Accelerating direct solvers using randomized methods

Gunnar Martinsson

The University of Colorado at Boulder

Collaborators: Robert van de Geijn, Adrianna Gillman, Nathan Heavner (PhD student), Grigorio Quintana-Ortí, Sergey Voronin (former postdoc).

Slides: Google “Gunnar Martinsson,” then go to “Talks” tab.

Research support by:

Synopsis: The talk will describe techniques for computing a low-rank approximation to a *dense* matrix through the use of randomized projections.

Context: Recent work on “Fast Direct Solvers” that exploit low-rank structures in otherwise dense matrices → “frontal matrices” in nested dissection, discretized Boundary Integral Operators, Dirichlet-to-Neumann operators, etc.

Environment 1: Given a dense $m \times n$ matrix $\mathbf{A}$ (whose singular values decay), compute an approximate factorization

$$\begin{array}{cccc} \mathbf{A} & \approx & \mathbf{Q} & \mathbf{B} \\ m \times n & & m \times k & k \times n \end{array}$$

where $k \ll \min(min)$. Typically, we are given a tolerance and need to determine k.

Environment 2: Given a dense $m \times n$ matrix $\mathbf{A}$ (with $m \geq n$) compute QR factorization

$$\begin{array}{ccccc} \mathbf{A} & \mathbf{P} & \approx & \mathbf{Q} & \mathbf{R} \\ m \times n & n \times n & & m \times k & k \times n \end{array}$$

for either $k = n$ (full factorization) or k comparable to $\min(m, n)$.

Environment 3: Given a *rank-structured* matrix $\mathbf{A}$ (HSS, HBS, HODLR, etc), compute a *data-sparse* representation of it.

Theme: Improve efficiency via *blocking* and *reducing communication.*

Environment 1 — low rank approximation of matrices

Let **A** denote a given $m \times n$ matrix. (We implicitly assume that its singular values decay, so that low-rank approximation makes sense.) Let $\varepsilon > 0$ be a given tolerance.

We then seek factors **Q** and **B** such that

$$\underset{m \times n}{\mathbf{A}} \;=\; \underset{m \times k}{\mathbf{Q}}\;\underset{k \times n}{\mathbf{B}} \;+\; \underset{m \times n}{\mathbf{E}},$$

where the error **E** satisfies

$$\|\mathbf{E}\| \leq \varepsilon.$$

We typically also require that **Q** has orthonormal columns.

- Determining a reasonably optimal rank k is part of the problem.
  We assume that k is substantially smaller than $\min(m, n)$.
- Standard software (LAPACK, Matlab, etc) lack built-in functionality for these tasks.
- Objective: Build “shell” algorithms that draw on BLAS3, LAPACK, etc, to solve the low-rank approximation problems efficiently.
- Simple post-processing of the small factor **B** allows the computation of approximate SVD $\mathbf{A} \approx \mathbf{U}_k\,\mathbf{D}_k\,\mathbf{V}_k^*$, and other standard factorizations.

Environment 1 — low rank approximation of matrices

Let **A** denote a given $m \times n$ matrix. (We implicitly assume that its singular values decay, so that low-rank approximation makes sense.) Let $\varepsilon > 0$ be a given tolerance.

We then seek factors **Q** and **B** such that

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times k}{\mathbf{Q}} \; \underset{k \times n}{\mathbf{B}} + \underset{m \times n}{\mathbf{E}},$$

where the error **E** satisfies

$$\|\mathbf{E}\| \leq \varepsilon.$$

We typically also require that **Q** has orthonormal columns.

References:

- P.G. Martinsson, V. Rokhlin, and M. Tygert, “A randomized algorithm for the decomposition of matrices”. *Applied and Computational Harmonic Analysis*, **30**(1), pp. 47–68, 2011.
  Based on 2006 Yale CS research report YALEU/DCS/RR-1361.
- N. Halko, P.G. Martinsson, J. Tropp, “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions.” *SIAM Review*, **53**(2), pp. 217–288, 2011.
- P.G. Martinsson and S. Voronin, “A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices.” *SIAM J. on Scientific Comp.*, **38**(5), S485 – S507, 2016.

An algorithmic template

We build a basis $\{\mathbf{q}_j\}_{j=1}^{k}$ for the column space of $\mathbf{A}$, using a “greedy” algorithm:

```
(1)  $\mathbf{Q}_0 = [\,];\ \mathbf{B}_0 = [\,];\ \mathbf{A}_0 = \mathbf{A};\ j = 0;$
(2)  while $\|\mathbf{A}_j\| > \varepsilon$
(3)      $j = j + 1$
(4)      Pick a unit vector $\mathbf{q}_j \in \mathrm{ran}(\mathbf{A}_{j-1})$.
(5)      $\mathbf{b}_j = \mathbf{q}_j^{*}\mathbf{A}_{j-1}$
(6)      $\mathbf{Q}_j = [\mathbf{Q}_{j-1}\ \mathbf{q}_j]$
(7)      $\mathbf{B}_j = \begin{bmatrix} \mathbf{B}_{j-1} \\ \mathbf{b}_j \end{bmatrix}$
(8)      $\mathbf{A}_j = \mathbf{A}_{j-1} - \mathbf{q}_j\mathbf{b}_j$
(9)  end while
```

Simple condition: On line (4), pick $\mathbf{q}_j$ as the largest column of $\mathbf{A}_{j-1}$.

Then we recover column-pivoted Gram-Schmidt, which is often an excellent algorithm.

(Round-off errors make some minor modifications necessary; we will discuss this later.)

Problem 1: Hard to *block* efficiently. (Can be done, via, e.g. “tournament pivoting”.)

Problem 2: “Typically” gives reasonably close to optimal results, but can be quite bad.

An algorithmic template

We build a basis $\{\mathbf{q}_j\}_{j=1}^{k}$ for the column space of $\mathbf{A}$, using a “greedy” algorithm:

```
(1)  $\mathbf{Q}_0 = [\;]$; $\mathbf{B}_0 = [\;]$; $\mathbf{A}_0 = \mathbf{A}$; $j = 0$;
(2)  while $\|\mathbf{A}_j\| > \varepsilon$
(3)      $j = j + 1$
(4)      Pick a unit vector $\mathbf{q}_j \in \text{ran}(\mathbf{A}_{j-1})$.
(5)      $\mathbf{b}_j = \mathbf{q}_j^* \mathbf{A}_{j-1}$
(6)      $\mathbf{Q}_j = [\mathbf{Q}_{j-1}\ \mathbf{q}_j]$
(7)      $\mathbf{B}_j = \begin{bmatrix} \mathbf{B}_{j-1} \\ \mathbf{b}_j \end{bmatrix}$
(8)      $\mathbf{A}_j = \mathbf{A}_{j-1} - \mathbf{q}_j \mathbf{b}_j$
(9)  end while
```

Optimal condition: On line (4), pick $\mathbf{q}_j$ as a *minimizer* of

$$\min_{\|\mathbf{q}\|=1} \|\mathbf{A}_{j-1} - \mathbf{q}\mathbf{q}^* \mathbf{A}_{j-1}\|.$$

Problem: Computationally hard to find the minimizer.

An algorithmic template — now *randomized*

We build a basis $\{\mathbf{q}_j\}_{j=1}^{k}$ for the column space of $\mathbf{A}$, using a “greedy” algorithm:

```
(1)   $\mathbf{Q}_0 = [\,]$; $\mathbf{B}_0 = [\,]$; $\mathbf{A}_0 = \mathbf{A}$; $j = 0$;
(2)   while $\|\mathbf{A}_j\| > \varepsilon$
(3)       $j = j + 1$
(4a)      Draw a random vector $\omega$ whose entries are iid Gaussian random variables.
(4b)      Set $\mathbf{y} = \mathbf{A}_{j-1}\omega$.
(4c)      Normalize so that $\mathbf{q}_j = \frac{1}{\|\mathbf{y}\|}\,\mathbf{y}$.
(5)       $\mathbf{b}_j = \mathbf{q}_j^{*}\mathbf{A}_{j-1}$
(6)       $\mathbf{Q}_j = [\mathbf{Q}_{j-1}\ \mathbf{q}_j]$
(7)       $\mathbf{B}_j = \begin{bmatrix} \mathbf{B}_{j-1} \\ \mathbf{b}_j \end{bmatrix}$
(8)       $\mathbf{A}_j = \mathbf{A}_{j-1} - \mathbf{q}_j\mathbf{b}_j$
(9)   end while
```

Simple to implement.

Often reasonably close to optimal.

Very easy to block.

An algorithmic template — now randomized and *blocked*

Pick a “block size” ℓ.

```
(1)  $\mathbf{Q} = [\,];\ \mathbf{B} = [\,];$
(2)  while $\|\mathbf{A}\| > \varepsilon$
(3)      Draw an $n \times \ell$ random matrix $\mathbf{R}$.
(4)      Compute the $m \times \ell$ matrix $\mathbf{Q}_{\rm new} = \mathtt{qr}(\mathbf{AR}, 0)$.
(5)      $\mathbf{B}_{\rm new} = \mathbf{Q}_{\rm new}^{*}\mathbf{A}$
(6)      $\mathbf{Q} = [\mathbf{Q}\ \mathbf{Q}_{\rm new}]$
(7)      $\mathbf{B} = \begin{bmatrix} \mathbf{B} \\ \mathbf{B}_{\rm new} \end{bmatrix}$
(8)      $\mathbf{A} = \mathbf{A} - \mathbf{Q}_{\rm new}\mathbf{B}_{\rm new}$
(9)  end while
```

The scheme presented works very well for matrices whose singular values decay rapidly.

Note that every line can be executed by BLAS3, except (4) for which we use LAPACK.

When the singular values do not decay rapidly, we apply a *power* of **A**.

An algorithmic template — now randomized, blocked, and *accuracy-enhanced*

Pick a “block size” ℓ, and a small integer q, say $q = 1$, or $q = 2$.

```
$\mathbf{Q} = [\,]$; $\mathbf{B} = [\,]$;
while $\|\mathbf{A}\| > \varepsilon$
    Draw an $n \times \ell$ random matrix $\mathbf{R}$.
    Compute the $m \times \ell$ matrix $\mathbf{Q}_{\rm new} = \texttt{qr}((\mathbf{A}\mathbf{A}^{*})^{q}\mathbf{A}\mathbf{R}, 0)$.
    $\mathbf{B}_{\rm new} = \mathbf{Q}_{\rm new}^{*}\mathbf{A}$
    $\mathbf{Q} = [\mathbf{Q}\ \mathbf{Q}_{\rm new}]$
    $\mathbf{B} = \begin{bmatrix} \mathbf{B} \\ \mathbf{B}_{\rm new} \end{bmatrix}$
    $\mathbf{A} = \mathbf{A} - \mathbf{Q}_{\rm new}\mathbf{B}_{\rm new}$
end while
```

The only thing remaining is to deal with loss of orthonormality due to round-off errors.

An algorithmic template — now randomized, blocked, and accuracy-enhanced

Pick a "block size" ℓ, and a small integer q, say $q = 0$, $q = 1$, or $q = 2$.

```
THE "RAND-QB" ALGORITHM

(1)  $\mathbf{Q} = [\,]; \mathbf{B} = [\,];$
(2)  while $\|\mathbf{A}\| > \varepsilon$
(3)      Draw an $n \times \ell$ random matrix $\mathbf{R}$.
(4a)     Compute the $m \times \ell$ matrix $\mathbf{Y} = \mathtt{qr}((\mathbf{A}\mathbf{A}^*)^q\mathbf{A}\mathbf{R}, 0)$.
(4b)     Reproject $\mathbf{Y}$ away from the range of $\mathbf{Q}$: $\mathbf{Y} = \mathbf{Y} - \mathbf{Q}(\mathbf{Q}^*\mathbf{Y})$.
(4c)     Compute the $m \times \ell$ matrix $\mathbf{Q}_{\rm new} = \mathtt{qr}(\mathbf{Y}, 0)$.
(5)      $\mathbf{B}_{\rm new} = \mathbf{Q}_{\rm new}^*\mathbf{A}$
(6)      $\mathbf{Q} = [\mathbf{Q}\ \mathbf{Q}_{\rm new}]$
(7)      $\mathbf{B} = \begin{bmatrix} \mathbf{B} \\ \mathbf{B}_{\rm new} \end{bmatrix}$
(8)      $\mathbf{A} = \mathbf{A} - \mathbf{Q}_{\rm new}\mathbf{B}_{\rm new}$
(9)  end while
```

With minor modifications, we can avoid updating **A**. This is crucial for sparse matrices, and in situations where we can only access **A** via its action on vectors.

Given the "QB-factorization," standard factorizations can easily be computed

Suppose that you have an approximate factorization

$$\mathbf{A} = \mathbf{QB} + \mathbf{E},$$

where $\mathbf{Q}$ is orthonormal and $\|\mathbf{E}\|$ is small.

How to get an approximate SVD: Perform a full SVD of the small matrix $\mathbf{B}$:

$$[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd}(\mathbf{B}, \texttt{'econ'}).$$

Then simply set $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$ and you will get a partial SVD

$$\mathbf{A} = \mathbf{UDV}^* + \mathbf{E}.$$

Note that the error is exactly the same as the error in the QB.

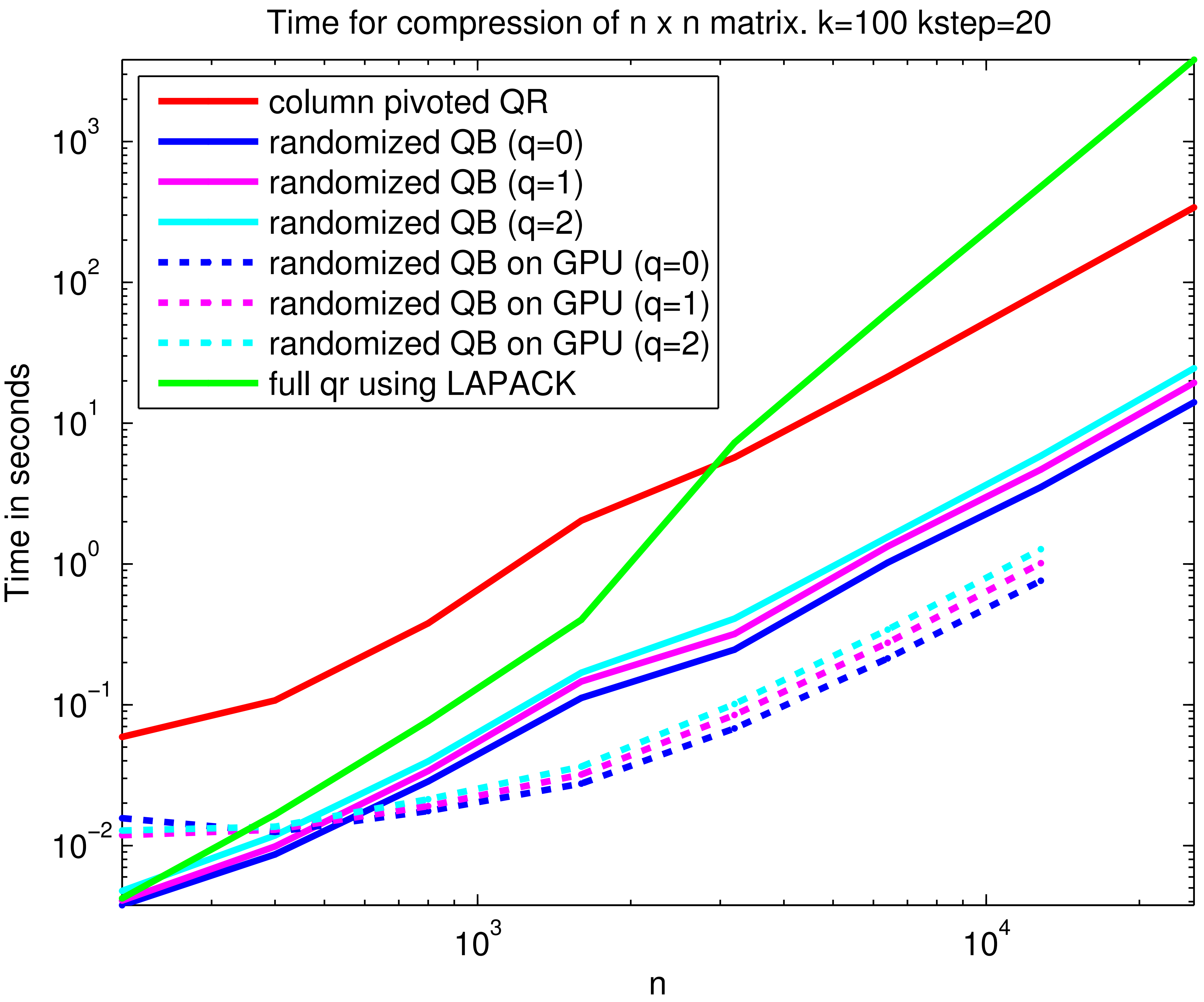
How to get an approximate QR: Perform a full QR of the small matrix $\mathbf{B}$:

$$[\hat{\mathbf{Q}}, \mathbf{R}, \mathbf{P}] = \texttt{qr}(\mathbf{B}, 0).$$

Then simply set $\tilde{\mathbf{Q}} = \mathbf{Q}\hat{\mathbf{Q}}$ and you will get a partial QR

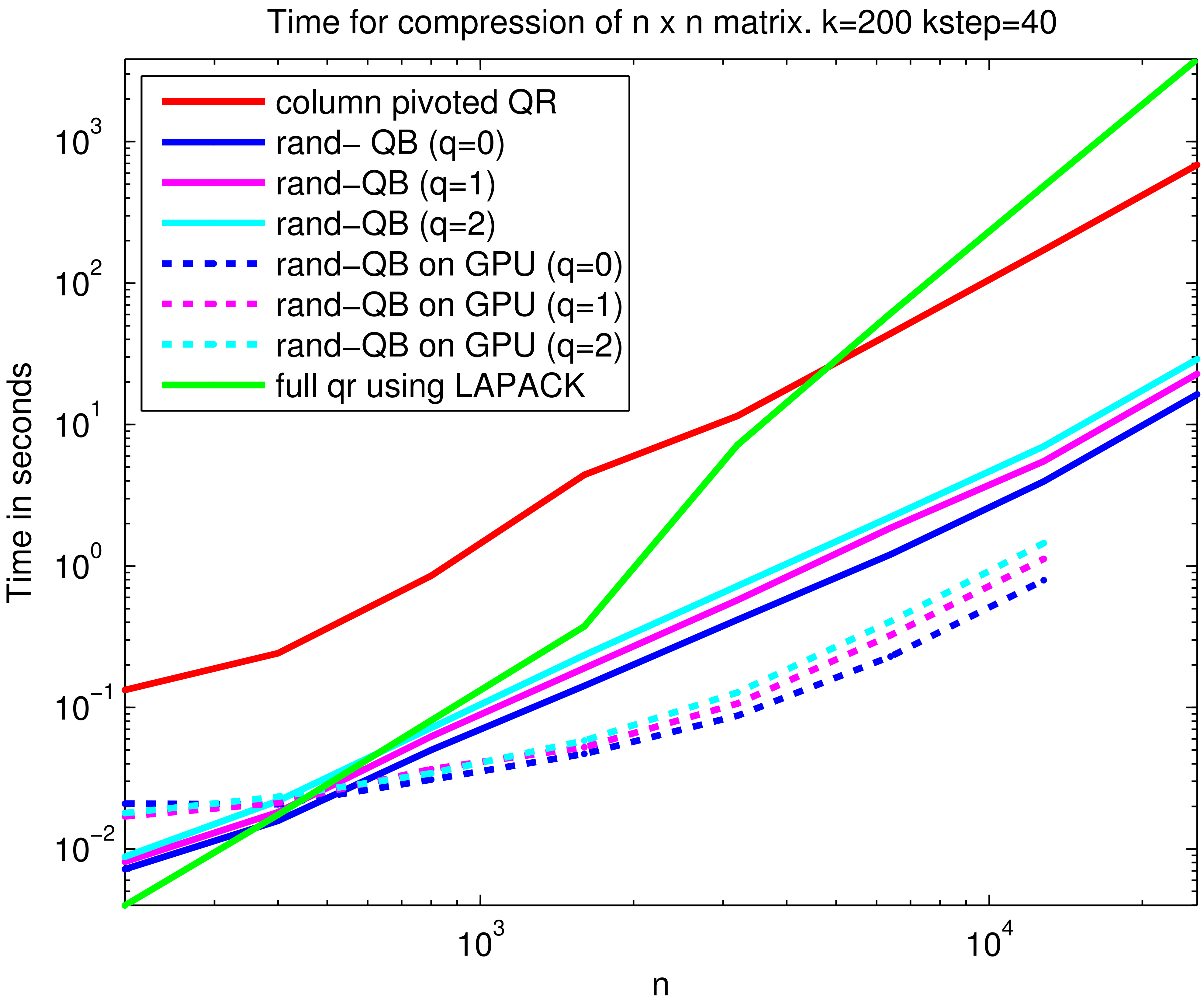
$$\mathbf{AP} = \tilde{\mathbf{Q}}\mathbf{R} + \mathbf{E}.$$

Again, the error is exactly the same as the error in the QB.



Everything is implemented in Matlab. The “full qr” line refers to Matlab built in qr.

Caveat: Matlab overhead makes column-pivoted QR slower than it could be.



Everything is implemented in Matlab. The “full qr” line refers to Matlab built in qr.

Caveat: Matlab overhead makes column-pivoted QR slower than it could be.

Example: Accuracy for synthetic matrix with rapidly decaying spectrum

Consider a matrix *defined* by its SVD

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times r & r \times r & r \times n \end{array}$$

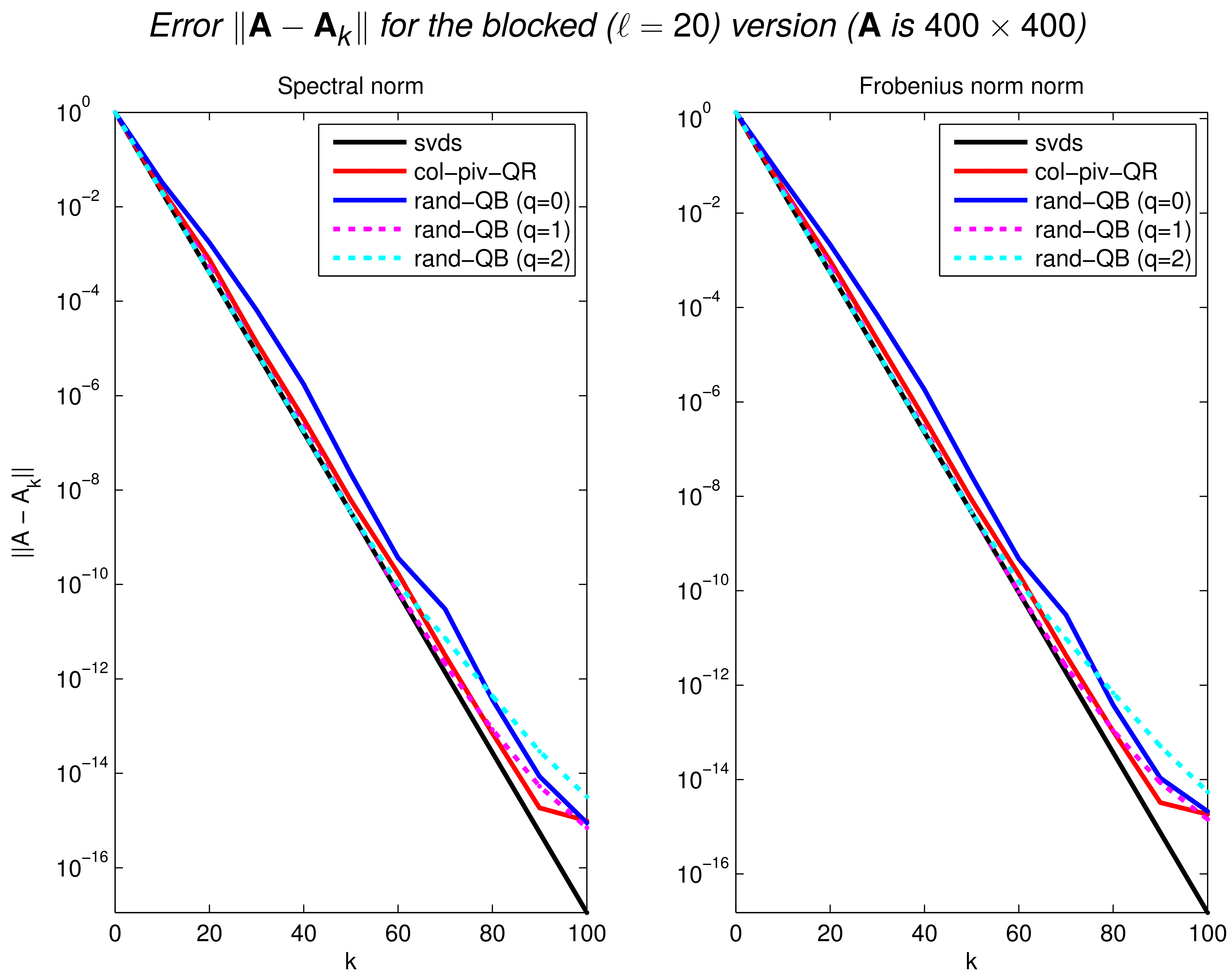
where $r = \min(m, n)$, where **U** and **V** are random orthonormal matrices, and

$$\mathbf{D} = \texttt{diag}(1, \alpha, \alpha^2, \alpha^3, \dots),$$

where α is chosen so that

$$\alpha^{90} = 10^{-15}.$$

In this example, $n = 400$.


