full factorisation of the matrix.

Standard options (all with complexity $O(n^3)$) include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Stability</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpivoted QR (QR)</td>
<td>Not always stable.</td>
<td>Fast.</td>
</tr>
<tr>
<td>Column pivoted QR (CPQR)</td>
<td>Always stable.</td>
<td>Slow.</td>
</tr>
<tr>
<td>Fully pivoted LU</td>
<td>Always stable.</td>
<td>Slow.</td>
</tr>
<tr>
<td>SVD</td>
<td>Always stable.</td>
<td>Slow.</td>
</tr>
</tbody>
</table>

![Computational time to factorize matrix](chart.png)
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

The culprit preventing us from attaining high performance is *pivoting* since it relies on a sequence of rank-one updates.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

The culprit preventing us from attaining high performance is *pivoting* since it relies on a sequence of rank-one updates.

**Randomisation to the rescue!** D. Stott Parker (1995) proposed an elegant solution:

1. Randomly mix the columns by right multiplying $A$ by a random unitary matrix $V$:

   $$ A_{\text{rand}} = AV. $$

2. Perform unpivoted QR on the new matrix

   $$ A_{\text{rand}} = UR $$

The resulting factorisation

$$ A = A_{\text{rand}} V^* = URV^* $$

is provably “rank-revealing” and leads to stable linear solves.

For computational efficiency, Parker introduced a random structured matrix (a bit ahead of the times) called a “random butterfly transform”.

Further refinements — Demmel, Dumitriu, Holtz, Grigori, Dongarra, etc.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Improved URV factorisation: Do $q$ steps of power iteration (for $q = 1$ or $q = 2$, say):

1. Draw an $n \times n$ Gaussian random matrix $G$ and form $Y = (AA^*)^q G$.
2. Perform unpivoted QR on $Y$ so that $Y = VR_{\text{trash}}$.
3. Perform unpivoted QR on $AV$ so that $AV = UR$.

This results in a factorisation

$$A = (AV)V^* = URV^*$$

that is excellent at revealing the rank of $A$. Faster than CPQR, despite far more flops.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Improved URV factorisation: Do $q$ steps of power iteration (for $q = 1$ or $q = 2$, say):

1. Draw an $n \times n$ Gaussian random matrix $G$ and form $Y = (AA^*)^q G$.
2. Perform unpivoted QR on $Y$ so that $Y = VR_{\text{trash}}$.
3. Perform unpivoted QR on $AV$ so that $AV = UR$.

This results in a factorisation

$$A = (AV)V^* = URV^*$$

that is excellent at revealing the rank of $A$. Faster than CPQR, despite far more flops.

**Exact and estimated singular values**

- **CPQR**
- **randURV $q=0$**
- **randURV $q=1$**
- **randURV $q=2$**
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Improved URV factorisation: Do $q$ steps of power iteration (for $q = 1$ or $q = 2$, say):
1. Draw an $n \times n$ Gaussian random matrix $G$ and form $Y = (AA^*)^q G$.
2. Perform unpivoted QR on $Y$ so that $Y = VR_{\text{trash}}$.
3. Perform unpivoted QR on $AV$ so that $AV = UR$.

This results in a factorisation

$$A = (AV)V^* = URV^*$$

that is excellent at revealing the rank of $A$. Faster than CPQR, despite far more flops.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Improved URV factorisation: Do $q$ steps of power iteration (for $q = 1$ or $q = 2$, say):

1. Draw an $n \times n$ Gaussian random matrix $G$ and form $Y = (AA^*)^q G$.
2. Perform unpivoted QR on $Y$ so that $Y = VR_{\text{trash}}$.
3. Perform unpivoted QR on $AV$ so that $AV = UR$.

This results in a factorisation

$$A = (AV)V^* = URV^*$$

that is excellent at revealing the rank of $A$. Faster than CPQR, despite far more flops.
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Improved URV factorisation: Do $q$ steps of power iteration (for $q = 1$ or $q = 2$, say):

1. Draw an $n \times n$ Gaussian random matrix $G$ and form $Y = (AA^*)^q G$.
2. Perform unpivoted QR on $Y$ so that $Y = VR_{\text{trash}}$.
3. Perform unpivoted QR on $AV$ so that $AV = UR$.