Error ‖A − A_k‖ for the blocked (ℓ = 20) version (A is 400 × 400)
Spectral norm
Frobenius norm norm
svds
col−piv−QR
rand−QB (q=0)
rand−QB (q=1)
rand−QB (q=2)
||A − A_k||
10^0
10^−2
10^−4
10^−6
10^−8
10^−10
10^−12
10^−14
10^−16
0
20
40
60
80
100
k

Example: Accuracy for synthetic matrix with slowly decaying spectrum

Consider a matrix *defined* by its SVD

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times r & r \times r & r \times n \end{array}$$

where $r = \min(m, n)$, where **U** and **V** are random orthonormal matrices, and

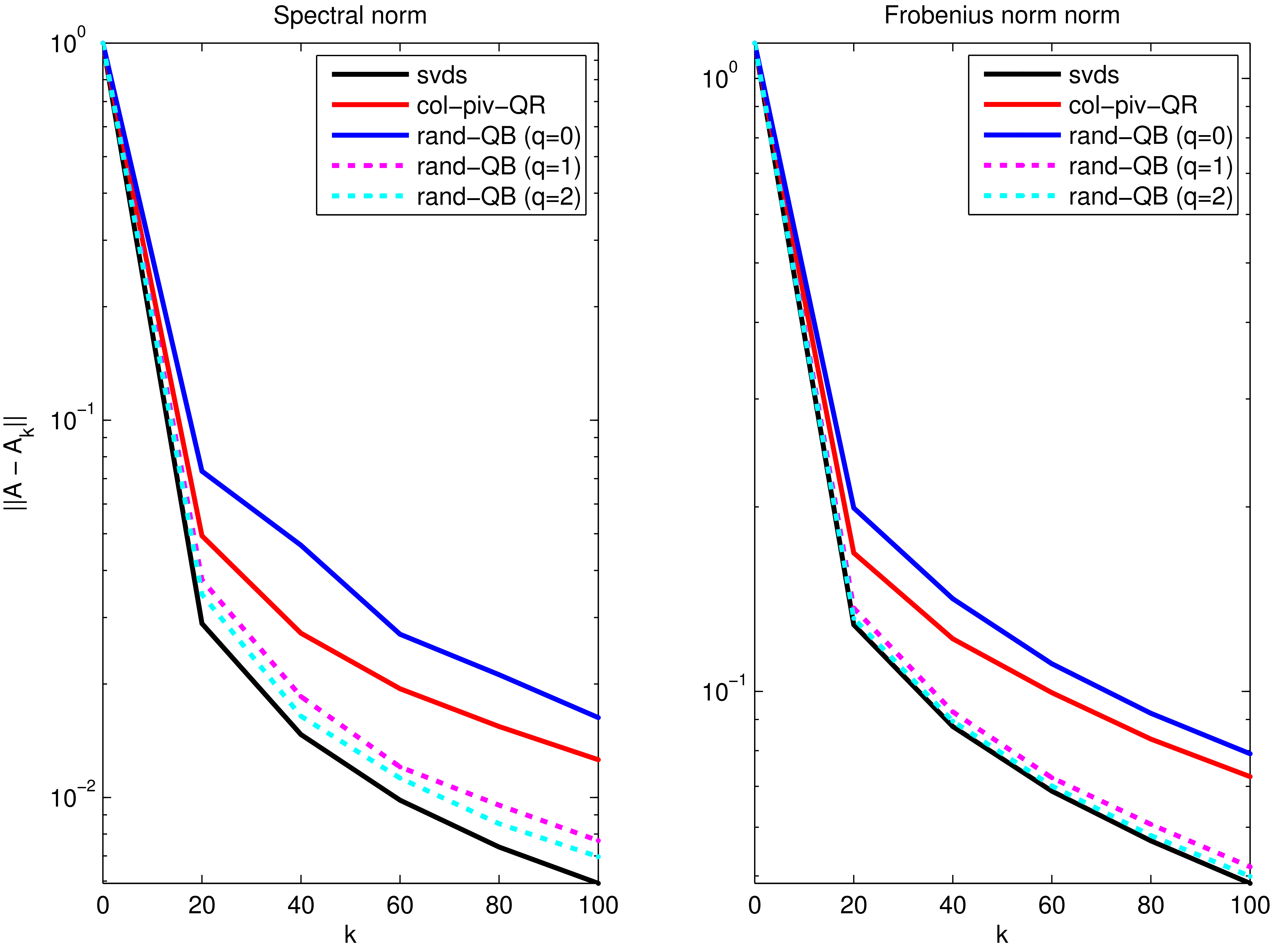
$$\mathbf{D} = \mathtt{diag}(\sigma_1,\, \sigma_2,\, \sigma_3,\, \dots),$$

where

$$\sigma_j = \frac{1}{\sqrt{1 + 3\,(j-1)}}.$$

In this example, $m = 500$ and $n = 300$.

Error $\|\mathbf{A} - \mathbf{A}_k\|$ *for the blocked* ($\ell = 20$) *version* ($\mathbf{A}$ *is* 500×300)



Randomized low-rank approximation of matrices

The methods described so far are *very* easy to implement. Can be done in Matlab, or in C/Fortran using standard subroutines (`dgemm`, `dgeqrf`).

Computational speed is good; in particular on GPUs.

The accuracy is very good. With two sweeps of power iteration ($q = 2$), it compares very favorably to column-pivoted QR, and is almost as good as SVD.

Randomized low-rank approximation of matrices

The methods described so far are *very* easy to implement. Can be done in Matlab, or in C/Fortran using standard subroutines (`dgemm`, `dgeqrf`).

Computational speed is good; in particular on GPUs.

The accuracy is very good. With two sweeps of power iteration ($q = 2$), it compares very favorably to column-pivoted QR, and is almost as good as SVD.

Caveat: This is efficient only when the rank k is small, $k \ll \min(m, n)$.

Question: Can we device a method that works well *for any rank?*

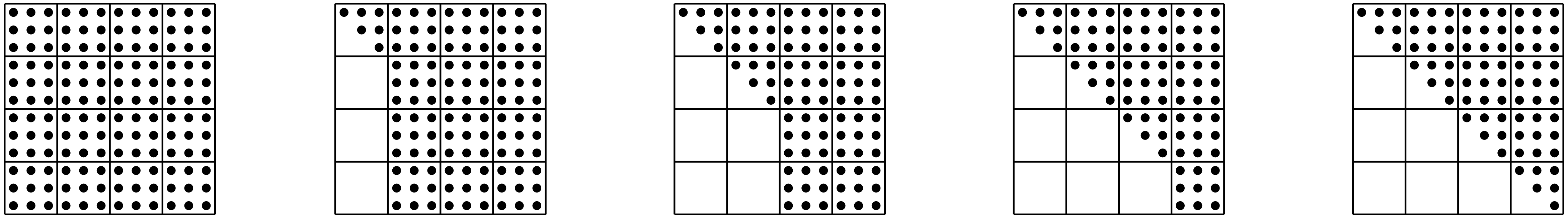
Environment 2: Computing a traditional QR factorization

Given a dense $m \times n$ matrix $\mathbf{A}$ (with $m \geq n$) compute QR (or RRQR)

$$\begin{array}{ccccc} \mathbf{A} & \mathbf{P} & \approx & \mathbf{Q} & \mathbf{R} \\ m \times n & n \times n & & m \times k & k \times n \end{array}$$

for either $k = n$ (full factorization) or k comparable to $\min(m, n)$. 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:



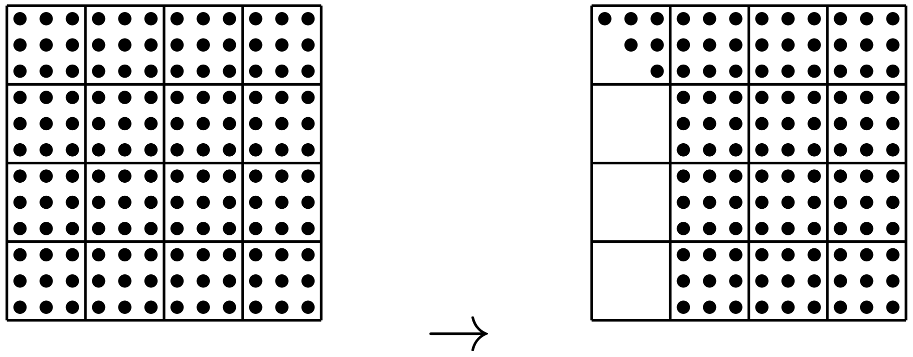
$\mathbf{A}_0 = \mathbf{A}$ $\quad$ $\mathbf{A}_1 = \mathbf{Q}_1^*\mathbf{A}_0\mathbf{P}_1$ $\quad$ $\mathbf{A}_2 = \mathbf{Q}_2^*\mathbf{A}_1\mathbf{P}_2$ $\quad$ $\mathbf{A}_3 = \mathbf{Q}_3^*\mathbf{A}_2\mathbf{P}_3$ $\quad$ $\mathbf{A}_4 = \mathbf{Q}_4^*\mathbf{A}_3\mathbf{P}_4$

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

Environment 2: Computing a traditional QR factorization

How to do block pivoting using randomization:

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



$$\mathbf{A} \qquad\qquad \mathbf{Q}^*\mathbf{AP}$$

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

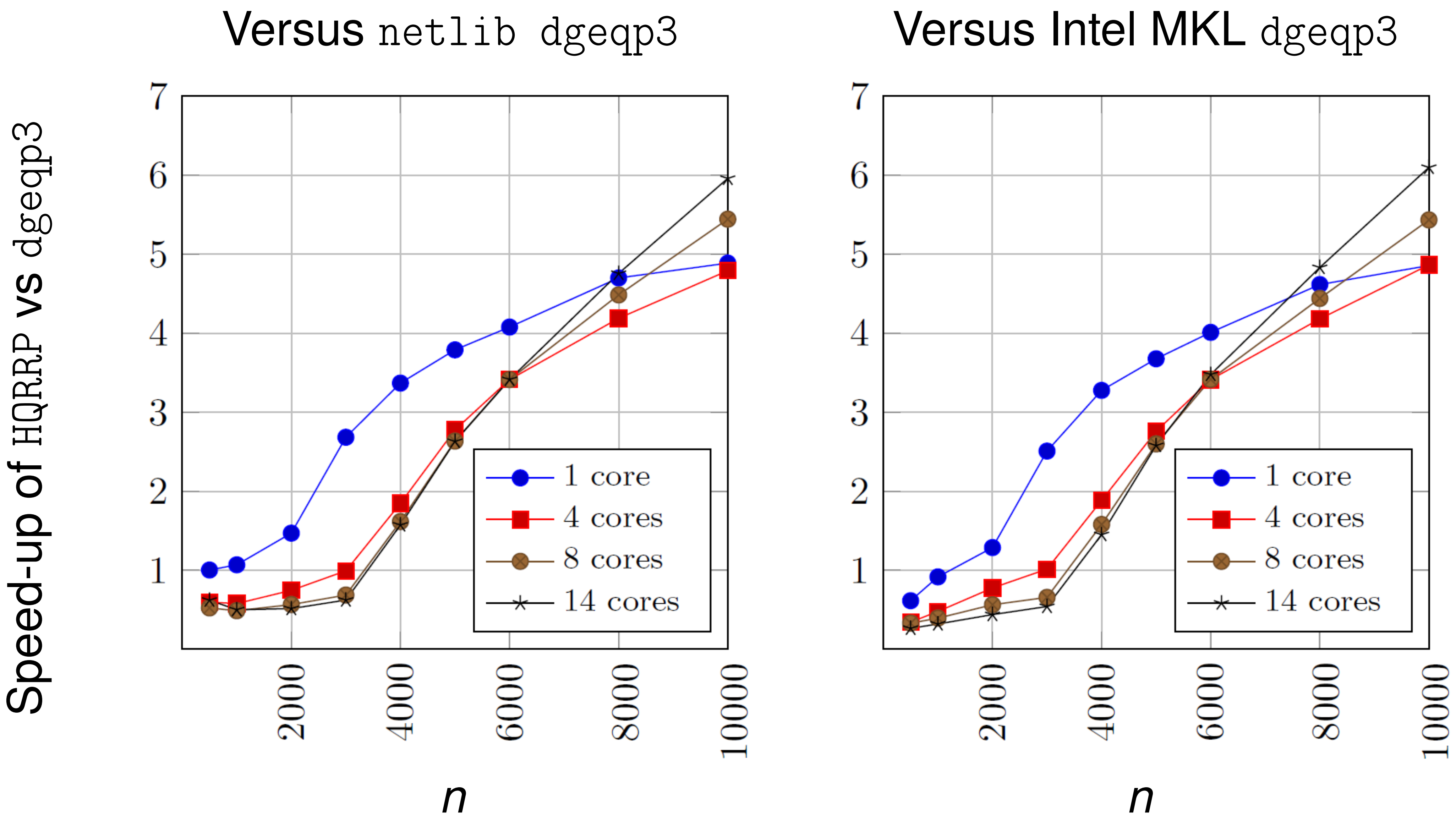
$$\begin{array}{cccc} \mathbf{Y} & = & \mathbf{G} & \mathbf{A} \\ b \times n & & b \times m & m \times n \end{array}$$

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**:

$$\begin{array}{ccccc} \mathbf{Y} & \mathbf{P} & = & \mathbf{Q}_{\rm trash} & \mathbf{R}_{\rm trash} \\ b \times n & n \times n & & b \times b & b \times n \end{array}$$

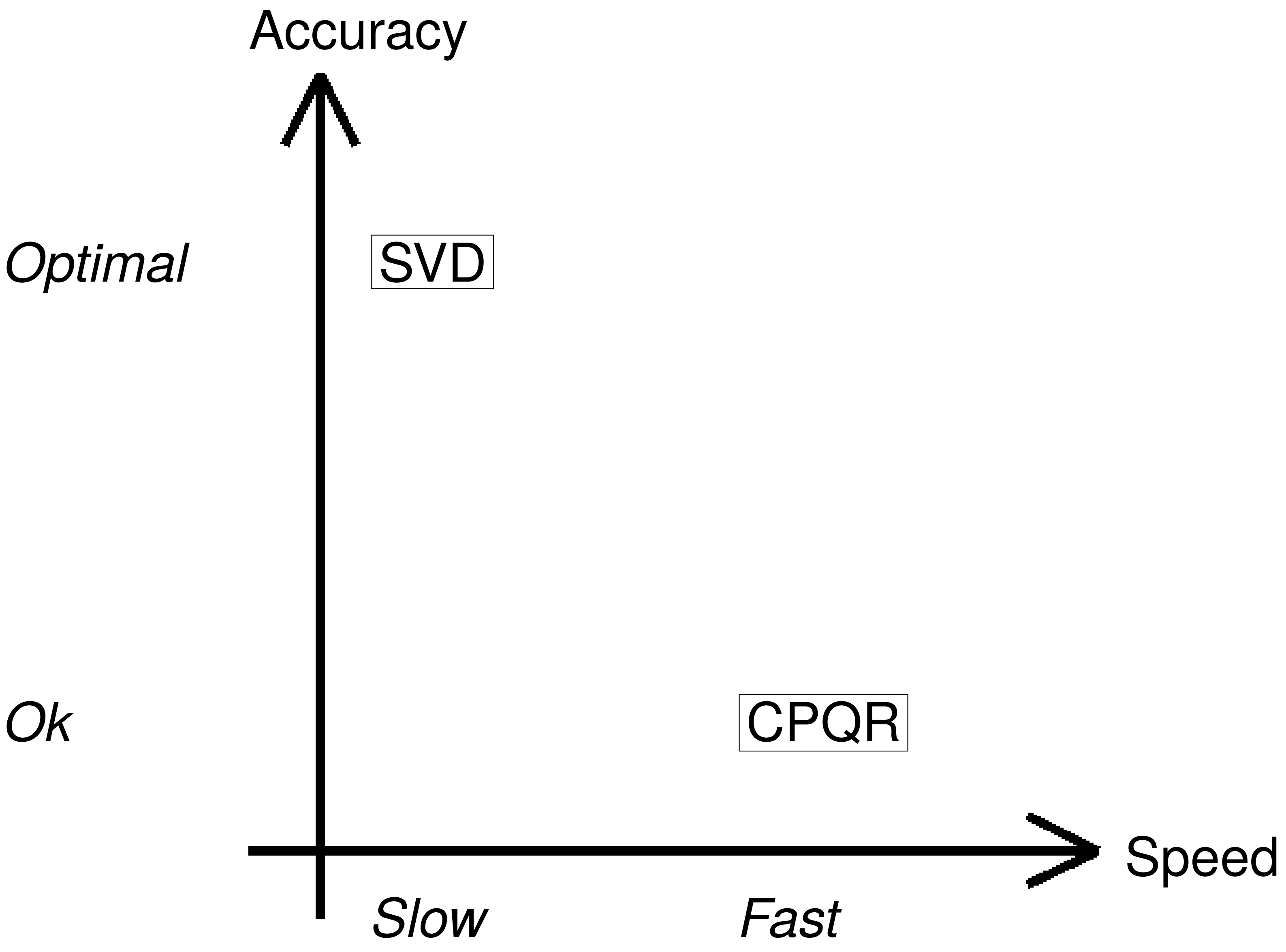
Environment 2: Computing a traditional QR factorization



Speedup attained by our randomized algorithm `HQRRP` *for computing a full column pivoted QR factorization of an $n \times n$ matrix. The speed-up is measured versus LAPACK's faster routine* `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/`

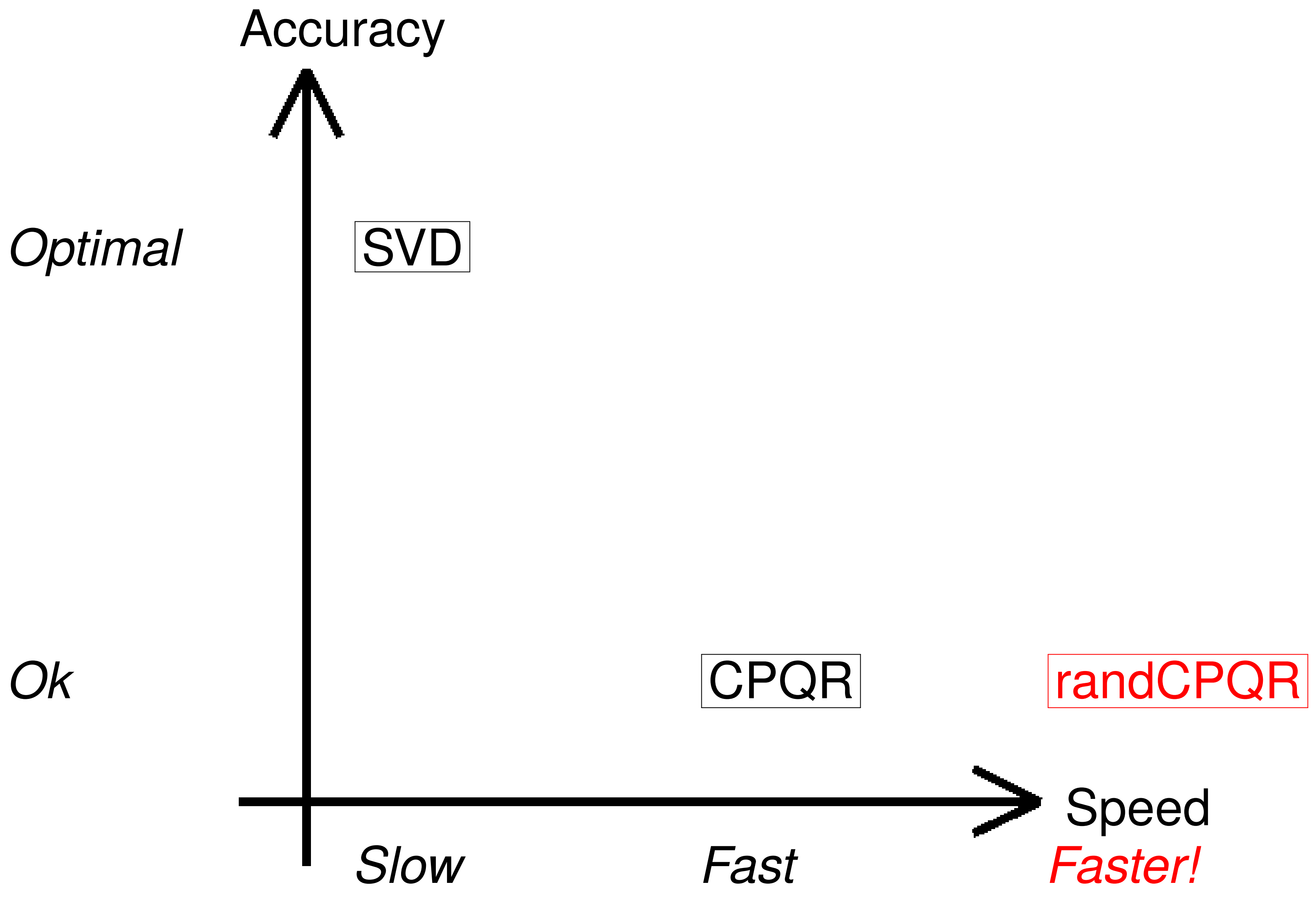
Environment 2: Computing a traditional QR factorization

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:



Environment 2: Computing a traditional QR factorization

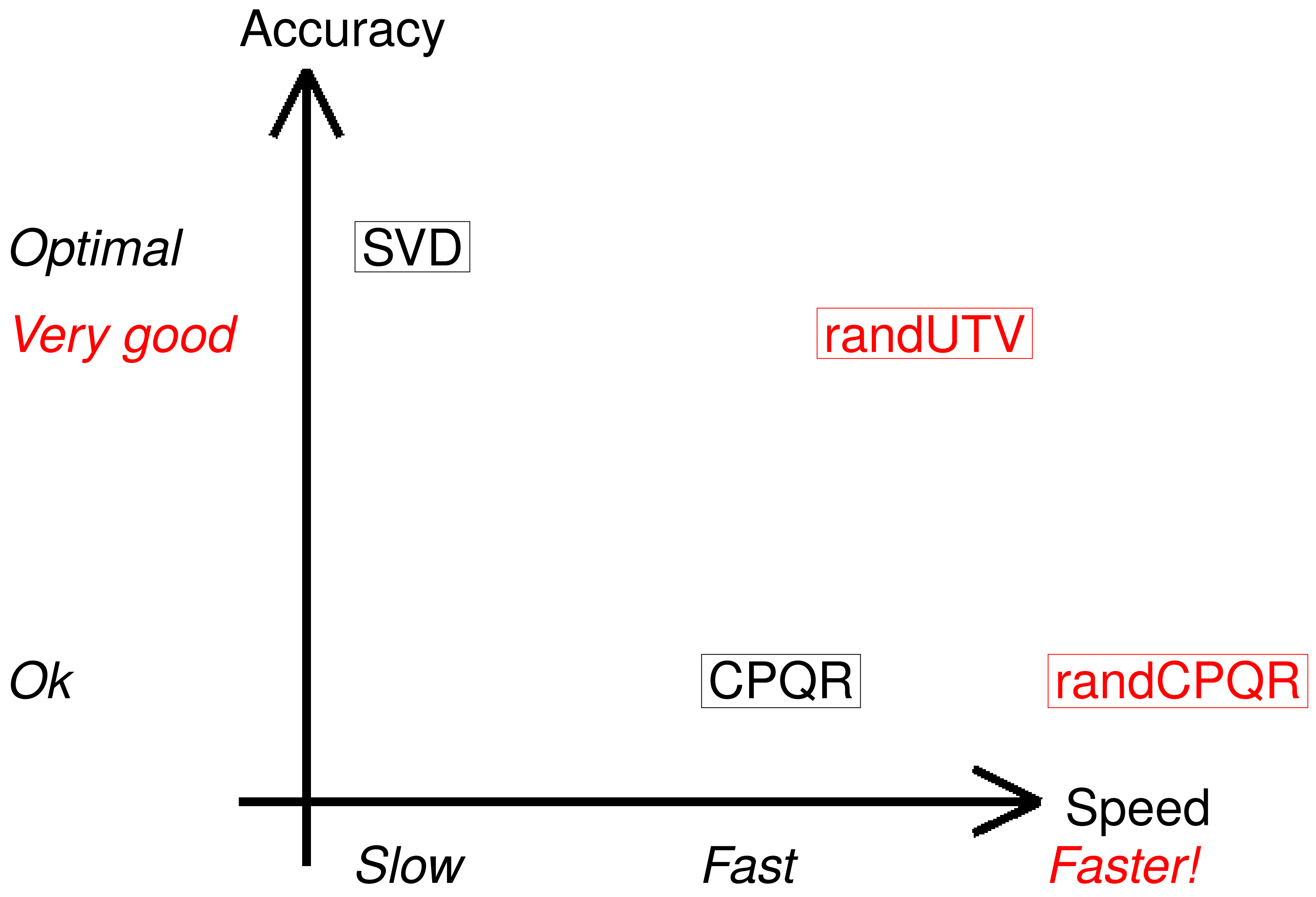
For the task of computing low-rank approximations to matrices, the classical choice is between SVD and column pivoted QR (CPQR). SVD is slow, and CPQR is inaccurate:



Randomized CPQR is faster than CPQR, but is no better in terms of accuracy.

Environment 2: Computing a traditional QR factorization

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:



Randomized CPQR is faster than CPQR, but is no better in terms of accuracy.

Randomized UTV is faster than CPQR, and attains very close to SVD accuracy!

Additionally, randUTV parallelizes well and supports partial factorization.

Environment 2: Accelerate FULL factorizations of matrices

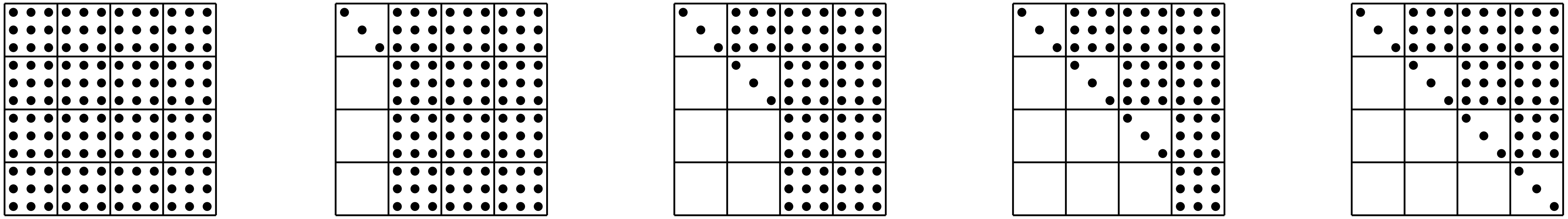
Given a dense $n \times n$ matrix $\mathbf{A}$, compute a factorization

$$\underset{n \times n}{\mathbf{A}} = \underset{n \times n}{\mathbf{U}} \; \underset{n \times n}{\mathbf{T}} \; \underset{n \times n}{\mathbf{V}^*},$$

where $\mathbf{T}$ is upper triangular, $\mathbf{U}$ and $\mathbf{V}$ are unitary.

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

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

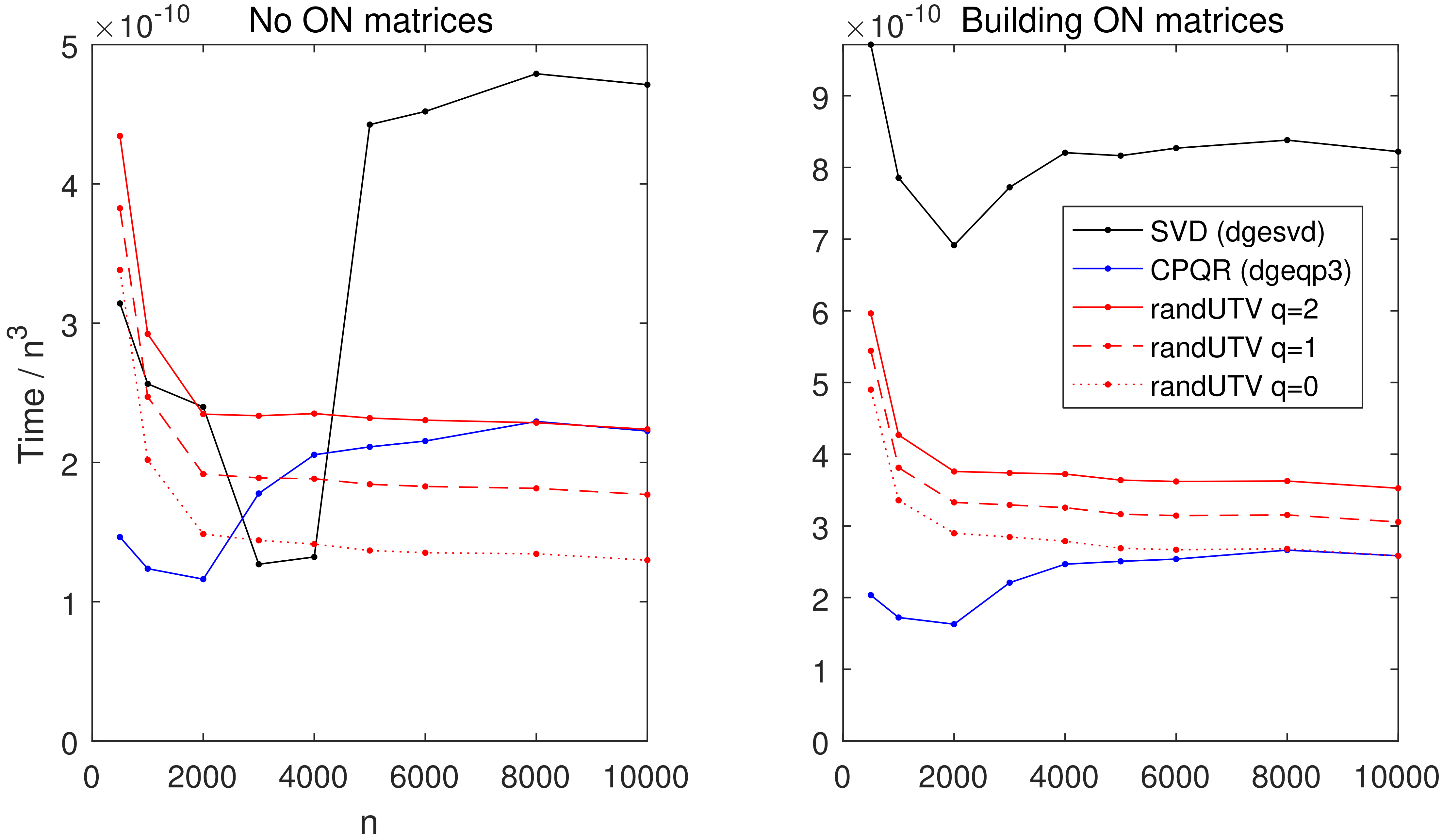


$\mathbf{A}_0 = \mathbf{A}$ $\quad$ $\mathbf{A}_1 = \mathbf{U}_1^*\mathbf{A}_0\mathbf{V}_1$ $\quad$ $\mathbf{A}_2 = \mathbf{U}_2^*\mathbf{A}_1\mathbf{V}_2$ $\quad$ $\mathbf{A}_3 = \mathbf{U}_3^*\mathbf{A}_2\mathbf{V}_3$ $\quad$ $\mathbf{A}_4 = \mathbf{U}_4^*\mathbf{A}_3\mathbf{V}_4$

Both $\mathbf{U}_j$ and $\mathbf{V}_j$ are (mostly...) products of b Householder reflectors.

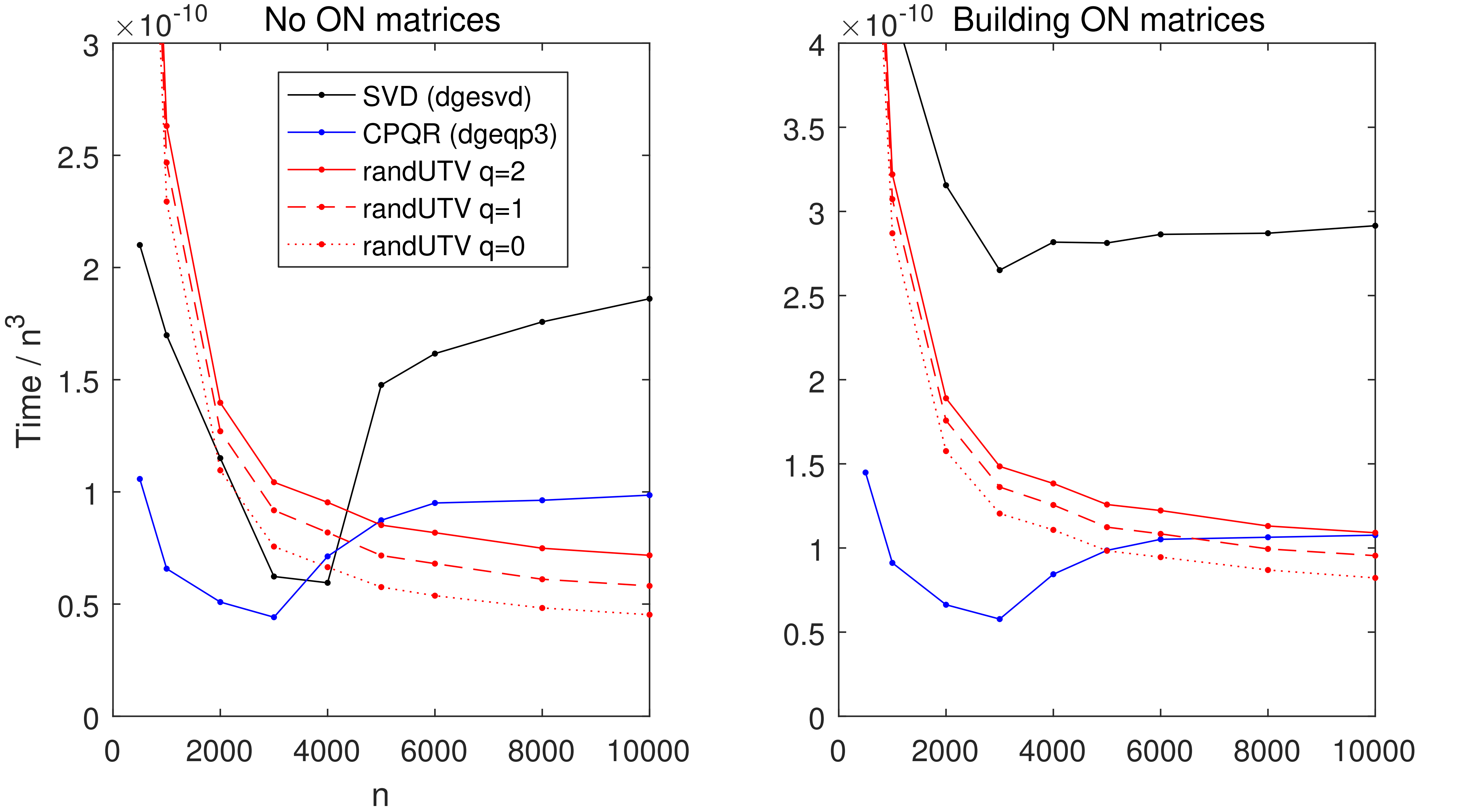
Our objective is in each step to find an approximation *to the linear subspace* spanned by the b dominant singular vectors of a matrix. The randomized range finder is perfect for this, especially when a small number of power iterations are performed. Easier and more natural than choosing pivoting vectors.

Computational speed of randUTV — 1 core



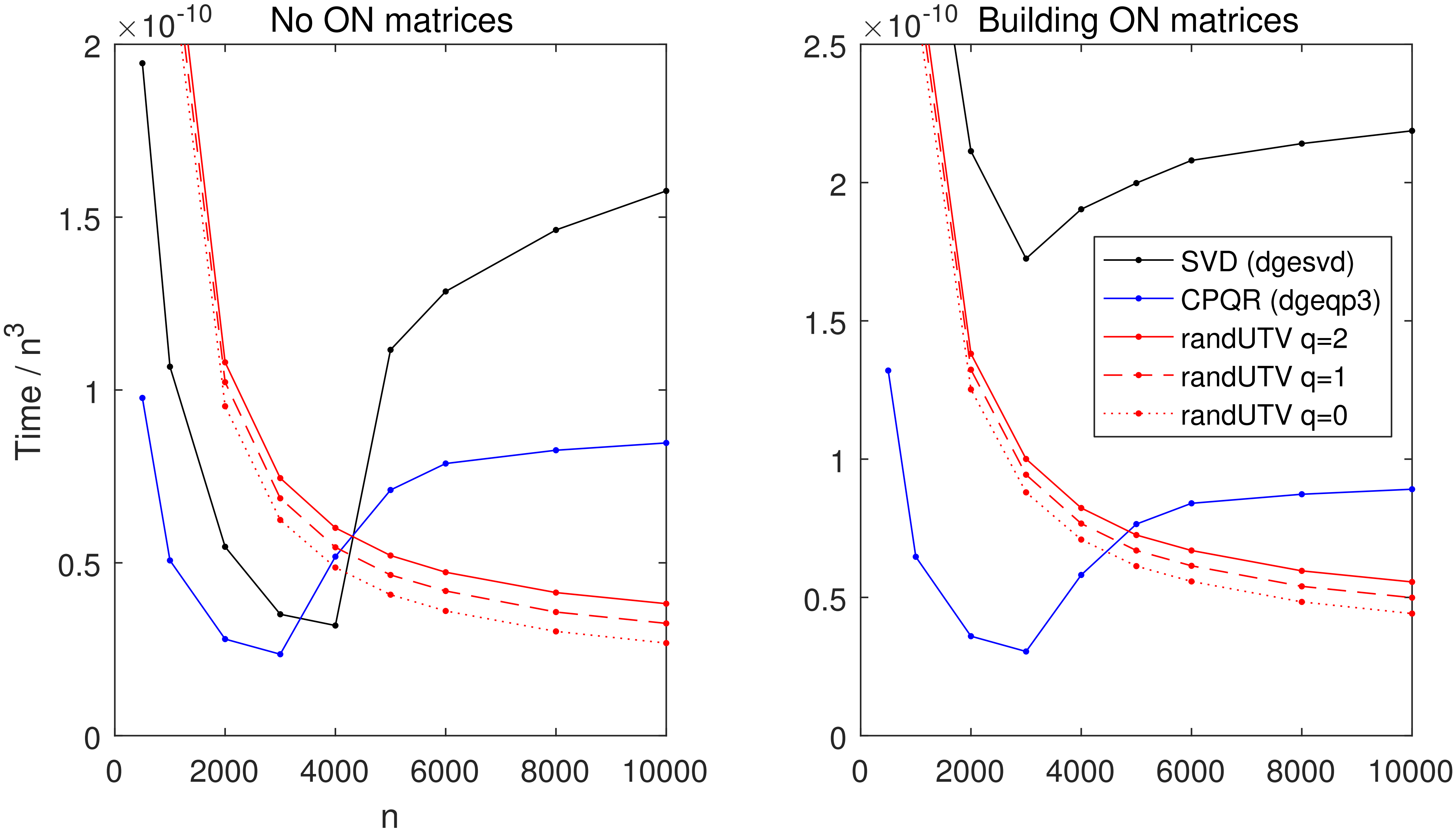
Computational speed of randUTV (red) compared to SVD (black) and column pivoted QR (blue). Each run is for a full factorization of an $n \times n$ matrix. We compare against Intel MKL LAPACK. Observe that the vertical axis is execution time scaled by n^3!

Computational speed of randUTV — 4 cores

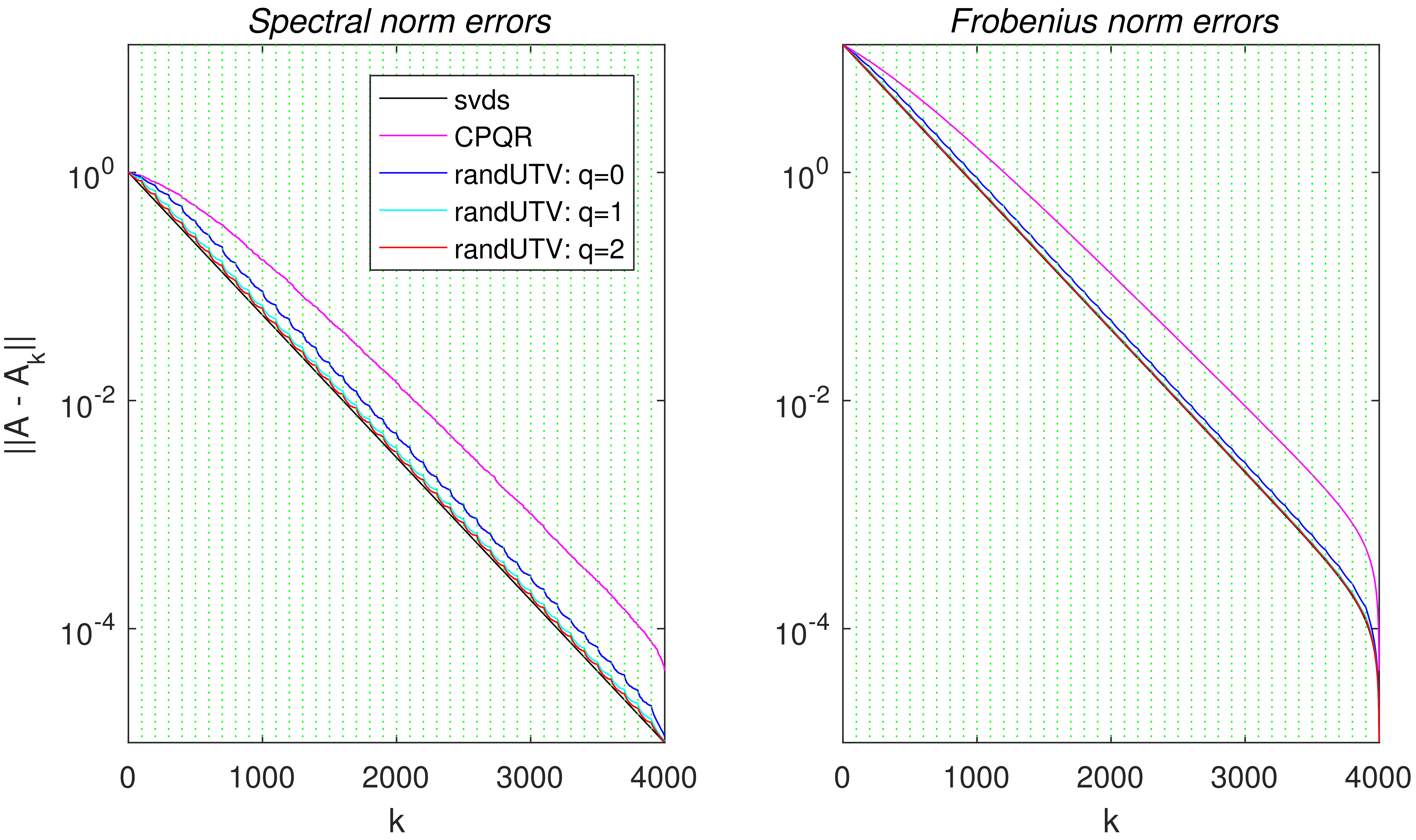


Computational speed of randUTV (red) compared to SVD (black) and column pivoted QR (blue). Each run is for a full factorization of an $n \times n$ matrix. We compare against Intel MKL LAPACK. Observe that the vertical axis is execution time scaled by n^3!

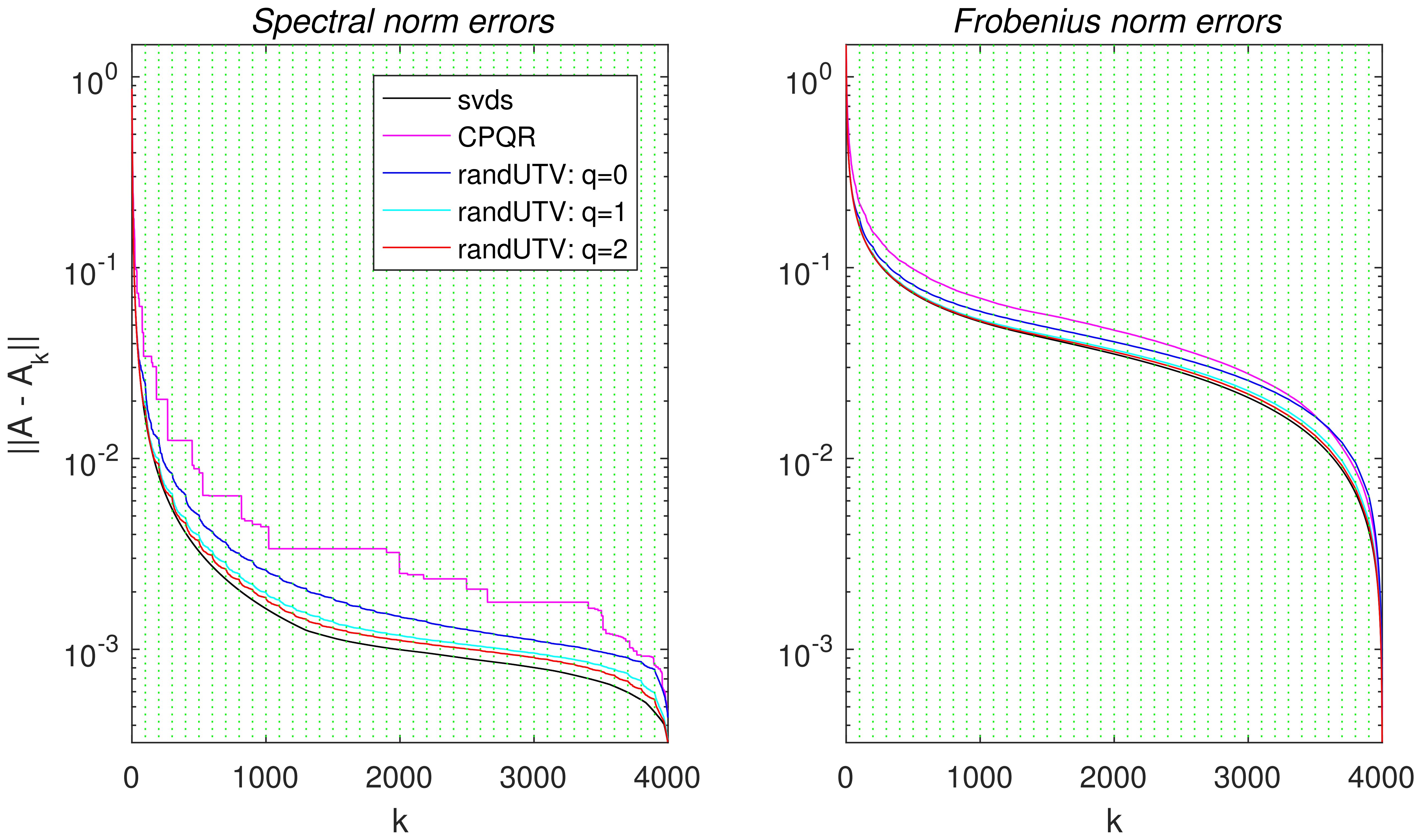
Computational speed of randUTV — 12 cores



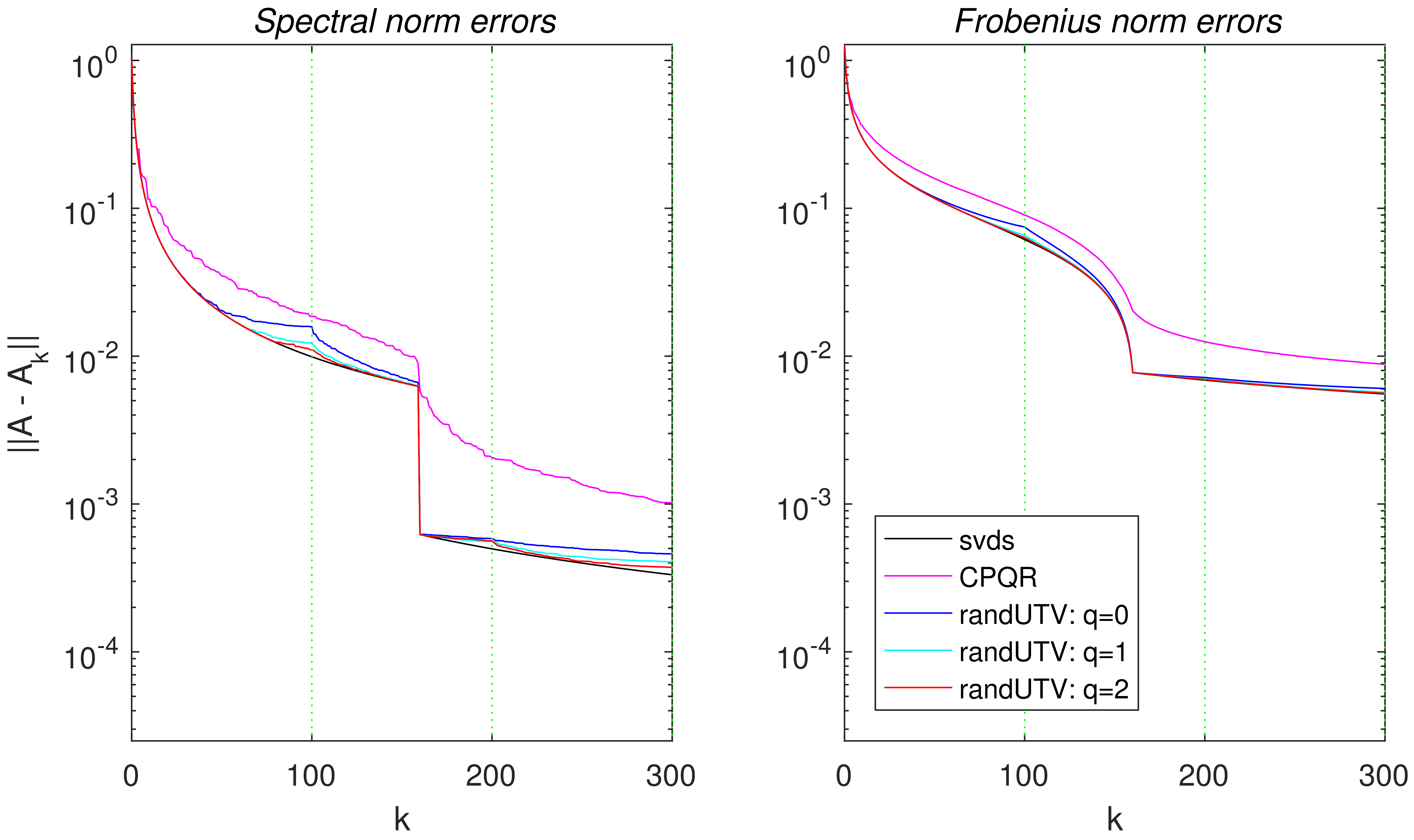
Computational speed of randUTV (red) compared to SVD (black) and column pivoted QR (blue). Each run is for a full factorization of an $n \times n$ matrix. We compare against Intel MKL LAPACK. Observe that the vertical axis is execution time scaled by n^3!



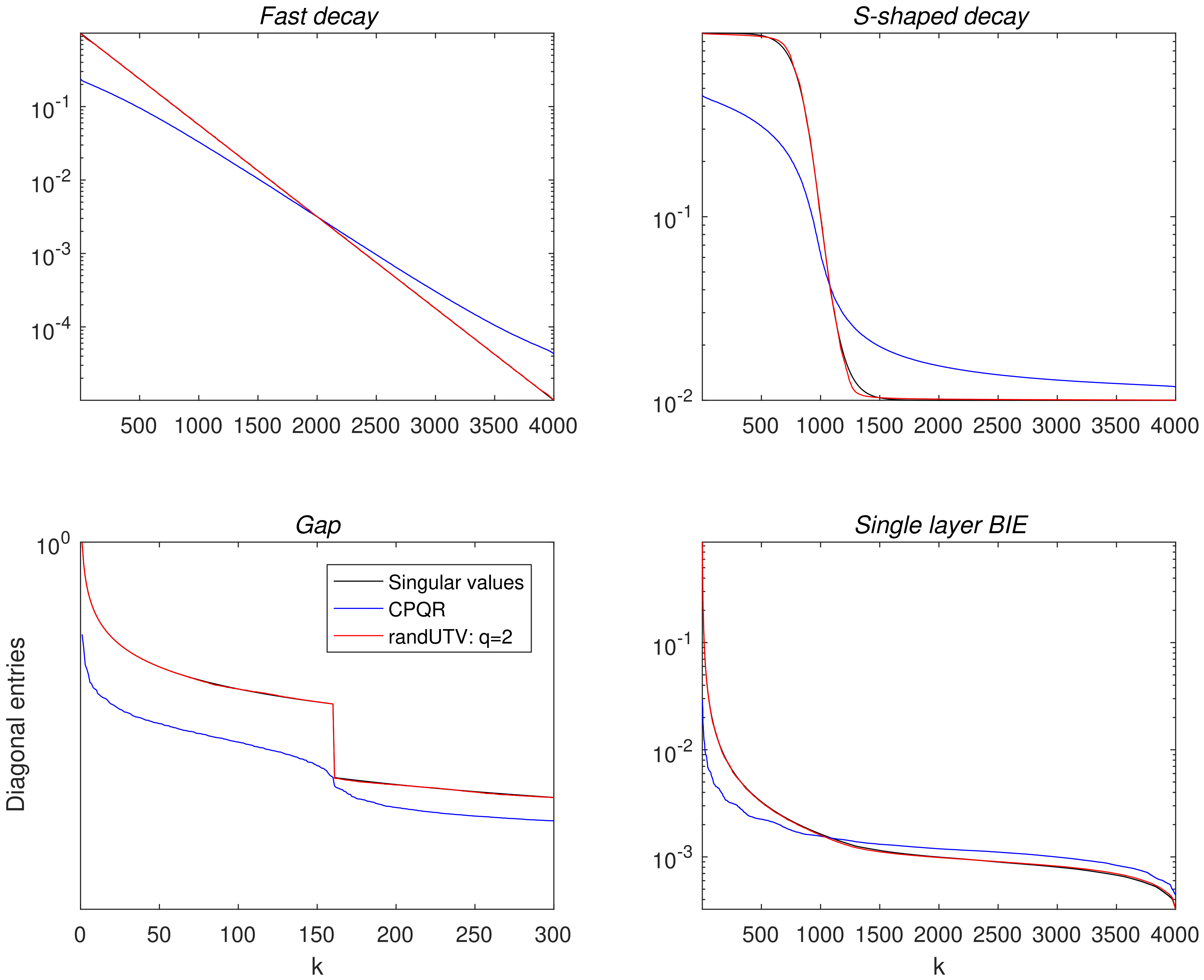
Rank-k approximation errors for the matrix "Fast Decay" of size 4000×4000*. The black lines mark the theoretically minimal errors. The block size was* $b = 100$ *and the green vertical lines mark block limits.*



Rank-k approximation errors for the matrix "BIE" of size 4000×4000*. 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 4000×4000. 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).*

Environment 2: Accelerate FULL factorizations of matrices

Key points:

- All operations are *blocked.*
- Interaction with **A** is only through matrix-matrix multiply.
- Very fast Householder QR with column pivoting. `https://github.com/flame/hqrrp/`
- Randomized UTV factorization:
  - Accuracy close to SVD.
  - Very fast: similar or faster than CPQR.
  - Admits partial factorizations, given a tolerance.
  - Very communication efficient. On GPU we see ×15 acceleration over SVD.