This results in a factorisation

$$A = (AV)V^* = URV^*$$

that is excellent at revealing the rank of $A$. Faster than CPQR, despite far more flops.

The method is extremely simple to code:

```matlab
G = randn(n);
for j = 1:q
    G = A*(A'*G);
end
[V,~] = qr(G); [U,R] = qr(A*V);
```

But ...
But ... the times for CPQR refer to classical deterministic CPQR.

It turns out that we can greatly accelerate this computation through randomisation.
Randomised methods for solving $\mathbf{Ax} = \mathbf{b}$: $O(n^3)$ complexity methods

Given a dense $n \times n$ matrix $\mathbf{A}$, compute a column pivoted QR factorisation

$$\mathbf{A} \mathbf{P} \approx \mathbf{Q} \mathbf{R},$$

where, as usual, $\mathbf{Q}$ should be ON, $\mathbf{P}$ is a permutation, and $\mathbf{R}$ is upper triangular.

The technique proposed is based on a *blocked* version of classical Householder QR:

Each $\mathbf{P}_j$ is a permutation matrix computed via randomised sampling.
Each $\mathbf{Q}_j$ is a product of Householder reflectors.

The key challenge has been to find good permutation matrices.
We seek $\mathbf{P}_j$ so that the set of $\mathbf{b}$ chosen columns *has maximal spanning volume.*

Perfect for randomised sampling! The likelihood that any block of columns is “hit” by the random vectors is directly proportional to its volume. Perfect optimality is *not* required.
**Randomised methods for solving** $Ax = b$: $O(n^3)$ complexity methods

*How to do block pivoting using randomisation:*

Let $A$ be of size $m \times n$, and let $b$ be a block size.

\[ A \rightarrow Q^*AP \]

$Q$ is a product of $b$ Householder reflectors.

$P$ is a permutation matrix that moves $b$ “pivot” columns to the leftmost slots.

We seek $P$ so that the set of chosen columns has maximal spanning volume.

Draw a Gaussian random matrix $G$ of size $b \times m$ and form

\[ Y = GA \]

The rows of $Y$ are random linear combinations of the rows of $A$.

Then compute the pivot matrix $P$ for the first block by executing traditional column pivoting on the small matrix $Y$: 

\[ YP = Q_{\text{trash}} R_{\text{trash}} \]

\[ b \times n \quad n \times n \quad b \times b \quad b \times n \]
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Speed-up attained by our randomised algorithm HQRRP for computing a full column pivoted QR factorisation of an $n \times n$ matrix. The speed-up is measured versus LAPACK’s faster routine $\text{dgeqp3}$ as implemented in Netlib (left) and Intel’s MKL (right). Our implementation was done in C, and was executed on an Intel Xeon E5-2695. Joint work with G. Quintana-Ortí, N. Heavner, and R. van de Geijn. Available at: https://github.com/flame/hqrrp/
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

Given a dense $n \times n$ matrix $A$, compute a factorisation

$$A = U^* T V^*, \quad n \times n \quad n \times n \quad n \times n \quad n \times n \quad n \times n$$

where $T$ is upper triangular, $U$ and $V$ are unitary.

Observe: More general than CPQR since we used to insist that $V$ be a permutation.

The technique proposed is based on a blocked version of classical Householder QR:

Both $U_j$ and $V_j$ are (mostly...) products of $b$ Householder reflectors.

Our objective is in each step to find an approximation to the linear subspace spanned by the $b$ dominant singular vectors of a matrix. The randomised range finder is perfect for this, especially when a small number of power iterations are performed. Easier and more natural than choosing pivoting vectors.
Rank-k approximation errors for the matrix “Fast Decay” of size $4000 \times 4000$. The black lines mark the theoretically minimal errors. The block size was $b = 100$ and the green vertical lines mark block limits.
Rank-k approximation errors for the matrix “BIE” of size $4000 \times 4000$. The black lines mark the theoretically minimal errors. The block size was $b = 100$ and the green vertical lines mark block limits.
Rank-k approximation errors for $k \leq 300$ for the matrix “Gap” of size $4000 \times 4000$. The black lines mark the theoretically minimal errors. The block size was $b = 100$ and the green vertical lines mark block limits.
The diagonal entries of the T-matrix in the UTV decomposition (red) provide excellent approximations to the true singular values (black).
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

For the task of computing low-rank approximations to matrices, the classical choice is between SVD and column pivoted QR (CPQR). SVD is slow, and CPQR is inaccurate:

![Accuracy vs Speed Graph]

- **Accuracy**:
  - **Optimal**
  - **Ok**

- **Speed**:
  - **Slow**
  - **Fast**

- **SVD**
- **CPQR**

The randomised algorithm randUTV combines the best properties of both factorisations. Additionally, randUTV parallelizes better, and allows the computation of partial factorisations (like CPQR, but unlike SVD).
Randomised methods for solving $Ax = b$: $O(n^3)$ complexity methods

For the task of computing low-rank approximations to matrices, the classical choice is between SVD and column pivoted QR (CPQR). SVD is slow, and CPQR is inaccurate:

The randomised algorithm \textit{randUTV} combines the best properties of both factorisations. Additionally, \textit{randUTV} parallelizes better, and allows the computation of partial factorisations (like CPQR, but unlike SVD).
Randomised methods for solving $Ax = b$: Strassen-type methods

The essential feature of the randomised methods described is that they enable us to expend almost all flops on the matrix-matrix computation, which is much faster per flop than other matrix operations.

Alternatively, use *asymptotically* faster methods for the matrix-matrix multiplication:

- **Strassen**: $O(n^{2.83})$. Stable. Reasonable breakeven point.
- **Coppersmith-Winograd etc.**: $O(n^{2.37})$. Unstable. Unreasonable breakeven point.

**Observation:**

Randomisation allows you to use “fast” matrix-matrix multiplication algorithms to compute rank-revealing factorisations in a numerically stable way. In particular:

$$\text{fast+stable matrix-matrix multiplication } \Rightarrow \text{ fast+stable linear system solve}$$

Randomised methods for solving $Ax = b$: RSVĐ as pre-conditioner

Let us consider $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ a symmetric positive definite matrix. A standard solution technique here is conjugate gradients (CG). The error at step $k$ is known to converge to zero with at least the speed $O(\gamma^k)$ where

$$\gamma = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1},$$

and where $\kappa(A)$ is the condition number of $A$. 

But the clustering of the spectrum, matters! Consider four spectra with $\lambda_{\text{max}} / \lambda_{\text{min}} = 10$: 

(a) (b) (c) (d)
Randomised methods for solving $Ax = b$: RSVĐ as pre-conditioner

Let us consider $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ a symmetric positive definite matrix. A standard solution technique here is conjugate gradients (CG). The error at step $k$ is known to converge to zero with at least the speed $O(\gamma^k)$ where

$$\gamma = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1},$$

and where $\kappa(A)$ is the condition number of $A$. But the clustering of the spectrum, matters! Consider four spectra with $\lambda_{\text{max}}/\lambda_{\text{min}} = 10$:

(a) CG converges to the exact answer after 2 iterations.
Randomised methods for solving $Ax = b$: RSVD as pre-conditioner

Let us consider $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ a symmetric positive definite matrix. A standard solution technique here is conjugate gradients (CG). The error at step $k$ is known to converge to zero with at least the speed $O(\gamma^k)$ where

$$\gamma = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1},$$

and where $\kappa(A)$ is the condition number of $A$. But the clustering of the spectrum, matters! Consider four spectra with $\lambda_{\text{max}}/\lambda_{\text{min}} = 10$:

(b) RSVD provides a preconditioner! Suppose $A \approx UDU^*$ captures the $k$ largest eigenmodes. Then use

$$M = \frac{1}{\lambda_{\text{approx}}^{k+1}} UDU + (I - UDU^*).$$

as a preconditioner to “attenuate” the outlying large eigenvalues.
Randomised methods for solving $Ax = b$: RSVD as pre-conditioner

Let us consider $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ a symmetric positive definite matrix. A standard solution technique here is conjugate gradients (CG). The error at step $k$ is known to converge to zero with at least the speed $O(\gamma^k)$ where

$$\gamma = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1},$$

and where $\kappa(A)$ is the condition number of $A$. But the clustering of the spectrum, matters! Consider four spectra with $\lambda_{\text{max}}/\lambda_{\text{min}} = 10$:

(c) & (d): Subject of current research …
Randomised methods for solving $Ax = b$: Randomised pre-conditioning

Let us consider

$$Ax = b$$

for $A \in \mathbb{R}^{m \times n}$ with $m \gg n$. Complexity of standard solvers: $O(mn^2)$
Randomised methods for solving $Ax = b$: Randomised pre-conditioning

Let us consider

$$Ax = b$$

for $A \in \mathbb{R}^{m \times n}$ with $m \gg n$. Complexity of standard solvers: $O(mn^2)$

Method proposed by Rokhlin and Tygert (PNAS 2008): Form a “sketched equation”

$$X^*Ax = X^*b$$

where $X$ is an $m \times \ell$ SRFT. Compute QR factorisation of the new coefficient matrix

$$X^*A = QRP^*.$$ 

Form a preconditioner

$$M = RP^*.$$ 

Solve the preconditioned linear system

$$(AM^{-1})(Mx) = b$$

for the new unknown $y$. Complexity of randomised solver: $O((\log(n) + \log(1/\varepsilon))mn + n^3)$.

Later improvements include BLENDENPIK by Avron, Maymounkov, Toledo (2010).
Randomised methods for solving $Ax = b$: Randomised pre-conditioning

Let us consider

$$Ax = b$$

for $A \in \mathbb{R}^{m \times n}$ with $m \gg n$. Complexity of standard solvers: $O(mn^2)$

Method proposed by Rokhlin and Tygert (PNAS 2008): Form a “sketched equation”

$$X^*Ax = X^*b$$

where $X$ is an $m \times \ell$ SRFT. Compute QR factorisation of the new coefficient matrix

$$X^*A = QRP^*.$$ 

Form a preconditioner

$$M = RP^*.$$ 

Solve the preconditioned linear system

$$\left(AM^{-1}\right)\left(Mx\right) = b$$

for the new unknown $y$. Complexity of randomised solver: $O((\log(n) + \log(1/\varepsilon))mn + n^3)$. 

Later improvements include BLENDENPIK by Avron, Maymounkov, Toledo (2010).
Randomised methods for solving $Ax = b$: Randomised Kaczmarz

The classical Kaczmarz algorithm:
With $A \in \mathbb{R}^{m \times n}$, we seek to solve $Ax = b$ through an iterative procedure. Given an approximate solution $x_{\text{old}}$, compute an improved solution $x_{\text{new}}$ as follows:

1. Pick a row index $i \in \{1, 2, \ldots, m\}$.
2. Require that $x_{\text{new}}$ is picked so that row $i$ of the system is satisfied exactly.
3. Within the hyperplane defined by (2), pick $x_{\text{new}}$ as the point closest to $x_{\text{old}}$.

The resulting formula is

$$x_{\text{new}} = x_{\text{old}} + b(i) - A(i,:)^\top x_{\text{old}} / \|A(i,:)^\top\|_2.$$
Randomised methods for solving $Ax = b$: Randomised Kaczmarz

The classical Kaczmarz algorithm:
With $A \in \mathbb{R}^{m \times n}$, we seek to solve $Ax = b$ through an iterative procedure. Given an approximate solution $x_{\text{old}}$, compute an improved solution $x_{\text{new}}$ as follows:

1. Pick a row index $i \in \{1, 2, \ldots, m\}$.
2. Require that $x_{\text{new}}$ is picked so that row $i$ of the system is satisfied exactly.
3. Within the hyperplane defined by (2), pick $x_{\text{new}}$ as the point closest to $x_{\text{old}}$.