  To be slightly provocative: *Better than CPQR in basically every respect!*

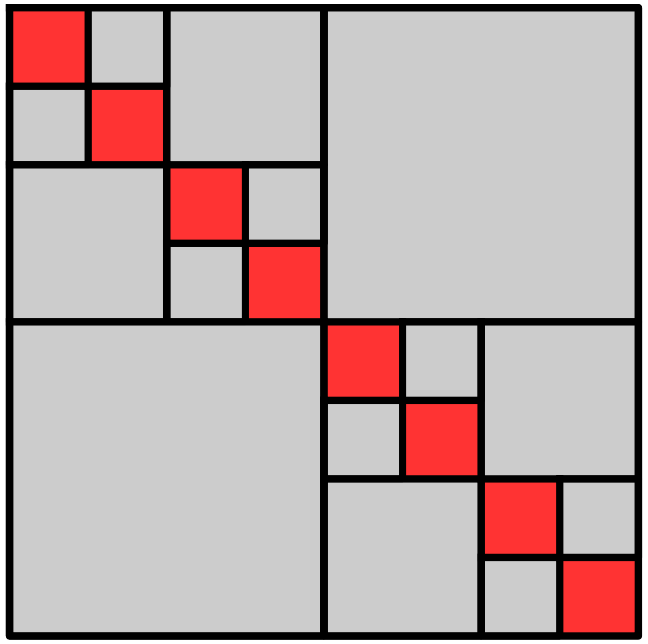
References:

- P.G. Martinsson, *Blocked rank-revealing QR factorizations: How randomized sampling can be used to avoid single-vector pivoting.* arXiv.org report #1505.08115, 2015.
- P.G. Martinsson, Gregorio Quintana-Ortí, Nathan Heavner, and R. van de Geijn, *Householder QR Factorization With Randomization for Column Pivoting (HQRRP).* To appear in SISC.
- P.G. Martinsson, Gregorio Quintana-Ortí, Nathan Heavner, *randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization.* Appearing on arXiv within weeks.
- Recent work by Ming Gu and Jed Duersch of UC-Berkeley.

Environment 3: Randomized approximation of rank-structured matrices. (Plug!)

Loosely speaking, a matrix is *rank-structured* if its off-diagonal blocks have low rank to some given precision. These matrices arise upon discretization of integral operators, in accelerating nested dissection, in simulating Monte Carlo processes, etc.

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.



Environment 3: Randomized approximation of rank-structured matrices

Loosely speaking, a matrix is *rank-structured* if its off-diagonal blocks have low rank to some given precision. These matrices arise upon discretization of integral operators, in accelerating nested dissection, in simulating Monte Carlo processes, etc.

Many “formats” have been proposed, including:

- “Fast Multipole Method” matrices.
- $\mathcal{H}$- and $\mathcal{H}^2$-matrices.
- Hierarchically Block Separable (HBS) matrices, a.k.a. “HSS” matrices.
- HODLR matrices (a.k.a. $\mathcal{S}$-matrices).

All these formats allow for (more or less) efficient matrix computations involving a range of operations such as matrix-vector multiply, matrix-matrix multiply, LU factorization, matrix inversion, forming of Schur complements, etc.

Objective: Suppose a matrix $\mathbf{A}$ is rank-structured, that you are *given* a tessellation pattern, and that you have an efficient technique for evaluating the matrix-vector product $\mathbf{x} \mapsto \mathbf{Ax}$. We then seek to build all factors in the rank-structured representation of $\mathbf{A}$.

Applications: Build “frontal matrices” in nested dissection. Matrix-matrix multiplication of two structured matrices. Convert from, say, FMM format, to HBS format. Et cetera.

Let **A** be a rank-structured matrix, for which we can rapidly evaluate $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ and $\mathbf{x} \mapsto \mathbf{A}^*\mathbf{x}$.

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

- P.G. Martinsson, *A fast randomized algorithm for computing a Hierarchically Semi-Separable representation of a matrix.* 2008 arxiv report. 2011 SIMAX paper.
- 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 $\mathbf{A}^*$.

The method is still very fast in many situations, and can save messy coding work. For instance, implementing the matrix-matrix mutliplication, or changing the partition tree, are quite hard to implement efficiently.

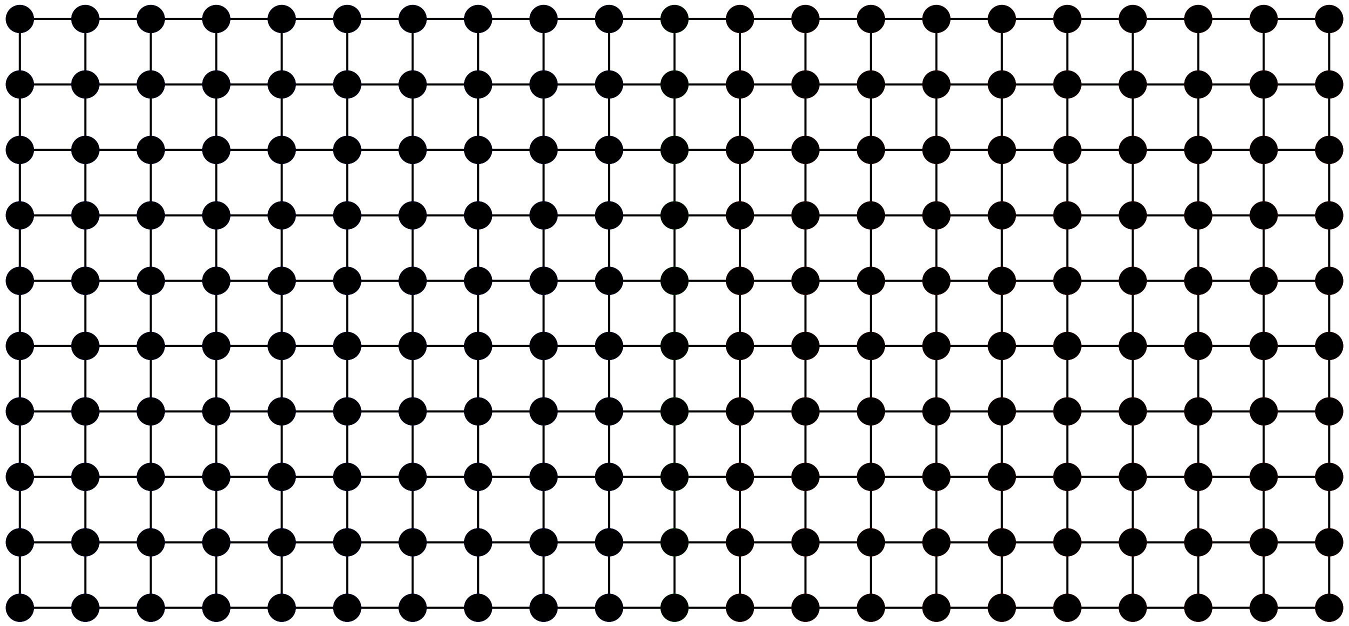
- L. Lin, J. Lu, L. Ying, *Fast construction of hierarchical matrix representation from matrix-vector multiplication*, JCP, **230**(10), 2011.
- P.G. Martinsson, *Compressing rank-structured matrices via randomized sampling.* SISC, **38**(4), 2016.

Question: What about Fast Direct Solvers?

Linear complexity nested dissection (ND)

ND is a well-known “divide-and-conquer” technique, due to George (1973).

To illustrate the idea, consider a rectangular grid with $N = (2n+1) \times n$ gridpoints:

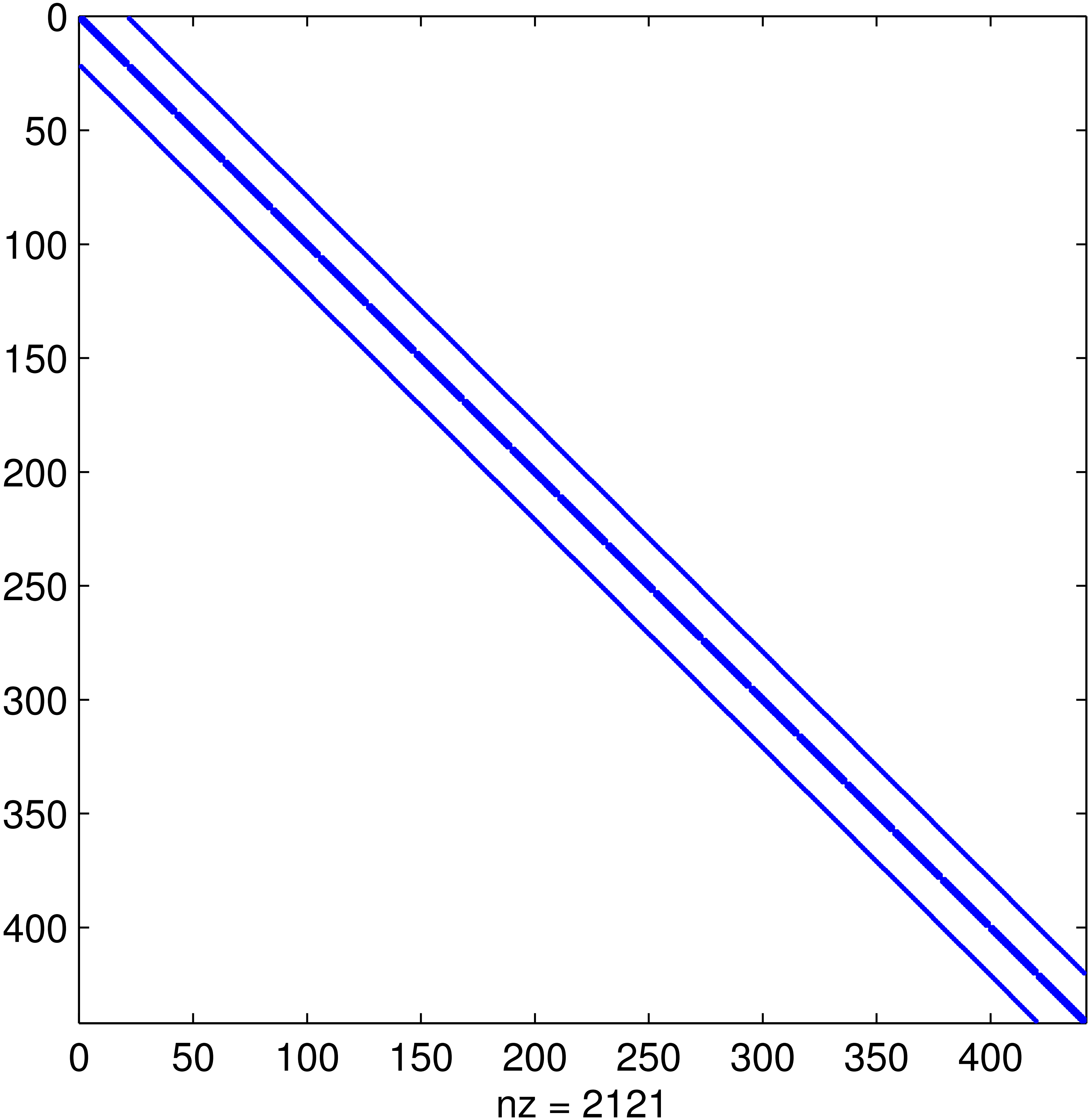


Let $\mathbf{A}$ denote the $N \times N$ matrix arising upon discretizing $-\Delta u = f$ using the classical 5-point stencil on this regular grid.

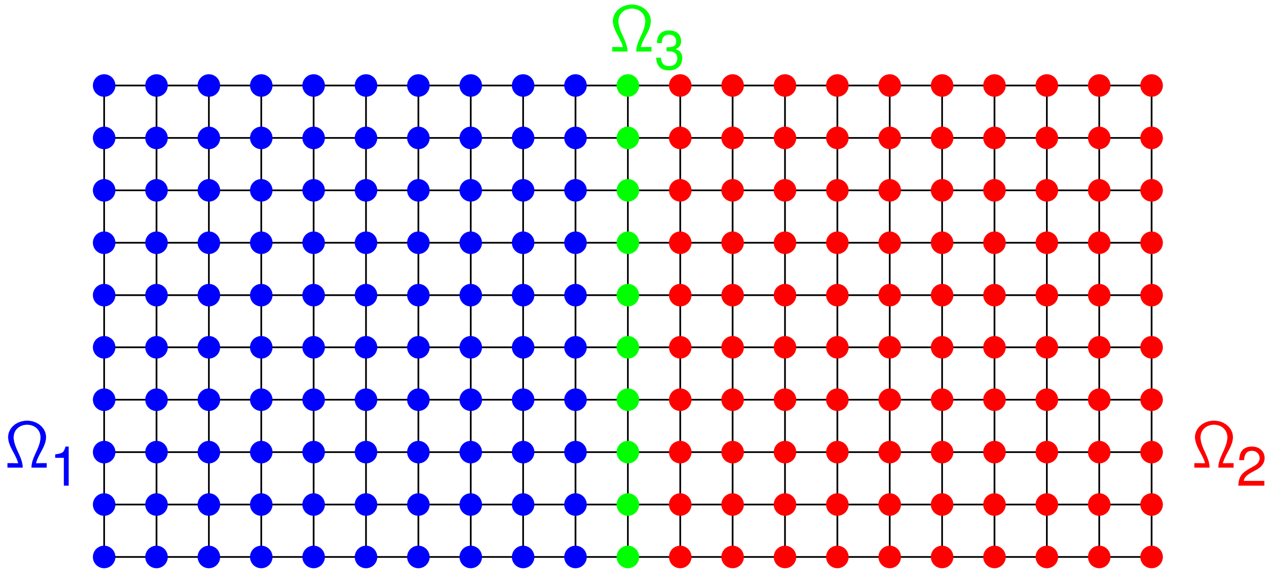
We seek to compute an LU-factorization of $\mathbf{A}$.

Important: $\mathbf{A}$ has a lot of sparsity.

Sparsity pattern of **A** with column wise grid ordering.



Let us divide the domain into three pieces:



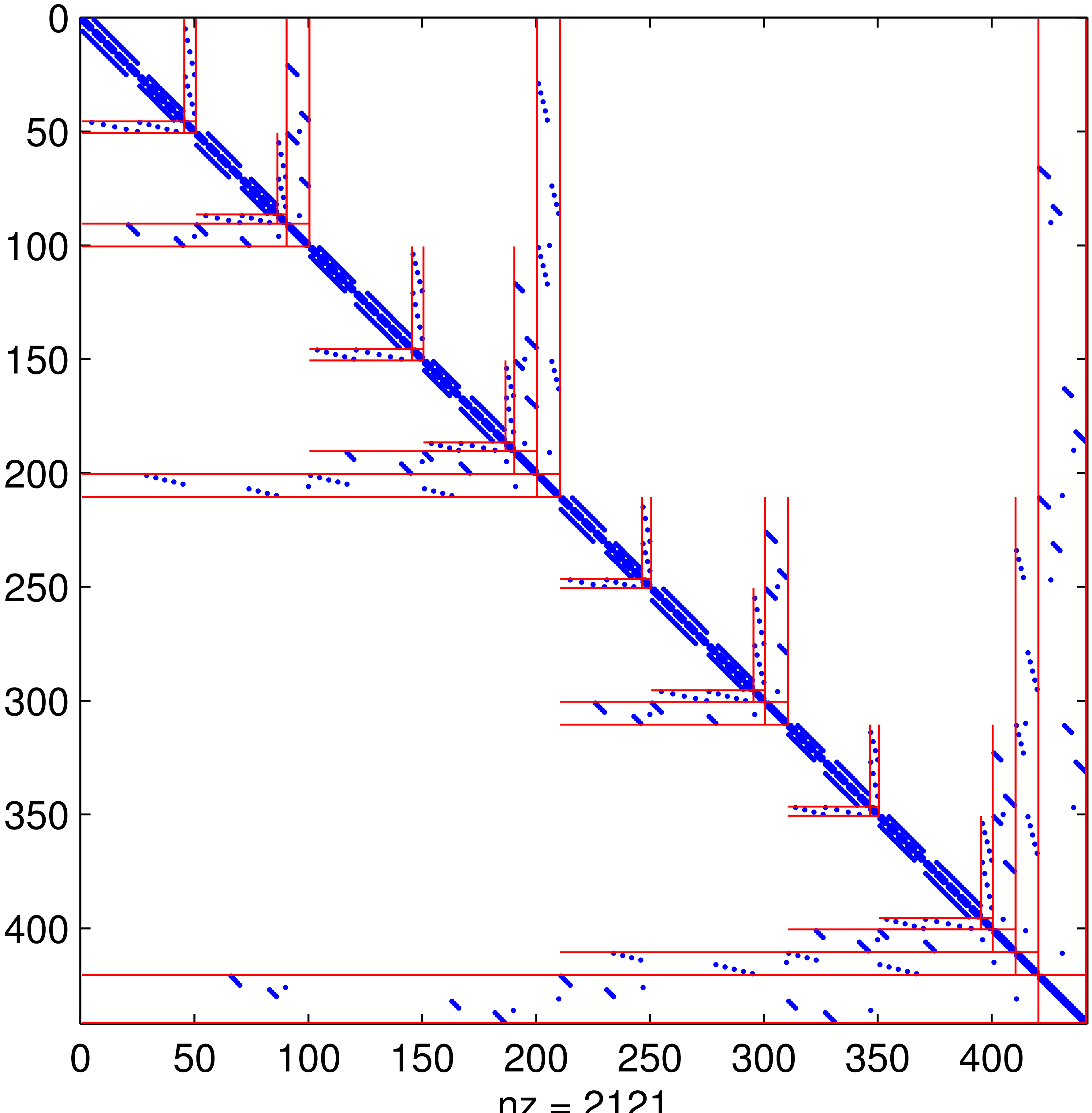
Note that there are no connections between nodes in Ω_1 and Ω_2.

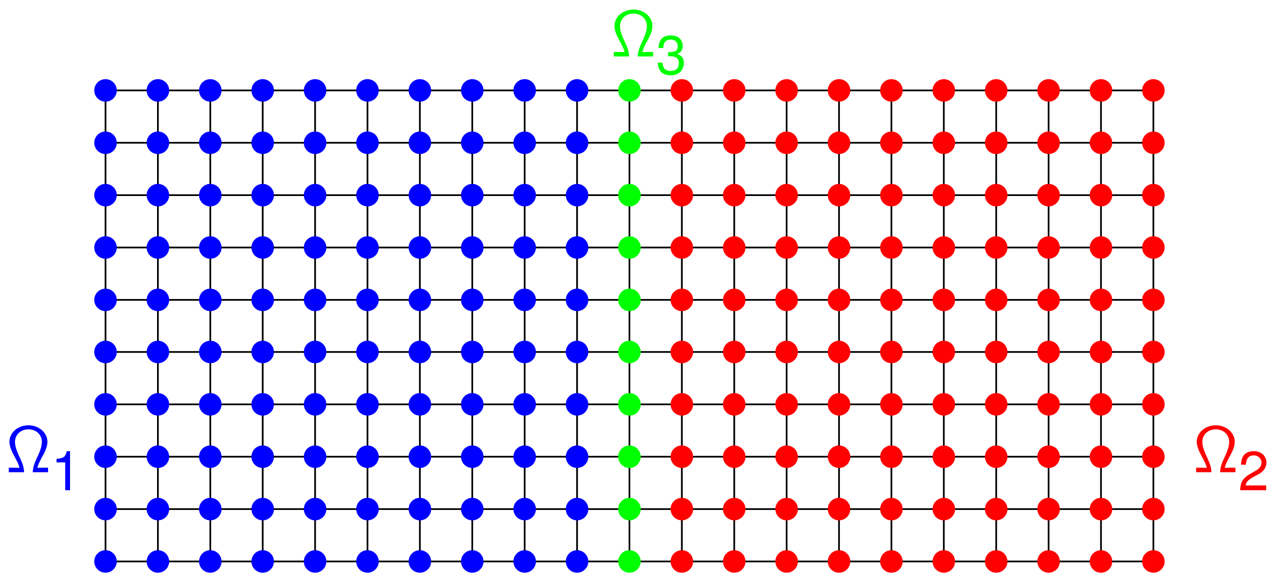
In consequence, the non-zero blocks of the coefficient matrix are:

$$\mathbf{A} = \left[\begin{array}{c|c|c} \mathbf{A}_{11} & & \mathbf{A}_{13} \\ \hline & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \hline \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{array}\right]$$

Now $\mathbf{A}_{13}$, $\mathbf{A}_{31}^{t}$, $\mathbf{A}_{23}$, and $\mathbf{A}_{32}^{t}$ have $n = O(N^{0.5})$ columns.

Sparsity pattern of **A** with nested dissection ordering.





Recall that the coefficient matrix is tessellated as

$$\mathbf{A} = \left[\begin{array}{c|c|c} \mathbf{A}_{11} & & \mathbf{A}_{13} \\ \hline & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \hline \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{array}\right].$$

Now suppose that we can somehow factor $\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{U}_{11}$ and $\mathbf{A}_{22} = \mathbf{L}_{22}\mathbf{U}_{22}$. Then

$$\mathbf{A} = \left[\begin{array}{c|c|c} \mathbf{L}_{11}\mathbf{U}_{11} & & \mathbf{A}_{13} \\ \hline & \mathbf{L}_{22}\mathbf{U}_{22} & \mathbf{A}_{23} \\ \hline \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{array}\right] = \left[\begin{array}{c|c|c} \mathbf{L}_{11} & & \\ \hline & \mathbf{L}_{22} & \\ \hline \mathbf{A}_{31}\mathbf{U}_{11}^{-1} & \mathbf{A}_{32}\mathbf{U}_{22}^{-1} & \mathbf{I} \end{array}\right] \left[\begin{array}{c|c|c} \mathbf{I} & & \\ \hline & \mathbf{I} & \\ \hline & & \mathbf{S}_{33} \end{array}\right] \left[\begin{array}{c|c|c} \mathbf{U}_{11} & & \mathbf{L}_{11}^{-1}\mathbf{A}_{13} \\ \hline & \mathbf{U}_{22} & \mathbf{L}_{22}^{-1}\mathbf{A}_{23} \\ \hline & & \mathbf{I} \end{array}\right].$$

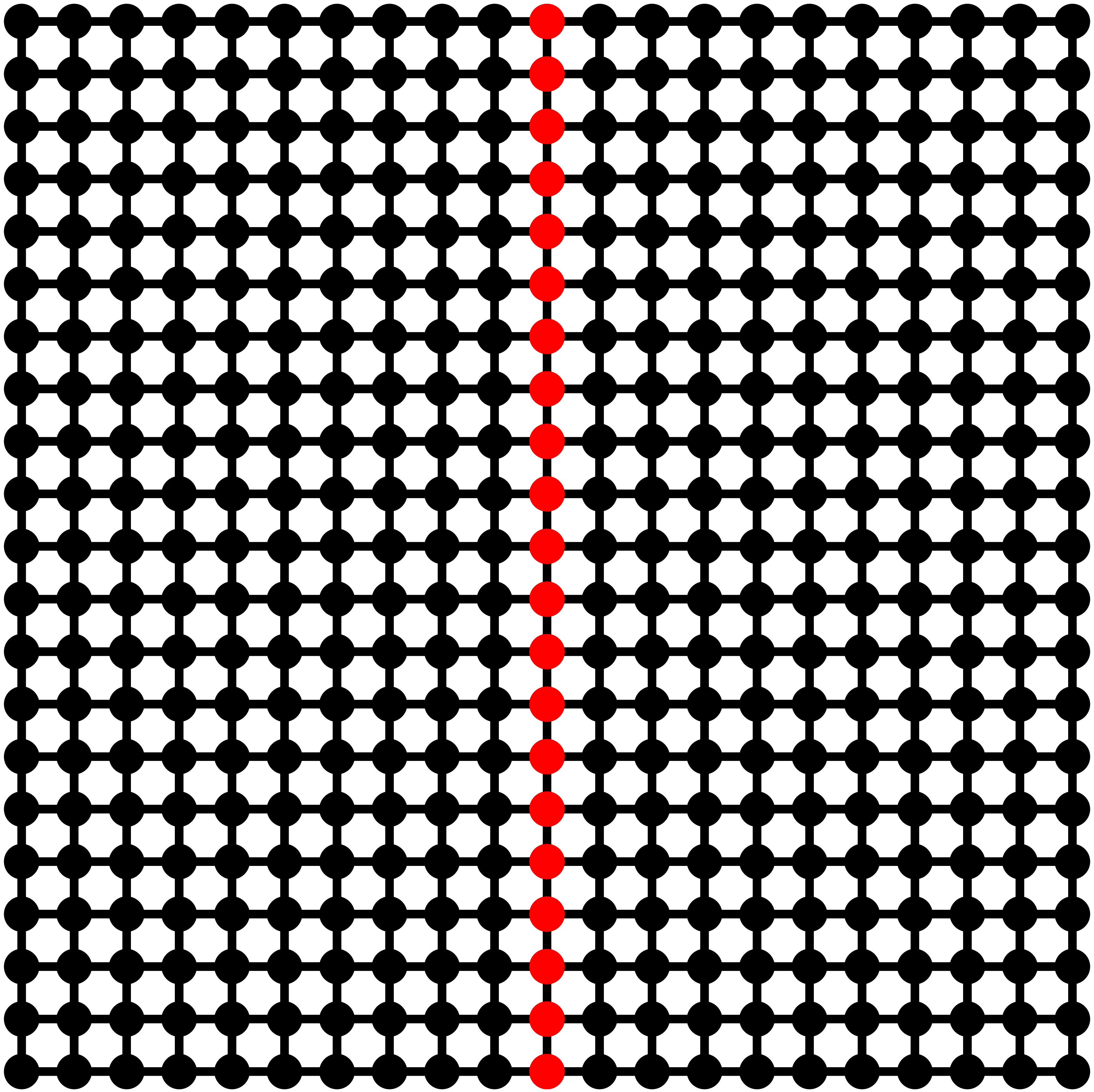
So in order to compute an LU factorization of $\mathbf{A}$, we need to:

- Compute the factorization $\mathbf{A}_{11} = \mathbf{L}_{11}\,\mathbf{U}_{11}$. *size* $\sim N/2 \times N/2$
- Compute the factorization $\mathbf{A}_{22} = \mathbf{L}_{22}\,\mathbf{U}_{22}$. *size* $\sim N/2 \times N/2$
- Compute the Schur complement $\mathbf{S}_{33} = \mathbf{A}_{33} - \mathbf{A}_{31}\mathbf{U}_{11}^{-1}\mathbf{L}_{11}^{-1}\mathbf{A}_{13} - \mathbf{A}_{32}\mathbf{U}_{22}^{-1}\mathbf{L}_{22}^{-1}\mathbf{A}_{23}$.