The resulting formula is
$$x_{\text{new}} = x_{\text{old}} + \frac{b(i) - (A(i,:) \cdot x_{\text{old}})}{\|A(i,:)\|^2} A(i,:)^*.$$

Question: How do you pick the row index $i$ in step (1)?
Randomised methods for solving $Ax = b$: Randomised Kaczmarz

The classical Kaczmarz algorithm:
With $A \in \mathbb{R}^{m \times n}$, we seek to solve $Ax = b$ through an iterative procedure. Given an approximate solution $x_{\text{old}}$, compute an improved solution $x_{\text{new}}$ as follows:

1. Pick a row index $i \in \{1, 2, \ldots, m\}$.
2. Require that $x_{\text{new}}$ is picked so that row $i$ of the system is satisfied exactly.
3. Within the hyperplane defined by (2), pick $x_{\text{new}}$ as the point closest to $x_{\text{old}}$.

The resulting formula is $x_{\text{new}} = x_{\text{old}} + \frac{b(i) - (A(i,:) \cdot x_{\text{old}})}{\|A(i,:)\|^2} A(i,:)^\ast$.

**Question:** How do you pick the row index $i$ in step (1)?

**Strohmer & Vershynin (2009):** Draw $i$ with probability proportional to $\|A(i,:)\|$.

**Theorem:** Let $x_\star$ denote the exact solution to $Ax = b$, and let $x_k$ denote the $k$’th iterate of the S&V randomised Kaczmarz method. Then

$$
\mathbb{E}[\|x_k - x_\star\|] \leq \left(1 - \frac{1}{\kappa(A)^2}\right)^k \|x_0 - x_\star\|,
$$

where $\kappa(A)$ is the “scaled” condition number $\kappa(A) = \|A\|_F \|A^{-1}\|_2$. 

Randomised methods for solving $Ax = b$: Randomised Kaczmarz

The classical Kaczmarz algorithm:

With $A \in \mathbb{R}^{m \times n}$, we seek to solve $Ax = b$ through an iterative procedure.

Given an approximate solution $x_{\text{old}}$, compute an improved solution $x_{\text{new}}$ as follows:

1. Pick a row index $i \in \{1, 2, \ldots, m\}$.
2. Require that $x_{\text{new}}$ is picked so that row $i$ of the system is satisfied exactly.
3. Within the hyperplane defined by (2), pick $x_{\text{new}}$ as the point closest to $x_{\text{old}}$.

The resulting formula is $x_{\text{new}} = x_{\text{old}} + \frac{b(i) - (A(i,:) \cdot x_{\text{old}})}{\|A(i,:)\|^2} A(i,:)^*$. 

Question: How do you pick the row index $i$ in step (1)?

Gower & Richtarik (2015): Draw an $m \times \ell$ random map $X$

$$x_{\text{new}} = \arg\min \{\|y - x_{\text{old}}\| : y \text{ satisfies } X^*Ay = X^*b\}.$$ 

Leads to stronger analysis, and a much richer set of dimension reducing maps. In particular, it improves practical performance since it enables blocking.

Note: An ideal weight for a group of rows would be their spanning volume . . .
Randomised methods for solving $Ax = b$: Randomised Newton-Schulz

Classical Newton-Schulz for computing $A^{-1}$: With $A \in \mathbb{R}^{n \times n}$, we build $B = A^{-1}$ through an iterative scheme. Given an approximation $B_{\text{old}}$, the improved one is

$$B_{\text{new}} = B_{\text{old}} - AB_{\text{old}}A.$$  

Converges rapidly from a good initial guess. But basin of convergence is not large.

Gower & Richtarik (2019): Find $B = A^{-1}$ by solving the equation

(1)  

$$A^* = A^*AB.$$  

Equation (1) is solved through sketching + iteration: Draw an $m \times \ell$ random map $X$

$$B_{\text{new}} = \arg\min\{\|M - B_{\text{old}}\| : M \text{ satisfies } X^*A^* = X^*A^*AM\}.$$  

Equivalent to iteration

$$B_{\text{new}} = B_{\text{old}} - A^*AX \left(X^*A^*AA^*AX\right)^\dagger X^*A^* (AB_{\text{old}} - I).$$  

Detailed error analysis exists. For instance:

The expectation of the error converges exponentially fast, regardless of starting point.
Randomised iterative solvers is a very active area: Recent and current work by H. Avron, P. Drineas, L.-H. Lim, M. Mahoney, D. Needell, V. Rokhlin, S. Toledo, J. Tropp, R. Ward, J. Weare, and many more.
Randomised methods for solving $Ax = b$: Graph Laplacians

Let us consider a linear system

$$Ax = b$$

involving a coefficient matrix that is a graph Laplacian with $n$ nodes and $m$ edges.

- $A = A^* \in \mathbb{R}^{n \times n}$.
- $A(i, j) \leq 0$ when $i \neq j$.
- $A(i, i) = -\sum_{j \neq i} A(i, j)$

We assume that the underlying graph is connected, in which case $A$ has a 1-dimensional nullspace. We enforce that $\sum_i x(i) = 0$ and $\sum_i b(i) = 0$ in everything that follows.

(a) A graph with $n = 5$ vertices, and $m = 6$ edges. The conductivities of each edge is marked with a Greek letter.

(b) The $5 \times 5$ graph Laplacian matrix associated with the graph shown in (a).
Randomised methods for solving $Ax = b$: Graph Laplacians

Let us consider a linear system

$$Ax = b$$

involving a coefficient matrix that is a *graph Laplacian* with $n$ nodes and $m$ edges.

**Standard solution techniques:**

- **Multigrid:** Works great for certain classes of matrices.

- **Cholesky:** Compute a decomposition

  $$A = CC^*,$$

  with $C$ lower triangular. Always works. Numerically stable (when pivoting is used). Can be expensive since the factor $C$ typically has far more non-zero entries than $A$.

- **Incomplete Cholesky:** Compute an approximate factorisation

  $$A \approx CC^*,$$

  where $C$ is constrained to be as sparse as $A$ (typically the same pattern). Then use CG to solve a system with the preconditioned coefficient matrix $C^{-1}AC^{-*}$. Can work very well, hard to analyze.
Randomised methods for solving $Ax = b$: Graph Laplacians

Let us consider a linear system

$$Ax = b$$

involving a coefficient matrix that is a graph Laplacian with $n$ nodes and $m$ edges.

**Randomised solution techniques:**

- **Spielman-Teng (2004):** Complexity $O(m \text{ poly}(\log n) \log(1/\varepsilon))$.
  
  Relies on graph theoretical constructs (low-stretch trees, graph sparsification, explicit expander graphs, ...). Important theoretical results.

- **Kyng-Lee-Sachdeva-Spielman (2016):** $O(m (\log n)^2)$.
  
  Relies on local sampling only. Much closer to a realistic algorithm.

The idea is to build an approximate sparse Cholesky factor that is accurate with high probability. For instance, the 2016 paper proposes to build factors for which

$$\frac{1}{2}A \preccurlyeq CC^* \preccurlyeq \frac{3}{2}A.$$

When this bound holds, CG converges as $O(\gamma^n)$ with $\gamma = \frac{\sqrt{3}-1}{\sqrt{3}+1} \approx 0.27$.

Sparsity is maintained by performing inexact rank-1 updates in the Cholesky procedure. As a group of edges in the graph is removed, a set of randomly drawn new edges are added, in a way that is correct in expectation.
Randomised methods for solving $Ax = b$: “Rank structured” matrices

Many matrices in applications have off-diagonal blocks that are of low rank:

- Matrices approximating integral equations associated with elliptic PDEs. (Essentially, discretized Calderón-Zygmund operators.)
- Scattering matrices in acoustic and electro-magnetic scattering.
- Inverses of (sparse) matrices arising upon FEM discretization of elliptic PDEs.
- Buzzwords: $\mathcal{H}$-matrices, HSS-matrices, HBS matrices, ... 