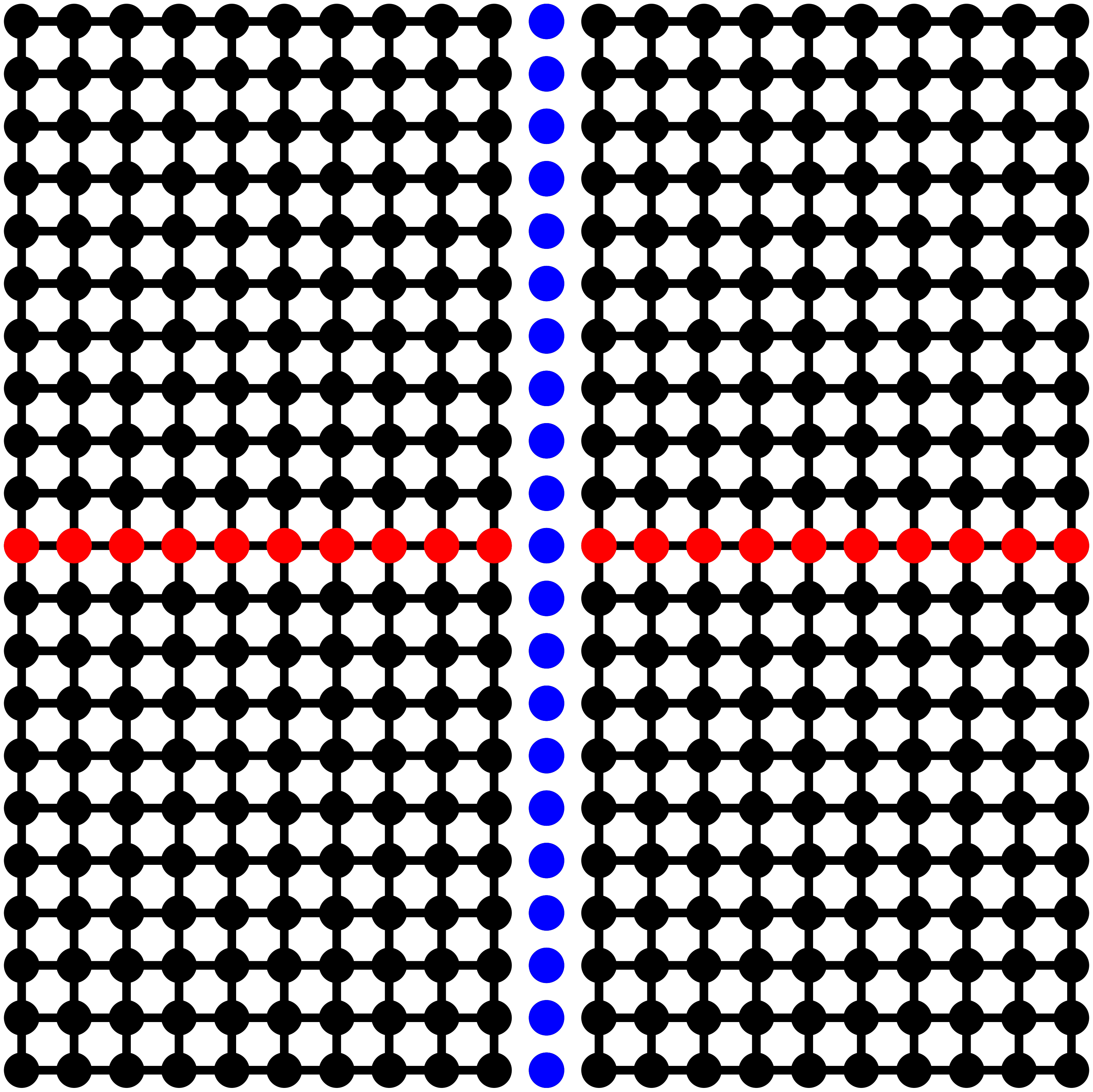
  Then compute the factorization $\mathbf{S}_{33} = \mathbf{L}_{33}\,\mathbf{U}_{33}$. *size* $\sim \sqrt{N} \times \sqrt{N}$

Notice the obvious recursion!

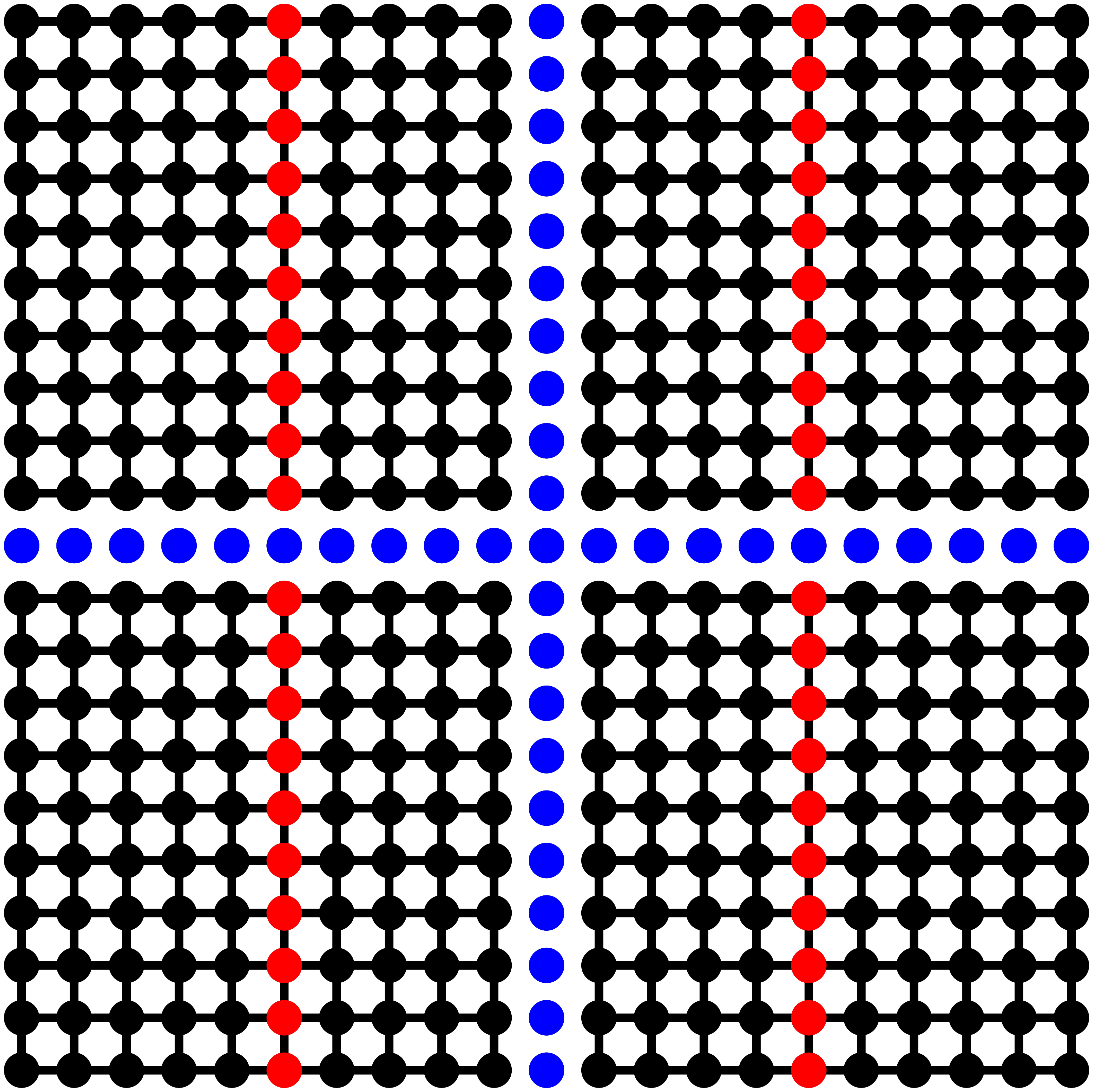
Nested dissection – level 0



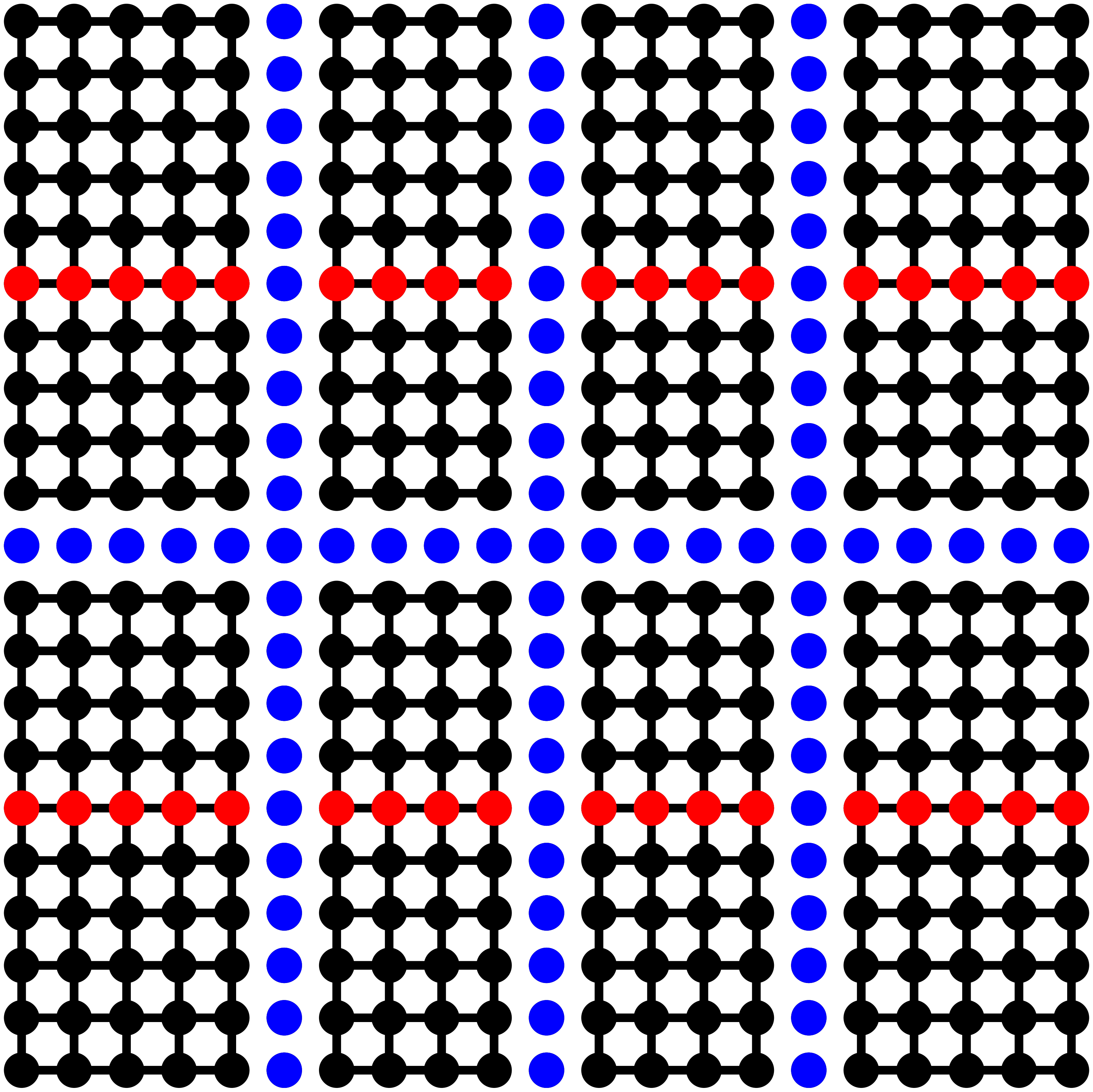
Nested dissection – level 1



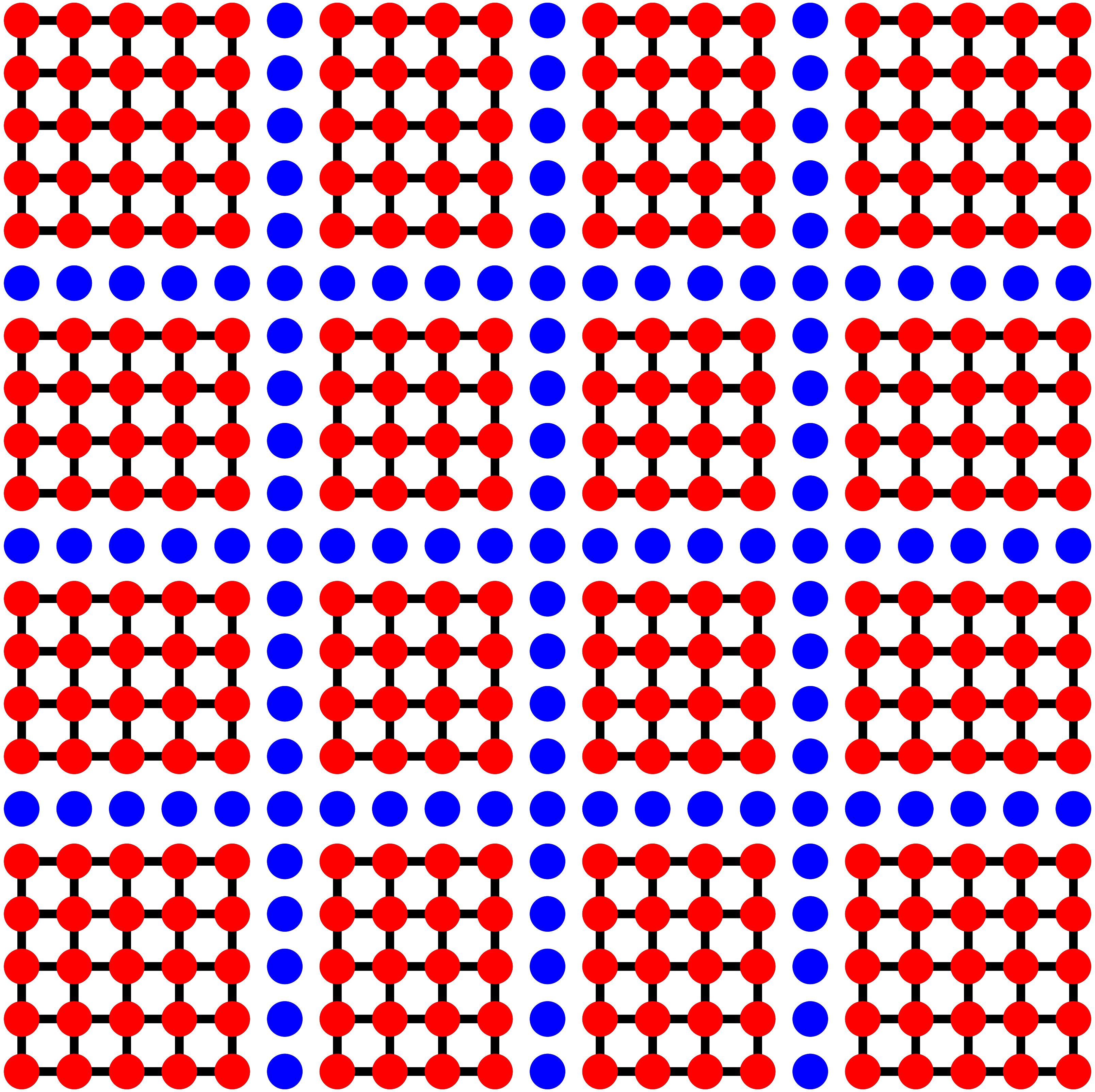
Nested dissection – level 2



Nested dissection – level 3



Nested dissection – level 4



Problem: The Schur complements that need to be factored are *dense.*

The top level Schur complement is of size $O(N^{0.5}) \times O(N^{0.5})$ → complexity $O(N^{1.5})$.

In 3D, this Schur complement is of size $O(N^{2/3}) \times O(N^{2/3})$ → complexity $O(N^{2})$.

	Build stage	*Solve stage*	*Memory*
2D	$O(N^{3/2})$	$O(N \log N)$	$O(N \log N)$
3D	$O(N^{2})$	$O(N^{4/3})$	$O(N^{4/3})$

Problem: The Schur complements that need to be factored are *dense.*

The top level Schur complement is of size $O(N^{0.5}) \times O(N^{0.5})$ → complexity $O(N^{1.5})$.

In 3D, this Schur complement is of size $O(N^{2/3}) \times O(N^{2/3})$ → complexity $O(N^{2})$.

	Build stage	*Solve stage*	*Memory*
2D	$O(N^{3/2})$	$O(N \log N)$	$O(N \log N)$
3D	$O(N^{2})$	$O(N^{4/3})$	$O(N^{4/3})$

Fix: The dense matrices are *rank-structured.* Conceptually, they behave like discretized integral operators on the interface (e.g. a discretized Dirichlet-to-Neumann operator). Such matrices admit very efficient LU factorization, inversion, etc.

By incorporating structured matrix algebra in nested dissection, solvers with overall $O(N)$ complexity result.

When run at low accuracy, excellent *pre-conditioners* often result.

To obtain a *direct solver,* fairly high local accuracies must be used:

- Local ranks grow.
- Still $O(N)$, but high cost per degree of freedom, in particular in regards to storage.

References:

Classical nested dissection:

- Original work by A. George (1973), A.J. Hoffman, M.S. Martin, and D.J. Rose (1973).
- Book "Direct Methods for Sparse Matrices" by I. Duff (1987).
- Book "Direct Methods for Sparse Linear Systems" by T. Davis (2006).
- Huge literature ... well-developed software packages ...

Acceleration to $O(N)$ complexity:

- Le Borne, Grasedyck, & Kriemann (2007).
- Xia, Chandrasekaran, Gu, & Li (2009). → Purdue group.
- Martinsson (2009), Gillman & Martinsson (2011).
- Schmitz & Ying (2012).
- Darve & Ambikasaran (2013).
- Ho & Ying (2015).
- Solovyev (2015).
- Ghysels, Li, Rouet, Williams, Napov (2015).
- Chavez, Turkiyyah, Keyes (2016).
- Sushnikova, Oseledets (2016).
- Currently very active field!

Claim: Combining *fast direct solvers* and *high-order discretization* is compelling:

- FDS have high costs per DOF, so using fewer DOFs is critical.
- High-order methods are required for frequency domain wave propagation, which is a primary target for FDS.
- High-order methods can be challenging for iterative solvers.

Claim: Combining *fast direct solvers* and *high-order discretization* is compelling:

- FDS have high costs per DOF, so using fewer DOFs is critical.
- High-order methods are required for frequency domain wave propagation, which is a primary target for FDS.
- High-order methods can be challenging for iterative solvers.

Problem: Combining *nested dissection* and *high-order discretization* is problematic:

Matrix	N	T_{build}	T_{solve}	R	E_{pot}	E_{pot}	E_{pot}
		(seconds)	(seconds)	(MB)	Helm-I	Helm-II	Helm-III
5-point stencil	40000	2.46e-1	5.32e-3	38.26	2.7e0	1.2e0	3.1e0
$O(h^2)$	160000	1.29	2.74e-2	211.43	2.0e1	2.5e1	1.9e1
	640000	6.87	1.33e-1	1073.39	3.1e-1	6.7e1	1.4e1
	2560000	49.86	6.98e-1	5959.00	6.1e-2	8.8e1	3.7e1
	10240000	277.31	3.134	27588.61	1.5e-2	1.6e1	3.5e1
9-point stencil	40000	6.78e-1	1.42e-2	123.82	≥5.5e-2	≥3.8e0	≥1.3e0
$O(h^4)$	160000	4.18	7.59e-2	726.62	≥8.0e-3	≥1.8e1	≥3.2e-1
	640000	35.83	3.80e-1	3509.28	≥1.4e-4	≥1.4e1	≥9.6e-1
	2560000	383.04	2.12	18817.29	≥1.0e-5	≥6.1e-1	≥2.4e0
13-point stencil	40000	1.51	2.81e-2	285.08	≥4.7e-4	≥8.7e1	≥8.6e-3
$O(h^6)$	160000	9.76	1.59e-1	1575.29	≥1.7e-5	≥1.1e1	≥1.3e-1
	640000	164.33	9.03e-1	86661.39	≥5.9e-7	≥4.9e-1	≥2.9e-1
	2560000	1581.11	5.19	42191.17	≥4.1e-9	≥8.3e-2	≥1.4e-1

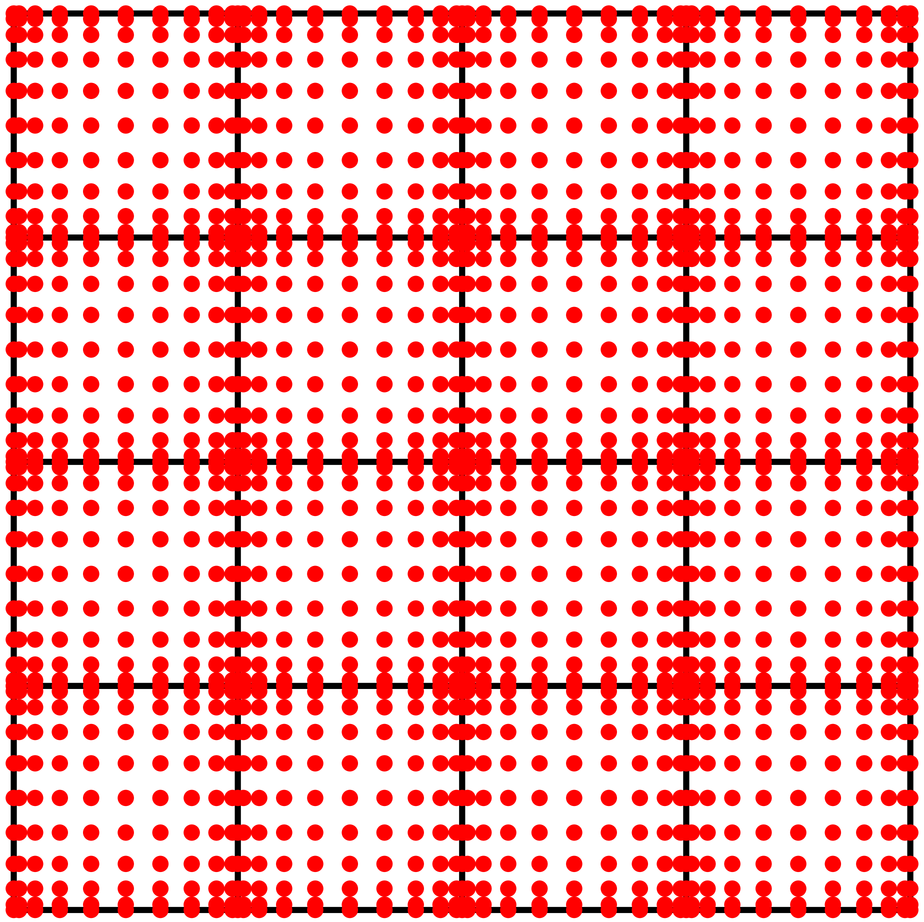
Question: How do you combine high-order schemes with fast direct solvers?

Potential solution: Use a discretization scheme specifically designed for FDS. It is based on a multidomain spectral collocation discretization. *(Joint work with A. Gillman.)*

For simplicity, let us consider a "variable wave speed" Helmholtz problem in 2D: Given f, g, and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.



The unknown function u is represented as a vector holding approximations to its point-wise values at the grid points (collocation). Across domain boundaries, we enforce continuity of potentials and normal derivatives.