Using randomised algorithms, we have developed $O(N)$-complexity methods for performing algebraic operations on dense matrices of this type. This leads to:

- Accelerated direct solvers for elliptic PDEs.
- $O(N)$ complexity in many situations.

A representative tessellation of a rank-structured matrix. Each off-diagonal block (gray) has low numerical rank. The diagonal blocks (red) are full rank, but are small in size. Matrices of this type allow efficient matrix-vector multiplication, matrix inversion, etc.
Randomised methods for solving $Ax = b$: “Rank structured” matrices

Let $A$ be a rank-structured matrix, for which we can rapidly evaluate $x \mapsto Ax$ and $x \mapsto A^*x$. There exist two classes of randomised algorithms for “compressing” $A$:

**Case 1:** Suppose that in addition to matvec, we can also evaluate individual entries of $A$. Then an HBS (a.k.a. HSS) representation can be computed in $O(N)$ operations. Very computationally efficient in practice — requires only $O(k)$ matvecs.

- Later improvements by Jianlin Xia, Sherry Li, etc.

**Case 2:** If all we have is the matvec, then we can still compute a rank-structured representation of $A$ using so called “peeling” algorithms. The price we have to pay is that we now need $O(k \times \log N)$ matvecs involving $A$ and $A^*$.

The method is still fast in many situations, and does save messy coding work. For instance, without this black-box method, implementing the matrix-matrix multiplication, or changing the partition tree, are quite hard to implement efficiently.

An example from data science: Kernel ridge regression

The matrices we represent using rank-structured formats are typically *kernel matrices*, which is to say that their entries can be written as

\[ A(i, j) = k(x_i, x_j) \]

for some set of points \( \{x_i\}_{i=1}^n \) in \( \mathbb{R}^d \).

The methods described are designed for problems in scientific computing where the dimension \( d \) is moderate. (Say \( d < 4 \).)

In data science, kernel matrices arise for point sets in much higher dimensions. For such problems, an approach based on *sampling* is often necessary. ("Sketch-to-solve" rather than "sketch-to-precondition".)
An example from data science: Kernel ridge regression

Task: We are given a set of pairs \( \{ \mathbf{x}_i, y_i \}_{i=1}^n \) where \( \mathbf{x}_i \in \mathbb{R}^d \) are data points, and where \( y_i \) are corresponding labels. We seek to build a function \( f : \mathbb{R}^d \to \mathbb{R} \) such that

\[
y_i \approx f(\mathbf{x}_i)
\]

for every point in the training set. The objective is to predict the label for any new unseen data point \( \mathbf{x} \).

Methodology: Let \( k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \) be a kernel function that measures how similar a pair of points are, scaled so that

\[
k(\mathbf{x}, \mathbf{y}) \approx 1 \quad \text{means } \mathbf{x} \text{ and } \mathbf{y} \text{ are similar},
\]

\[
k(\mathbf{x}, \mathbf{y}) \approx 0 \quad \text{means } \mathbf{x} \text{ and } \mathbf{y} \text{ are uncorrelated}.
\]

It is the job of the modeler to provide a “good” kernel function.

We then approximate \( f \) using the formula

\[
f(\mathbf{x}) = \sum_{i=1}^n k(\mathbf{x}, \mathbf{x}_i) \alpha_i,
\]

where the weights \( \{ \alpha_i \}_{i=1}^n \) are computed using the formula

\[
\alpha = (K + \lambda n I)^{-1} y,
\]

where \( K \) is the \( n \times n \) matrix with entries \( k(\mathbf{x}_i, \mathbf{x}_i) \). The number \( \lambda \) is a regularization parameter.

Challenge: \( K \) is very large, and computing an individual entry can be expensive.