Prior work: *The discretization scheme is similar to existing composite (or "multi-domain") spectral collocation methods by Hesthaven and others. In particular: Pfeiffer, Kidder, Scheel, Teukolsky, (2003).*

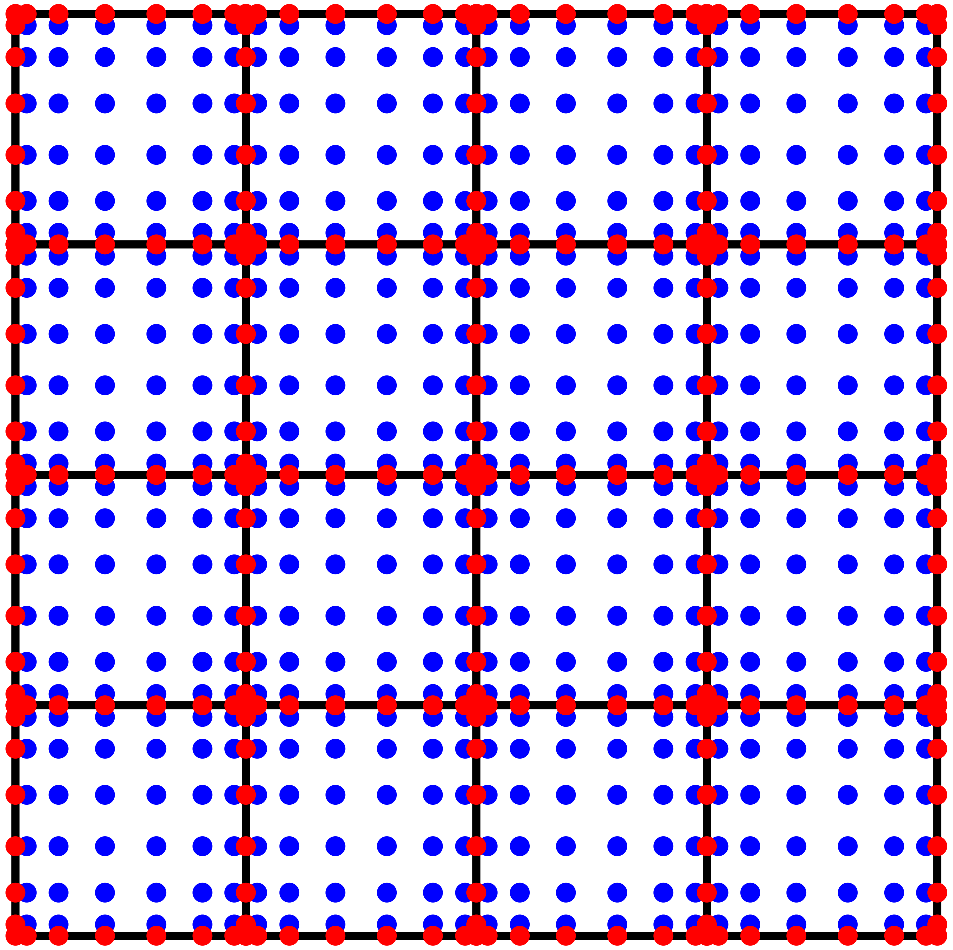
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\, u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Process leaves: Eliminate the interior (blue) nodes. ("Static condensation.")

Technically, we compute the Dirichlet-to-Neumann operator via a local spectral computation.

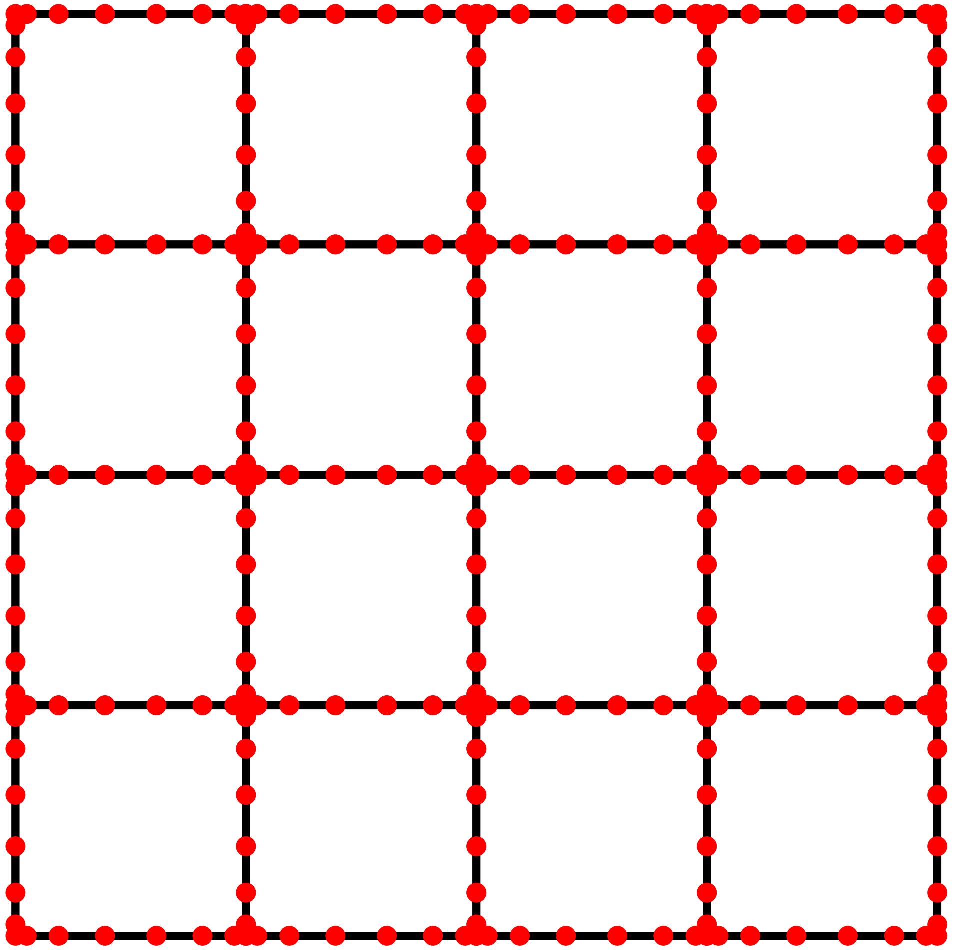


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Process leaves: Eliminate the interior (blue) nodes. (“Static condensation.”) Technically, we compute the Dirichlet-to-Neumann operator via a local spectral computation.

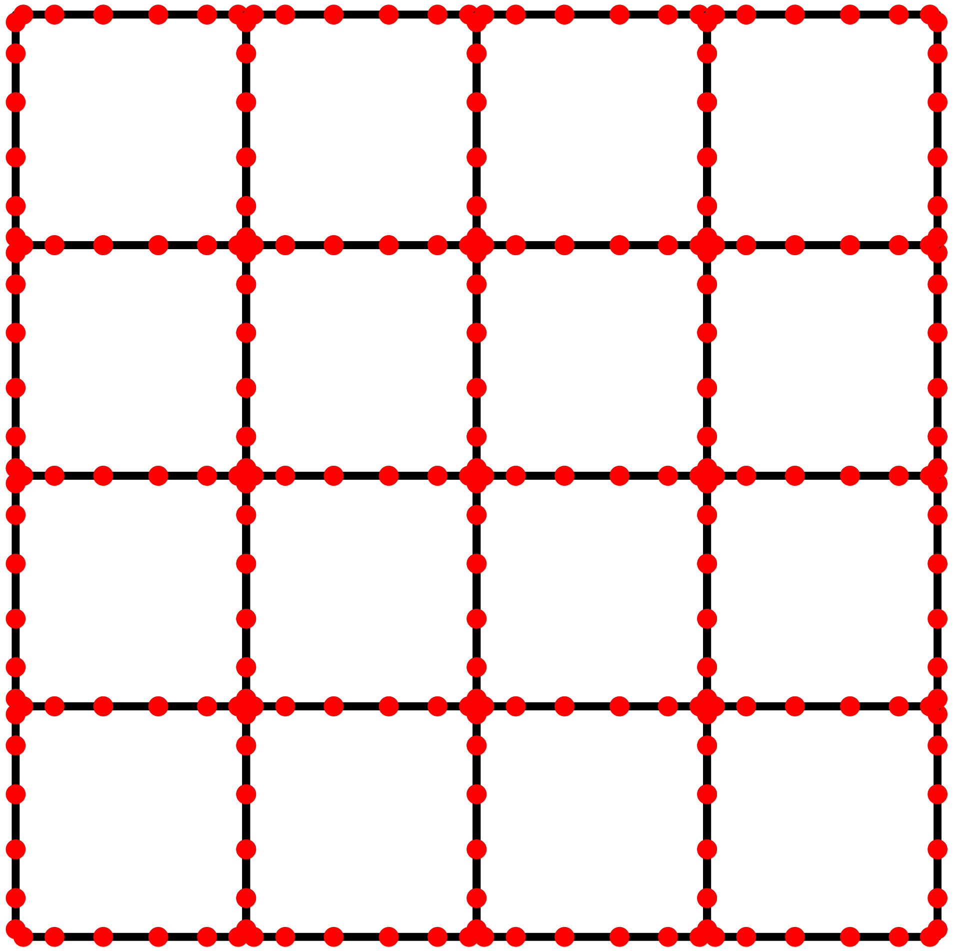


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Process leaves: Retabulate from Chebyshev to *Legendre nodes* on boundaries.



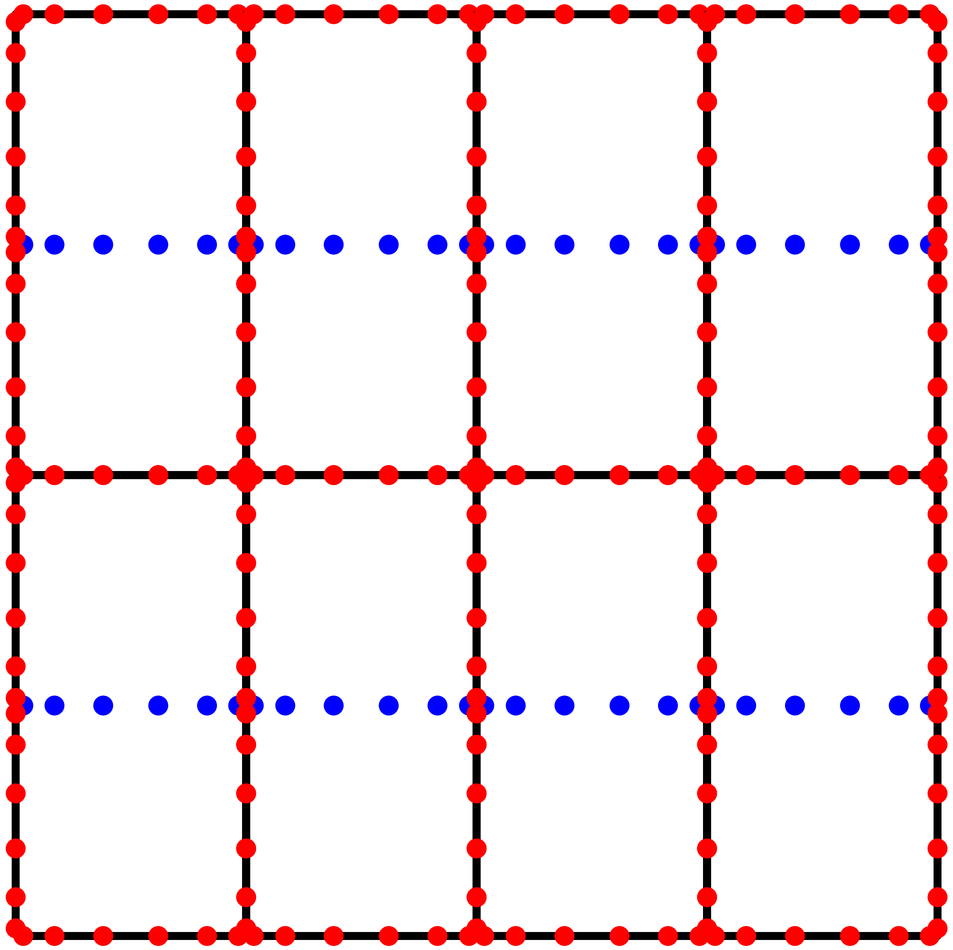
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0, 1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



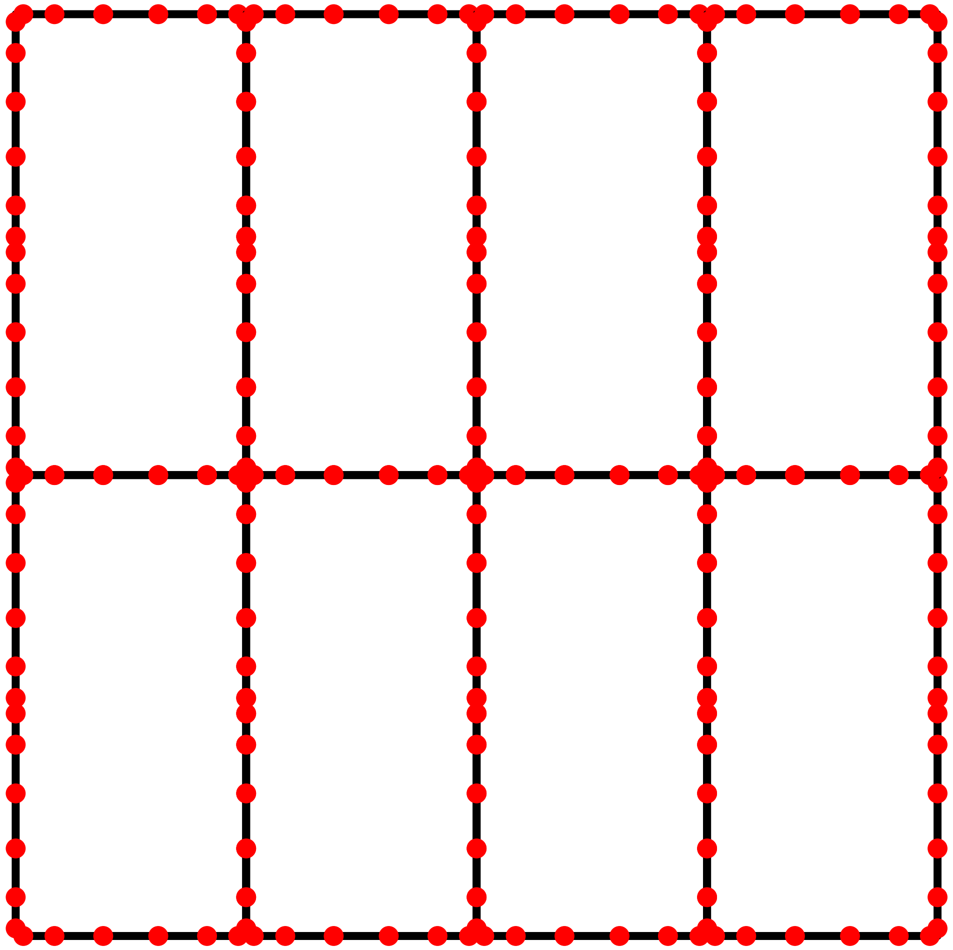
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\, u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0, 1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



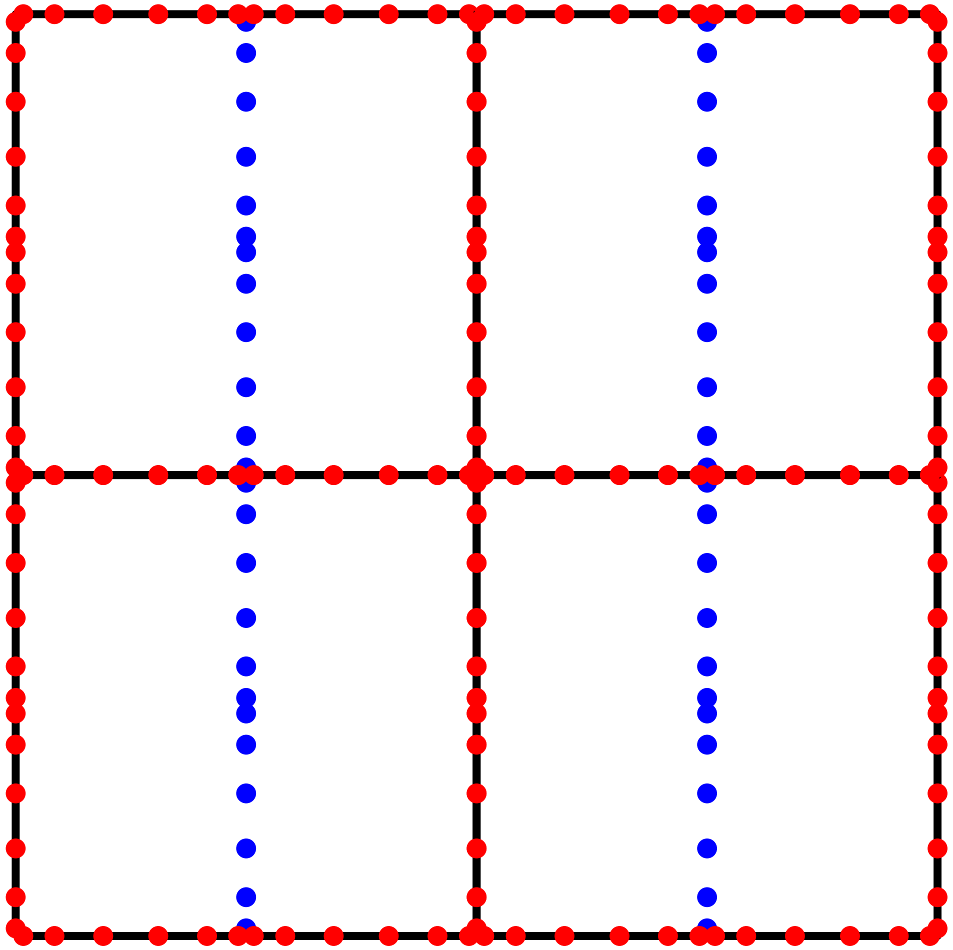
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\, u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0, 1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



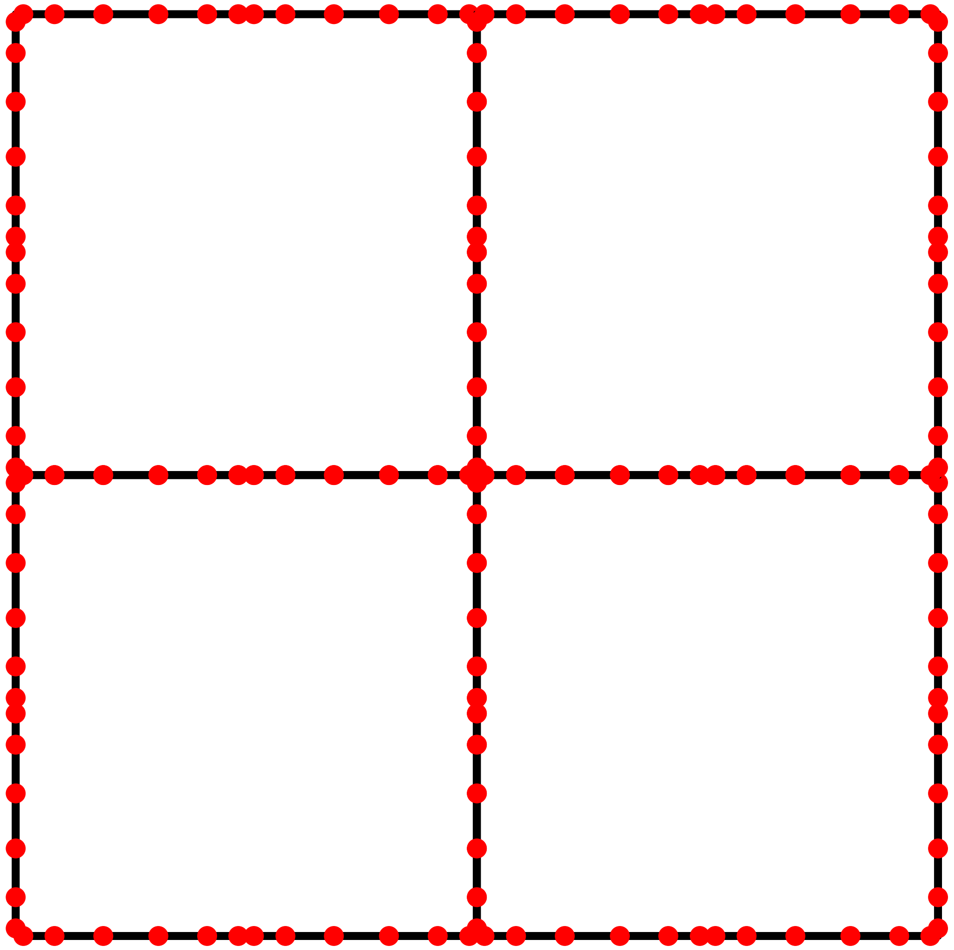
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



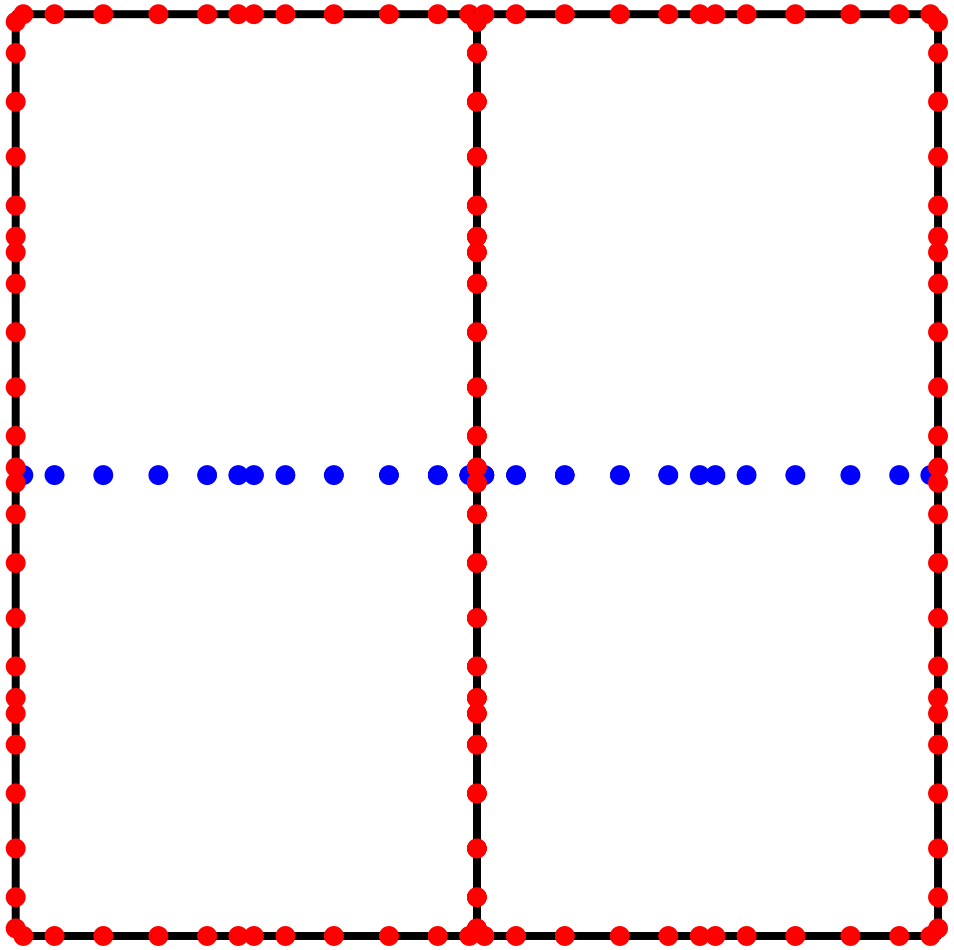
Model problem: Given *f* and *b*, find *u* such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume *u* is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of *u* and du/dn across interior boundaries. Compute the DtN operator for the larger box.

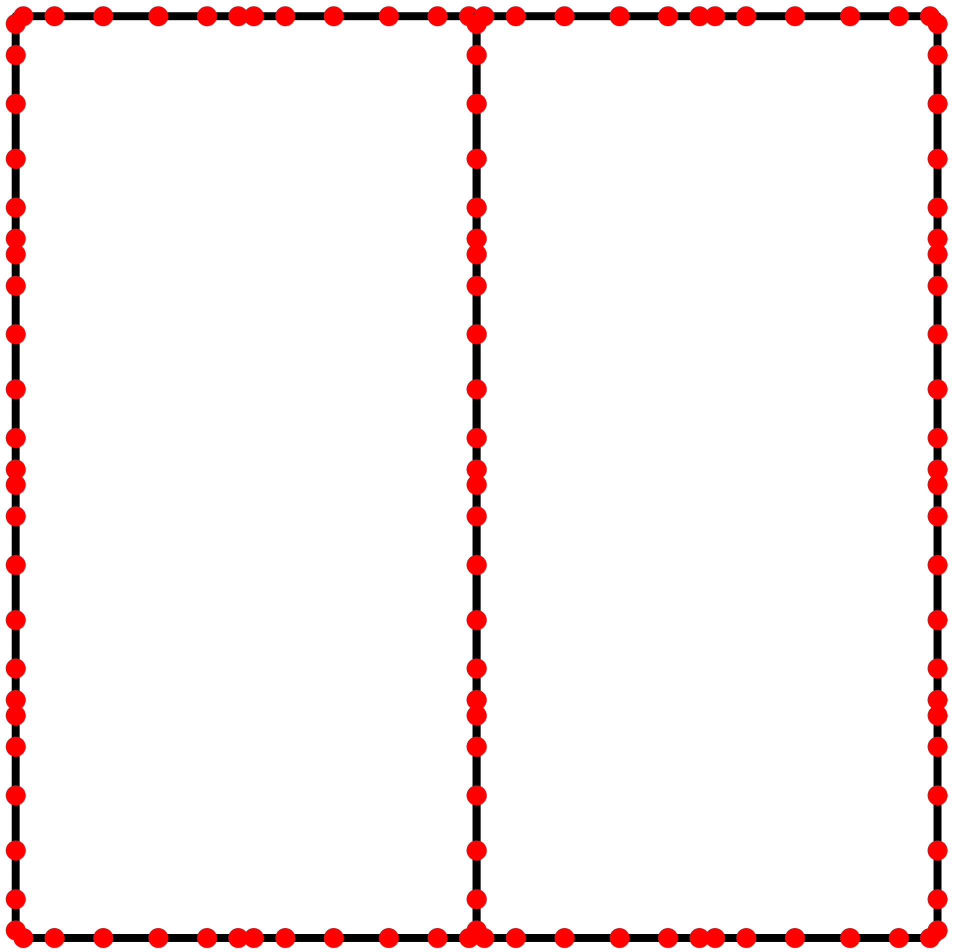


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.
To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



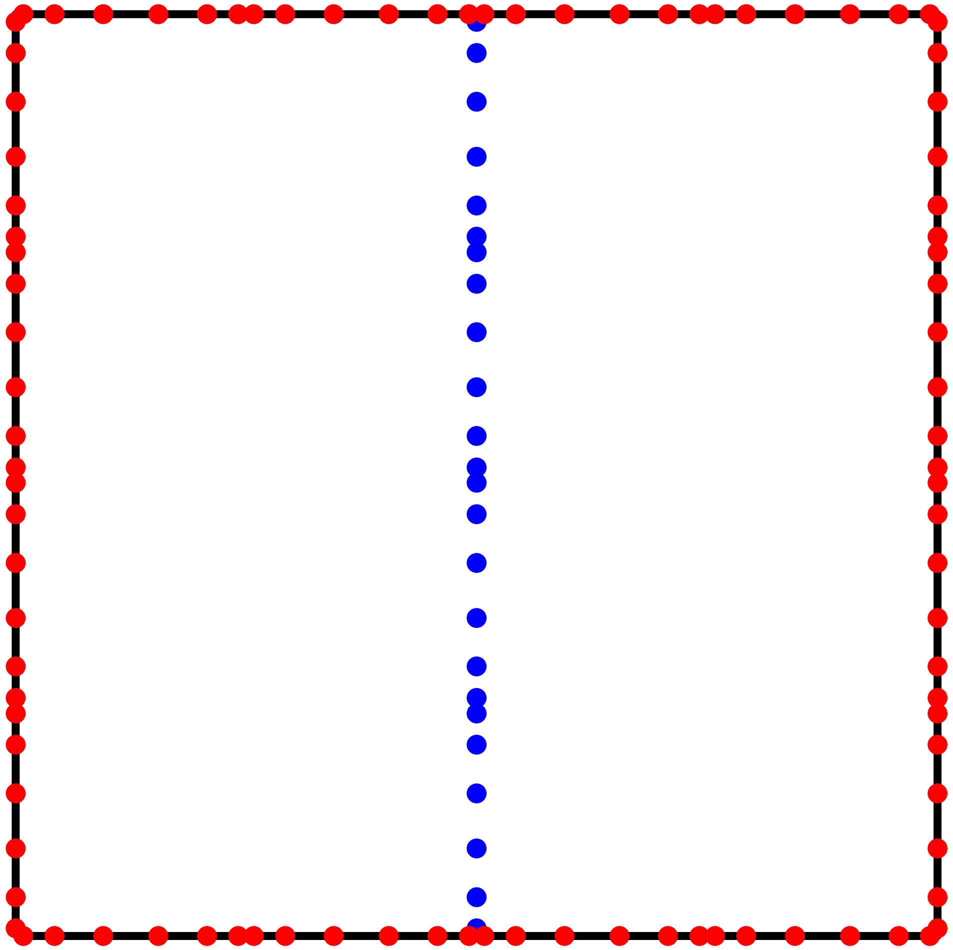
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\, u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0, 1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.



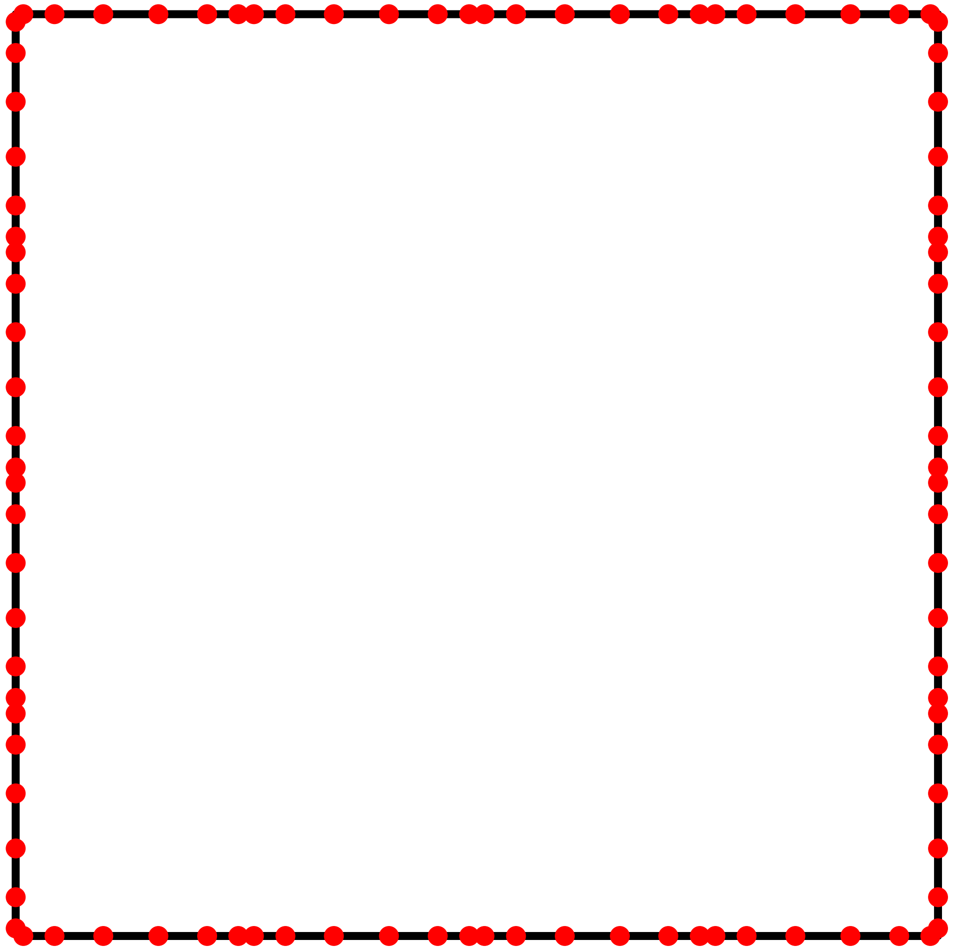
Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Upwards sweep: Merge boxes by pairs and eliminate the interior (blue) nodes.

To do this, use the computed DtN operators to enforce continuity of u and du/dn across interior boundaries. Compute the DtN operator for the larger box.

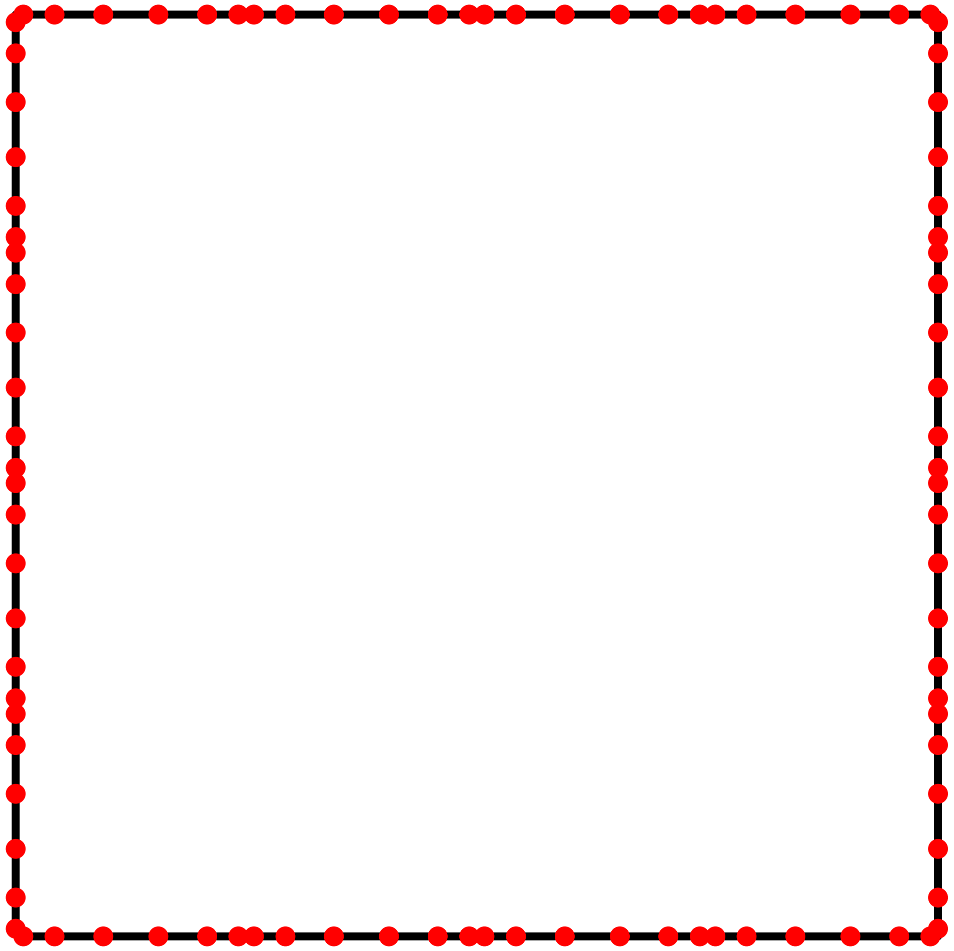


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\, u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Top level solve: Invert the DtN operator for the top level box.

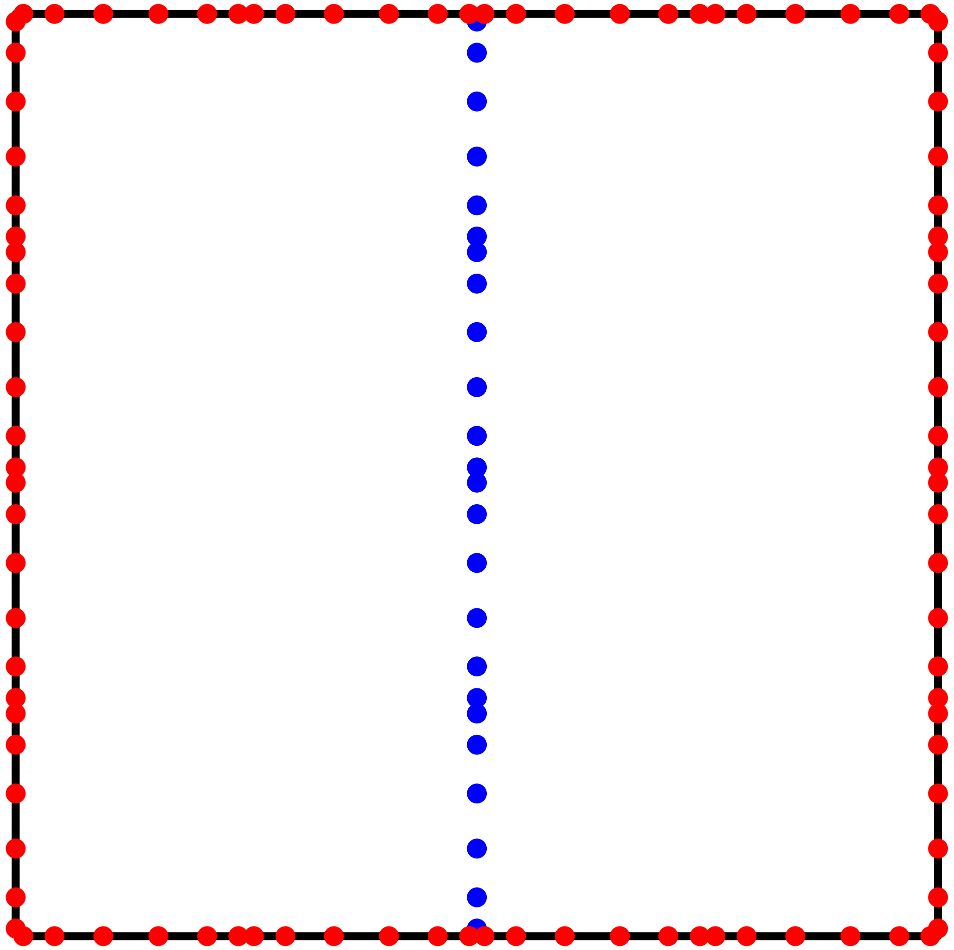


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Downwards sweep: We know u on the red nodes. We can use the computed DtN operators to reconstruct u on the blue nodes.

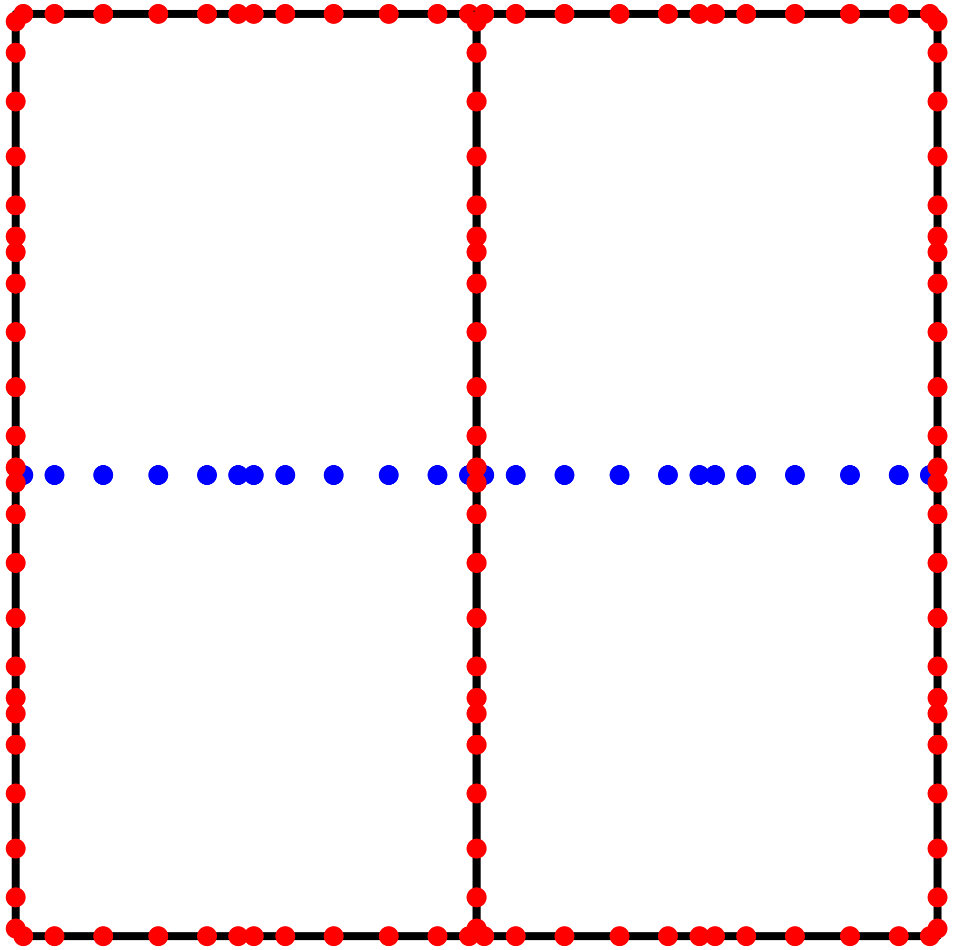


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Downwards sweep: We know u on the red nodes. We can use the computed DtN operators to reconstruct u on the blue nodes.

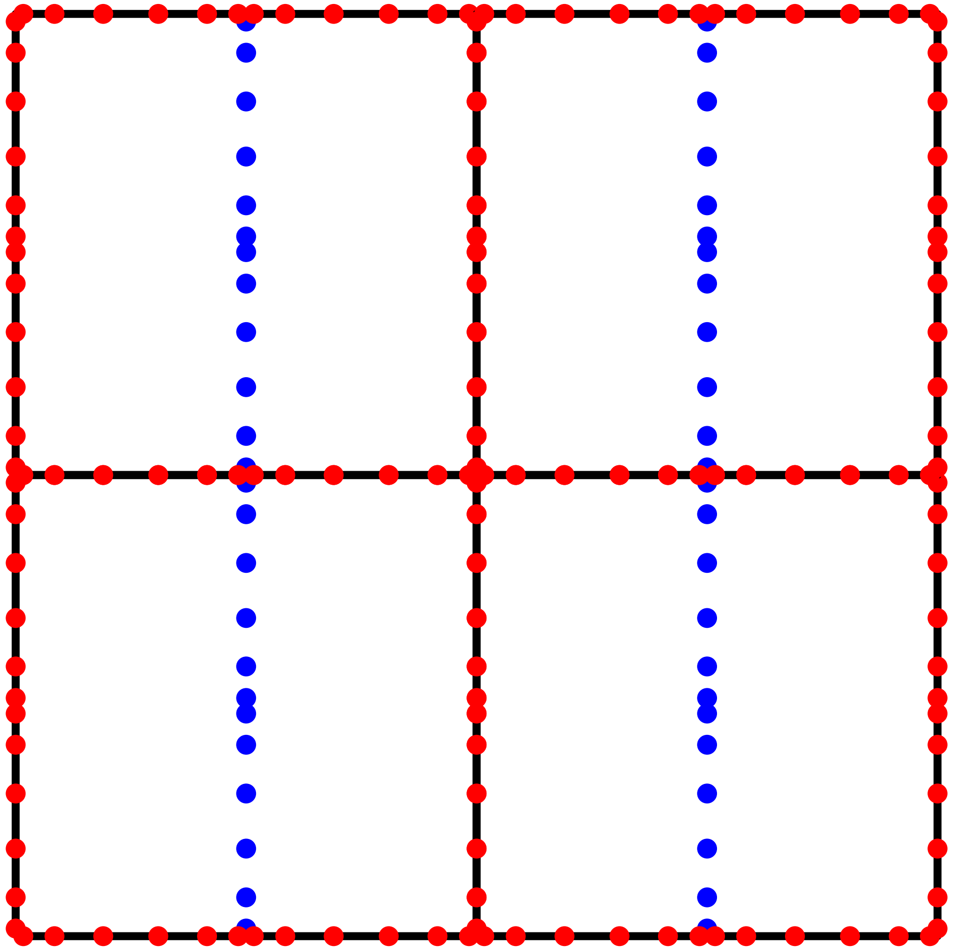


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Downwards sweep: We know u on the red nodes. We can use the computed DtN operators to reconstruct u on the blue nodes.

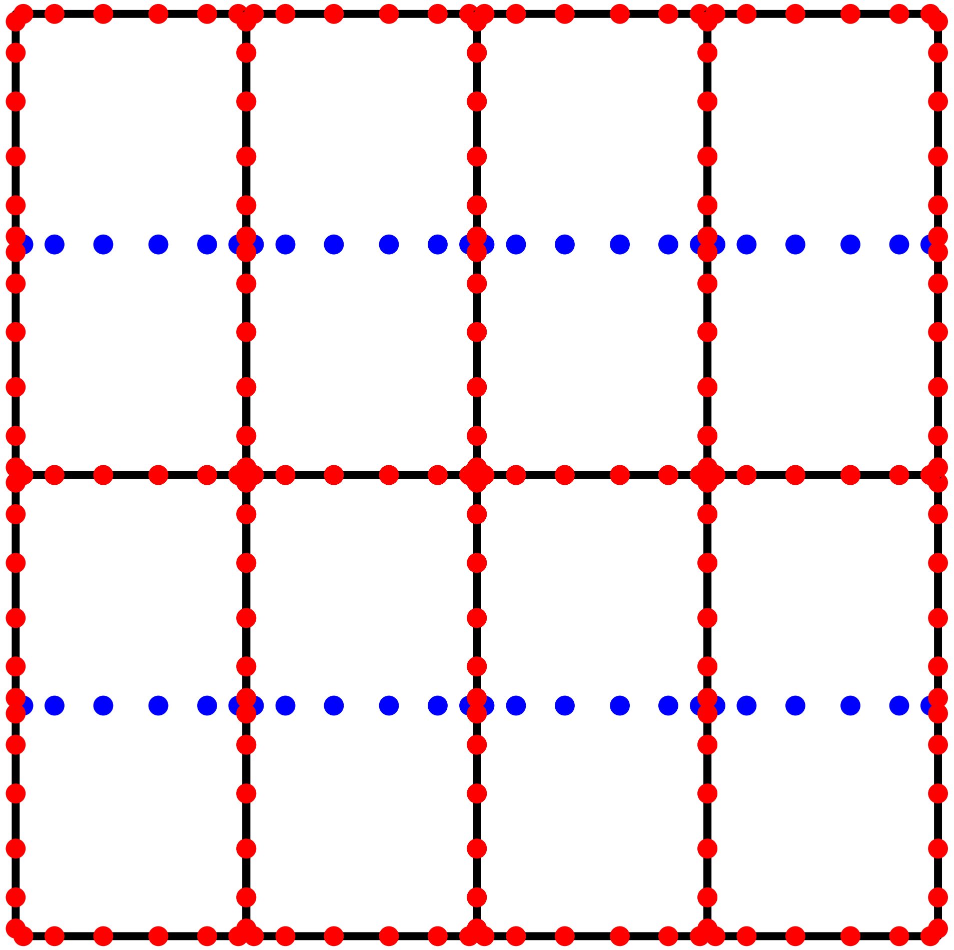


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Downwards sweep: We know u on the red nodes. We can use the computed DtN operators to reconstruct u on the blue nodes.

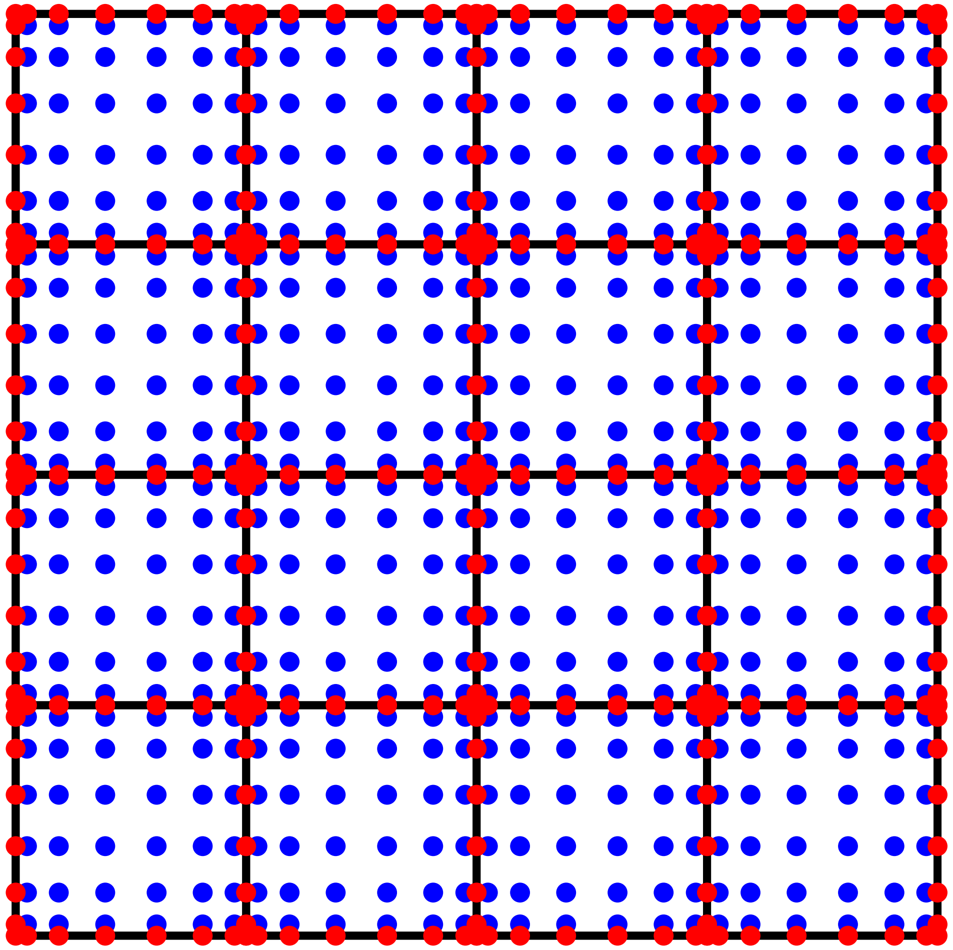


Model problem: Given f and b, find u such that

$$\begin{cases} -\Delta u(\boldsymbol{x}) - b(\boldsymbol{x})\,u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\Omega = [0,1]^2$ is the unit square and $\Gamma = \partial\Omega$. We assume u is smooth.

Downwards sweep: We know u on the red nodes. We can use the computed DtN operators to reconstruct u on the blue nodes.



Key novelty: A high order scheme, with razor thin borders between boxes.

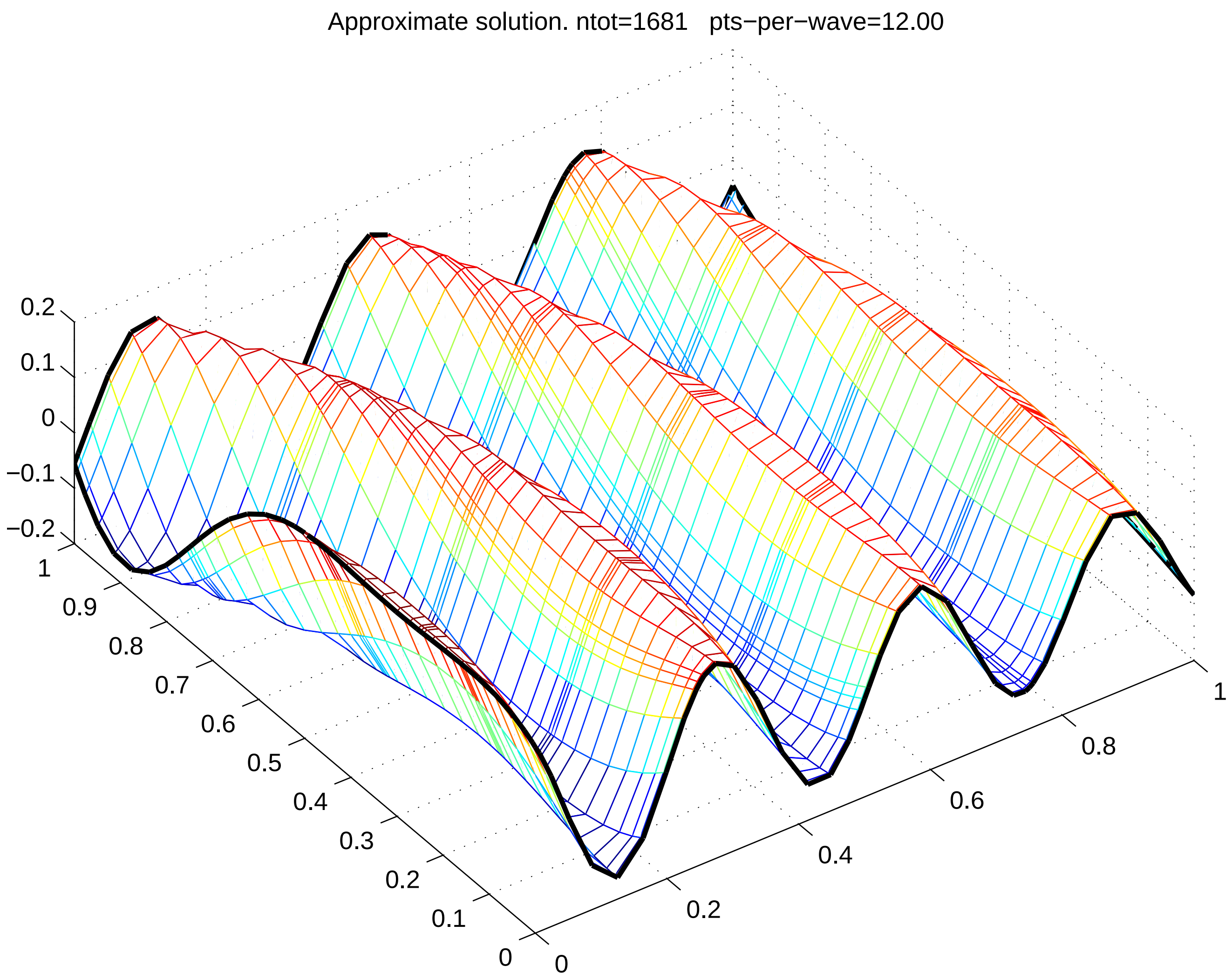
Hierarchical Poincaré-Steklov Method: numerical results

Set $\Omega = [0,1]^2$ and $\Gamma = \partial\Omega$. Consider the problem

$$\begin{cases} -\Delta u(\boldsymbol{x}) - \kappa^2 u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma. \end{cases}$$

We pick f as the restriction of a wave from a point source, $\boldsymbol{x} \mapsto Y_0(\kappa|\boldsymbol{x} - \hat{\boldsymbol{x}}|)$.