Randomized solution: Draw an in index vector \( J \subset \{1, 2, \ldots, n\} \) holding \( k \) sampled indices, and replace \( K \) by the formula

\[
K_{\text{approx}} = K(:, J) K(J, J)^\dagger K(J, ::).
\]
Key points:

- Randomised low-rank approximation ("randomised SVD").
  - Superior performance in many regards, in particular for very large problems.
  - For a fixed number of matrix-vector multiplies, Krylov methods are more accurate.

- Essential benefit of randomisation in linear algebra: \textit{Reduces communication}.
  - Enables processing of huge data sets. (Out-of-core / streaming / cloud computing / …)
  - Very fast on GPUs, distributed memory machines, etc.

- There is exciting ongoing work on randomised methods for solving $Ax = b$.
  - Acceleration of existing $O(n^3)$ solvers — work very well, recommended without caveats.
  - Randomized preconditioners — currently work very well in some environments.
  - Two quite different methodologies:
    \begin{itemize}
    \item \textit{Sketch-to-precondition:} Safe, highly recommended.
    \item \textit{Sketch-to-solve:} Enables solvers for otherwise inaccessible problems.
    \end{itemize}

- Even though the algorithms are randomised, \textit{the output can be trusted}.
  - The probability of failure can be made \textit{extremely} low (say $10^{-10}$).
  - In most situations, you can explicitly compute the residual error.
  - Cf. \textit{Monte Carlo} vs. \textit{Las Vegas} methods.
Future and ongoing work:

1. **Accelerate full factorisations of matrices.**
   New randomised column pivoted QR algorithm is much faster than LAPACK.
   New “UTV” factorisation method is almost as accurate as SVD and much faster.

2. **Randomised algorithms for structured matrices.**
   Use randomisation to accelerate key numerical solvers for PDEs, for simulating
   Gaussian processes, etc.

3. **[High risk/high reward] Accelerate linear solvers for “general” systems $\mathbf{Ax} = \mathbf{b}$.**
   The goal is methods with complexity $O(n^\gamma)$ for $\gamma < 3$. Crucially, we seek methods
   that retain stability, and have high practical efficiency for realistic problem sizes.

4. **Use randomised projections to accelerate non-linear algebraic tasks.**
   Faster nearest neighbor search, faster clustering algorithms, etc. The idea is to use
   randomised projections for *sketching* to develop a rough map of a large data set.
   Then use high-accuracy deterministic methods for the actual computation.
Future and ongoing work:

1. **Accelerate full factorisations of matrices.**
   
   New randomised column pivoted QR algorithm is much faster than LAPACK. New “UTV” factorisation method is almost as accurate as SVD and much faster.

2. **Randomised algorithms for structured matrices.**
   
   Use randomisation to accelerate key numerical solvers for PDEs, for simulating Gaussian processes, etc.

3. **[High risk/high reward] Accelerate linear solvers for “general” systems \( \mathbf{A} \mathbf{x} = \mathbf{b} \).**
   
   The goal is methods with complexity \( O(n^\gamma) \) for \( \gamma < 3 \). Crucially, we seek methods that retain stability, and have high practical efficiency for realistic problem sizes.

4. **Use randomised projections to accelerate non-linear algebraic tasks.**
   
   Faster nearest neighbor search, faster clustering algorithms, etc. The idea is to use randomised projections for *sketching* to develop a rough map of a large data set. Then use high-accuracy deterministic methods for the actual computation.

Great potential for new discoveries in linear algebra!
Papers (see also http://users.oden.utexas.edu/~pgm/main_publications.html):


Tutorials, summer schools, etc:

- 2014: CBMS summer school at Dartmouth College. 10 lectures on YouTube.

Software packages:

- Column pivoted QR: https://github.com/flame/hqrrp *(much faster than LAPACK!)*
- Randomized UTV: https://github.com/flame/randutv
- RSVDPACK: https://github.com/sergeyvoronin
- ID: http://tygert.com/software.html
Fast Direct Solvers for Elliptic PDEs

PER-GUNNAR MARTINSSON
The University of Texas at Austin