We then know the exact solution, $u_{\mathrm{exact}}(\boldsymbol{x}) = Y_0(\kappa|\boldsymbol{x} - \hat{\boldsymbol{x}}|)$.



Hierarchical Poincaré-Steklov Method: numerical results

Set $\Omega = [0,1]^2$ and $\Gamma = \partial\Omega$. Consider the problem

$$\begin{cases} -\Delta u(\boldsymbol{x}) - \kappa^2 u(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma. \end{cases}$$

We pick f as the restriction of a wave from a point source, $\boldsymbol{x} \mapsto Y_0(\kappa|\boldsymbol{x} - \hat{\boldsymbol{x}}|)$.

We then know the exact solution, $u_{\rm exact}(\boldsymbol{x}) = Y_0(\kappa|\boldsymbol{x} - \hat{\boldsymbol{x}}|)$.

The spectral computation on a leaf involves 21×21 points.

κ is chosen so that there are 12 points per wave-length.

p	N	$N_{\rm wave}$	$t_{\rm build}$	$t_{\rm solve}$	$E_{\rm pot}$	$E_{\rm grad}$	M	M/N
			(sec)	(sec)			(MB)	(reals/DOF)
21	6561	6.7	0.23	0.0011	2.56528e-10	1.01490e-08	4.4	87.1
21	25921	13.3	0.92	0.0044	5.24706e-10	4.44184e-08	18.8	95.2
21	103041	26.7	4.68	0.0173	9.49460e-10	1.56699e-07	80.8	102.7
21	410881	53.3	22.29	0.0727	1.21769e-09	3.99051e-07	344.9	110.0
21	1640961	106.7	99.20	0.2965	1.90502e-09	1.24859e-06	1467.2	117.2
21	6558721	213.3	551.32	20.9551	2.84554e-09	3.74616e-06	6218.7	124.3

Error is measured in sup-norm: $e = \max_{\boldsymbol{x} \in \Omega} |u(\boldsymbol{x}) - u_{\rm exact}(\boldsymbol{x})|$.

Note 1: Translation invariance is *not* exploited.

Note 2: The times refer to a simple Matlab implementation executed on a $1k laptop.

Note 3: Keeping a fixed number of points per wave-length works well for this scheme!

Hierarchical Poincaré-Steklov Method: numerical results — $O(N)$ version

Problem	N	$T_{\rm build}$	$T_{\rm solve}$	MB
Laplace	1.7e6	91.68	0.34	1611.19
	6.9e6	371.15	1.803	6557.27
	2.8e7	1661.97	6.97	26503.29
	1.1e8	6894.31	30.67	106731.61
Helmholtz I	1.7e6	62.07	0.202	1611.41
	6.9e6	363.19	1.755	6557.12
	2.8e7	1677.92	6.92	26503.41
	1.1e8	7584.65	31.85	106738.85
Helmholtz II	1.7e6	93.96	0.29	1827.72
	6.9e6	525.92	2.13	7151.60
	2.8e7	2033.91	8.59	27985.41
Helmholtz III	1.7e6	105.58	0.44	1712.11
	6.9e6	510.37	2.085	7157.47
	2.8e7	2714.86	10.63	29632.89

(About six accurate digits in solution.)

Hierarchical Poincaré-Steklov Method

Some observations:

- Has been tested on a broad range of problems.
  - Variable coeffficients.
  - Curved domains.
  - Convection-diffusion, Yukawa, Helmholtz, etc.

  Performance is generally in line with the simple example shown.

- Acceleration to $O(N)$ complexity is done.

  Can easily handle $N \sim 10^8$ on a (good) desktop. Excellent constants.

Hierarchical Poincaré-Steklov Method

Some observations:

- Has been tested on a broad range of problems.
  - Variable coeffficients.
  - Curved domains.
  - Convection-diffusion, Yukawa, Helmholtz, etc.

  Performance is generally in line with the simple example shown.

- Acceleration to $O(N)$ complexity is done.

  Can easily handle $N \sim 10^8$ on a (good) desktop. Excellent constants.

  (Well, excellent in 2D. More work required for 3D...)

Hierarchical Poincaré-Steklov Method

Some observations:

- Has been tested on a broad range of problems.
  - Variable coeffficients.
  - Curved domains.
  - Convection-diffusion, Yukawa, Helmholtz, etc.

  Performance is generally in line with the simple example shown.

- Acceleration to $O(N)$ complexity is done.

  Can easily handle $N \sim 10^8$ on a (good) desktop. Excellent constants.

  (Well, excellent in 2D. More work required for 3D...)

Having a high-accuracy $O(N)$ direct solver opens interesting possibilities:

1. Very fast time-stepping of parabolic problems.
2. Parallel-in-time integration of hyperbolic problems.
3. Coupling of different solvers, for instance “FEM-BEM” coupling.

   Domain decomposition methods. Multiphysics.

Very fast time-stepping of parabolic problems

As a toy example, consider implicit time-stepping of the equation

$$\begin{cases} -\dfrac{\partial u(\boldsymbol{x},t)}{\partial t} = -\,\Delta u, & \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x},t) = f(\boldsymbol{x},t) & \boldsymbol{x} \in \Gamma, \\ u(\boldsymbol{x},0) = h(\boldsymbol{x}) & \boldsymbol{x} \in \Omega. \end{cases}$$

Say, for simplicity, that we use backwards Euler to discretize in time, with

$$\frac{\partial u^n}{\partial t} \approx \frac{1}{k}\left(u^n - e^{n-1}\right).$$

Then for each time-step we need to solve

$$\begin{cases} -\Delta u^n + \dfrac{1}{k}u^n = \dfrac{1}{k}u^{n-1}, & \Omega, \\ u^n = f^n & \Gamma. \end{cases}$$

This is very well suited for our direct solver.

Current work: Investigate stability with better time-stepping schemes (specifically ESDIRK). Numerical experiments are very promising. Extension to *Stokes*, low Reynolds number *Navier-Stokes*, etc.

Parallel-in-time integration of hyperbolic problems: Consider the equation

$$\begin{cases} \dfrac{\partial u(\boldsymbol{x},t)}{\partial t} = B\,u(\boldsymbol{x},t), & \boldsymbol{x}\in\Omega,\ t>0 \\ u(\boldsymbol{x},0) = f(\boldsymbol{x}) & \boldsymbol{x}\in\Omega, \end{cases}$$

where B is a skew-Hermitian operator (e.g. $B = \sqrt{\Delta}$ with Dirichlet/Neumann BC). The solution is

$$u(\boldsymbol{x},t) = \big[\exp(t\,B)\,f\big](\boldsymbol{x}),$$

where $\exp(t\,B)$ is the time-evolution operator. Now suppose that we can approximate the oscillatory function $x \mapsto \exp(ix)$ by a rational function

$$R_M(ix) = \sum_{m=-M}^{M} \frac{b_m}{ix - \alpha_m},$$

where $\{b_m\}$ and $\{\alpha_m\}$ are some complex numbers such that $|R_M(ix)| \leq 1$ for $x \in \mathbb{R}$. We require that

$$\left|e^{ix} - R_M(ix)\right| \leq \delta, \qquad x \in [-\tau\Lambda, \tau\Lambda],$$

where τ is a time step, and where Λ is a “band-width” — in other words, we accurately resolve the parts of B whose spectrum fall in the interval $[-i\Lambda, i\Lambda]$. *Very high accuracy can be attained* – say $\delta = 10^{-10}$ for about 5 – 10 points per wavelength [Beylkin, Haut]. Then approximate

$$\exp(\tau B) \approx \sum_{m=-M}^{M} b_m\,(B - \alpha_m)^{-1}.$$

Notes: The time-step τ *can be large.* Application of $\exp(\tau B)$ is almost instantaneous. Quite high memory demands, but distributed memory is fine.

Parallel-in-time integration of hyperbolic problems: Consider the equation

$$\begin{cases} \dfrac{\partial u(\boldsymbol{x},t)}{\partial t} = B\,u(\boldsymbol{x},t), & \boldsymbol{x}\in\Omega,\ t>0 \\ u(\boldsymbol{x},0) = f(\boldsymbol{x}) & \boldsymbol{x}\in\Omega, \end{cases}$$

where B is a skew-Hermitian operator (e.g. $B=\sqrt{\Delta}$ with Dirichlet/Neumann BC). The solution is

$$u(\boldsymbol{x},t) = \big[\exp(t\,B)\,f\big](\boldsymbol{x}),$$

where $\exp(t\,B)$ is the time-evolution operator. Now suppose that we can approximate the oscillatory function $x\mapsto\exp(ix)$ by a rational function

$$R_M(ix) = \sum_{m=-M}^{M}\frac{b_m}{ix-\alpha_m},$$

where $\{b_m\}$ and $\{\alpha_m\}$ are some complex numbers such that $|R_M(ix)|\leq 1$ for $x\in\mathbb{R}$. We require that

$$\big|e^{ix}-R_M(ix)\big|\leq\delta,\qquad x\in[-\tau\Lambda,\tau\Lambda],$$

where τ is a time step, and where Λ is a "band-width" — in other words, we accurately resolve the parts of B whose spectrum fall in the interval $[-i\Lambda,i\Lambda]$. *Very high accuracy can be attained* – say $\delta=10^{-10}$ for about 5 – 10 points per wavelength [Beylkin, Haut]. Then approximate

$$\exp(\tau B)\approx\sum_{m=-M}^{M}b_m\,(B-\alpha_m)^{-1}.$$

Notes: The time-step τ *can be large.* Application of $\exp(\tau B)$ is almost instantaneous. Quite high memory demands, but distributed memory is fine. ***Parallel in time!***

Parallel-in-time integration of hyperbolic problems: Consider the equation

$$\begin{cases} \dfrac{\partial u(\boldsymbol{x},t)}{\partial t} = B\,u(\boldsymbol{x},t), & \boldsymbol{x}\in\Omega,\ t>0 \\ u(\boldsymbol{x},0) = f(\boldsymbol{x}) & \boldsymbol{x}\in\Omega, \end{cases}$$

where B is a skew-Hermitian operator (e.g. $B=\sqrt{\Delta}$ with Dirichlet/Neumann BC). The solution is

$$u(\boldsymbol{x},t) = \big[\exp(t\,B)\,f\big](\boldsymbol{x}),$$

where $\exp(t\,B)$ is the time-evolution operator. Now suppose that we can approximate the oscillatory function $x \mapsto \exp(ix)$ by a rational function

$$R_M(ix) = \sum_{m=-M}^{M} \frac{b_m}{ix-\alpha_m},$$

where $\{b_m\}$ and $\{\alpha_m\}$ are some complex numbers such that $|R_M(ix)| \leq 1$ for $x\in\mathbb{R}$. We require that

$$\big|e^{ix} - R_M(ix)\big| \leq \delta, \qquad x\in[-\tau\Lambda,\tau\Lambda],$$

where τ is a time step, and where Λ is a "band-width" — in other words, we accurately resolve the parts of B whose spectrum fall in the interval $[-i\Lambda, i\Lambda]$. *Very high accuracy can be attained* – say $\delta = 10^{-10}$ for about 5 – 10 points per wavelength [Beylkin, Haut]. Then approximate

$$\exp(\tau B) \approx \sum_{m=-M}^{M} b_m\,(B-\alpha_m)^{-1}.$$

Notes: The time-step τ *can be large.* Application of $\exp(\tau B)$ is almost instantaneous. Quite high memory demands, but distributed memory is fine. ***Parallel in time!***

Current project: Shallow water equations on cubed sphere at LANL.

Hierarchical Poincaré-Steklov Method: Free space scattering

Consider the acoustic scattering problem

$$\begin{cases} -\Delta u_{\rm out}(\boldsymbol{x}) - \kappa^2\,(1 - b(\boldsymbol{x}))\,u_{\rm out}(\boldsymbol{x}) = \; -\kappa^2\,b(\boldsymbol{x})\,u_{\rm in}(\boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^2 \\ \lim\limits_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}\big(\partial_{|\boldsymbol{x}|} u_{\rm out}(\boldsymbol{x}) - i\kappa\,u_{\rm out}(\boldsymbol{x})\big) = 0 \end{cases}$$

Suppose that b is a smooth scattering potential such that for some rectangle Ω, we have

$$\text{support}(b) \subset \Omega.$$

We also suppose that $u_{\rm in}$ satisfies

$$-\Delta u_{\rm in}(\boldsymbol{x}) - \kappa^2\,u_{\rm in}(\boldsymbol{x}) = 0, \qquad \boldsymbol{x} \in \Omega.$$

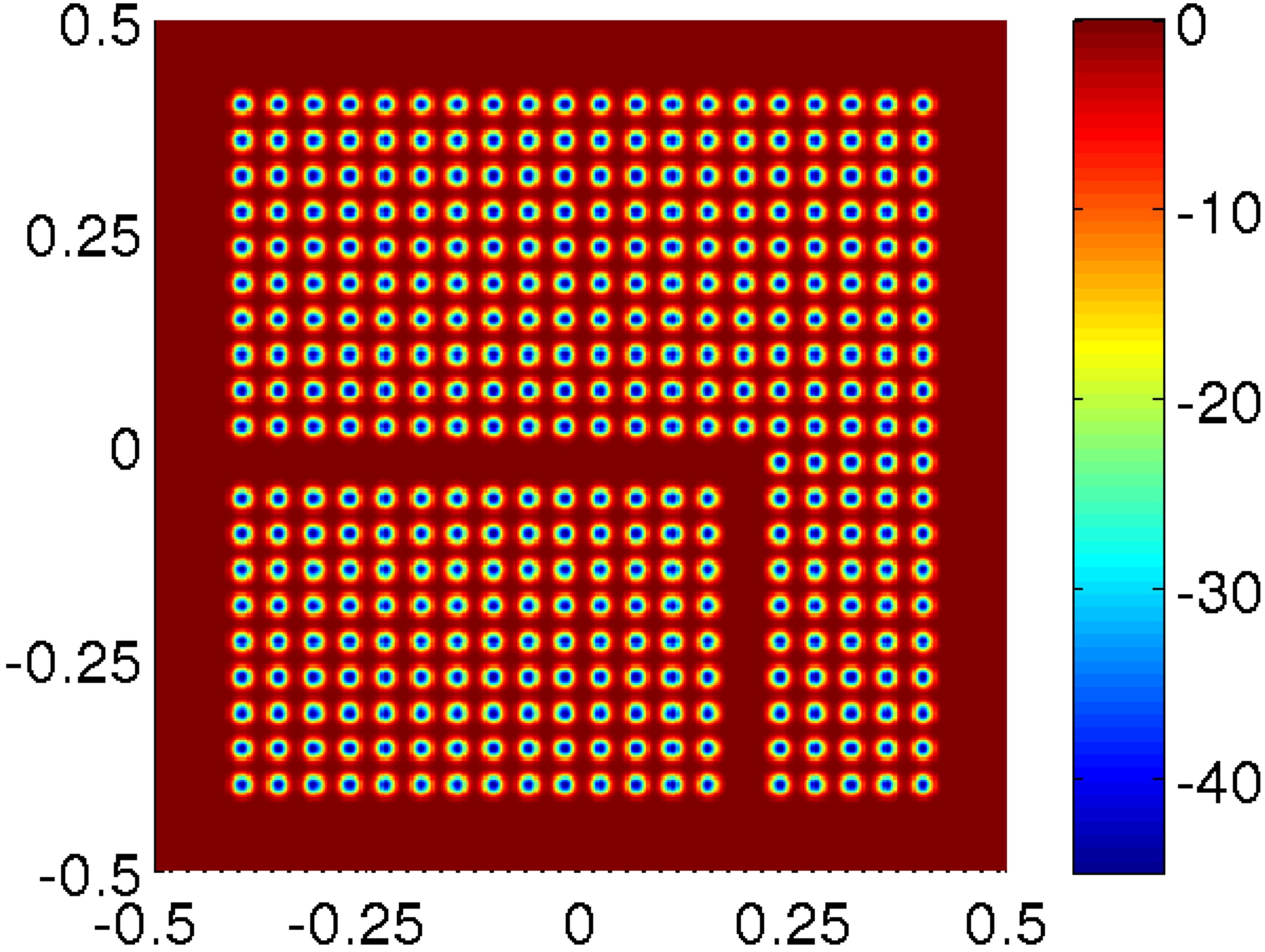
Solution strategy ("FEM-BEM coupling"):

1. Use HPS method to construct the DtN map for the variable coefficient problem in Ω.
2. Use boundary integral equation techniques to find the DtN map for the constant coefficient problem on $\Omega^{\rm c}$.
3. Glue the two DtN maps together and you're all set!

Joint work with Adrianna Gillman and Alex Barnett.

Example: Free space scattering $\begin{cases} -\Delta u_{\rm out}(\boldsymbol{x}) - \kappa^2 (1-b(\boldsymbol{x}))\, u_{\rm out}(\boldsymbol{x}) = -\kappa^2\, b(\boldsymbol{x})\, u_{\rm in}(\boldsymbol{x}) \\ \lim_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}\big(\partial_{|\boldsymbol{x}|} u_{\rm out}(\boldsymbol{x}) - i\kappa\, u_{\rm out}(\boldsymbol{x})\big) = 0 \end{cases}$

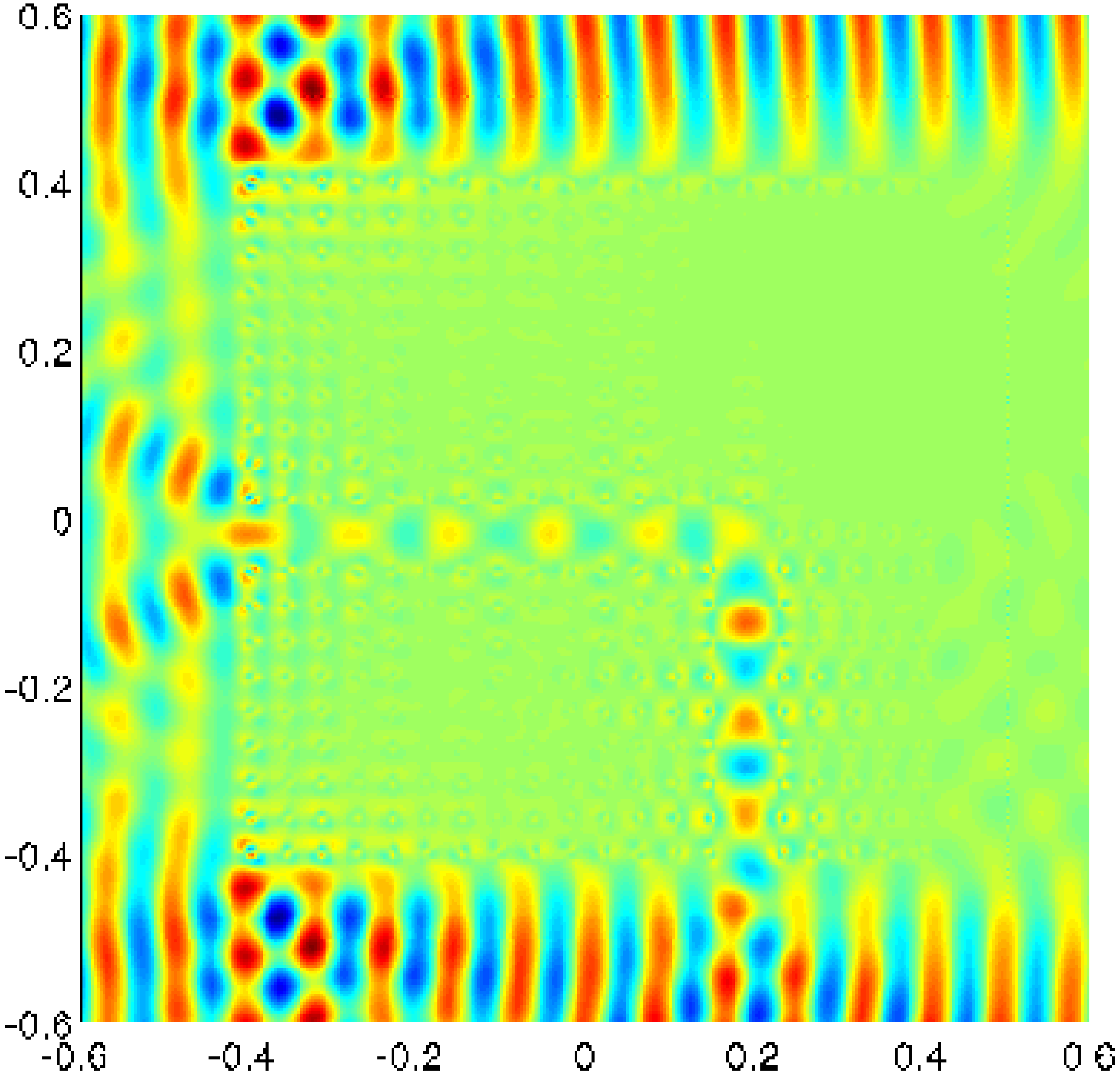
The scattering potential b — a "photonic crystal" with a wave guide.



Relative accuracy 10^{-7} with $N \approx 10^6$. Time to build solution operator is about 60 seconds. Time to build the solution for a given incident wave is about one second.

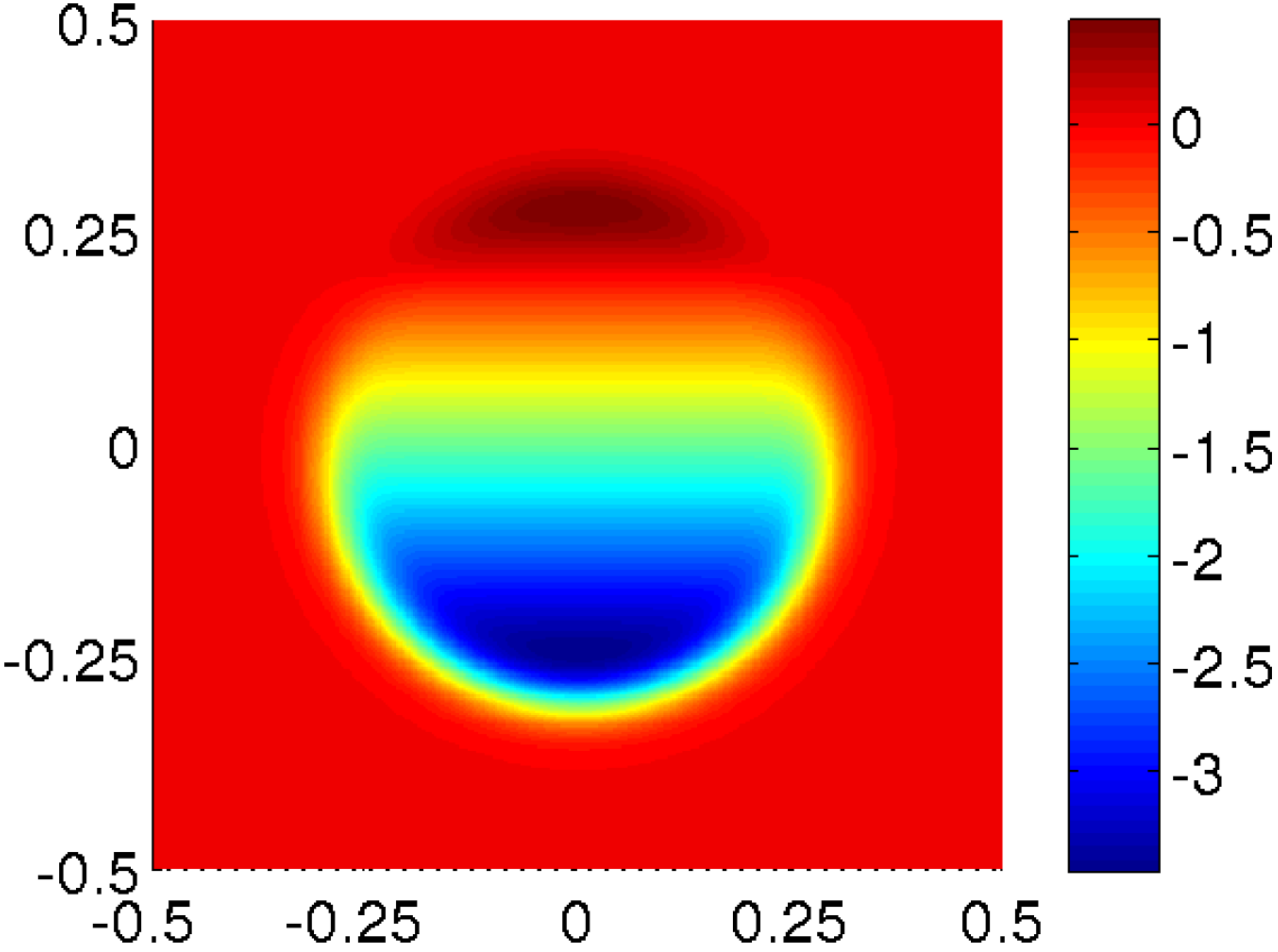
Example: Free space scattering $\begin{cases} -\Delta u_{\rm out}(\boldsymbol{x}) - \kappa^2\,(1 - b(\boldsymbol{x}))\,u_{\rm out}(\boldsymbol{x}) = -\kappa^2\,b(\boldsymbol{x})\,u_{\rm in}(\boldsymbol{x}) \\ \lim\limits_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}\big(\partial_{|\boldsymbol{x}|}u_{\rm out}(\boldsymbol{x}) - i\kappa\,u_{\rm out}(\boldsymbol{x})\big) = 0 \end{cases}$

The total field $u = u_{\rm in} + u_{\rm out}$ (resulting from an incoming plane wave $u_{\rm in}(x) = \cos(\kappa\,x_1)$).



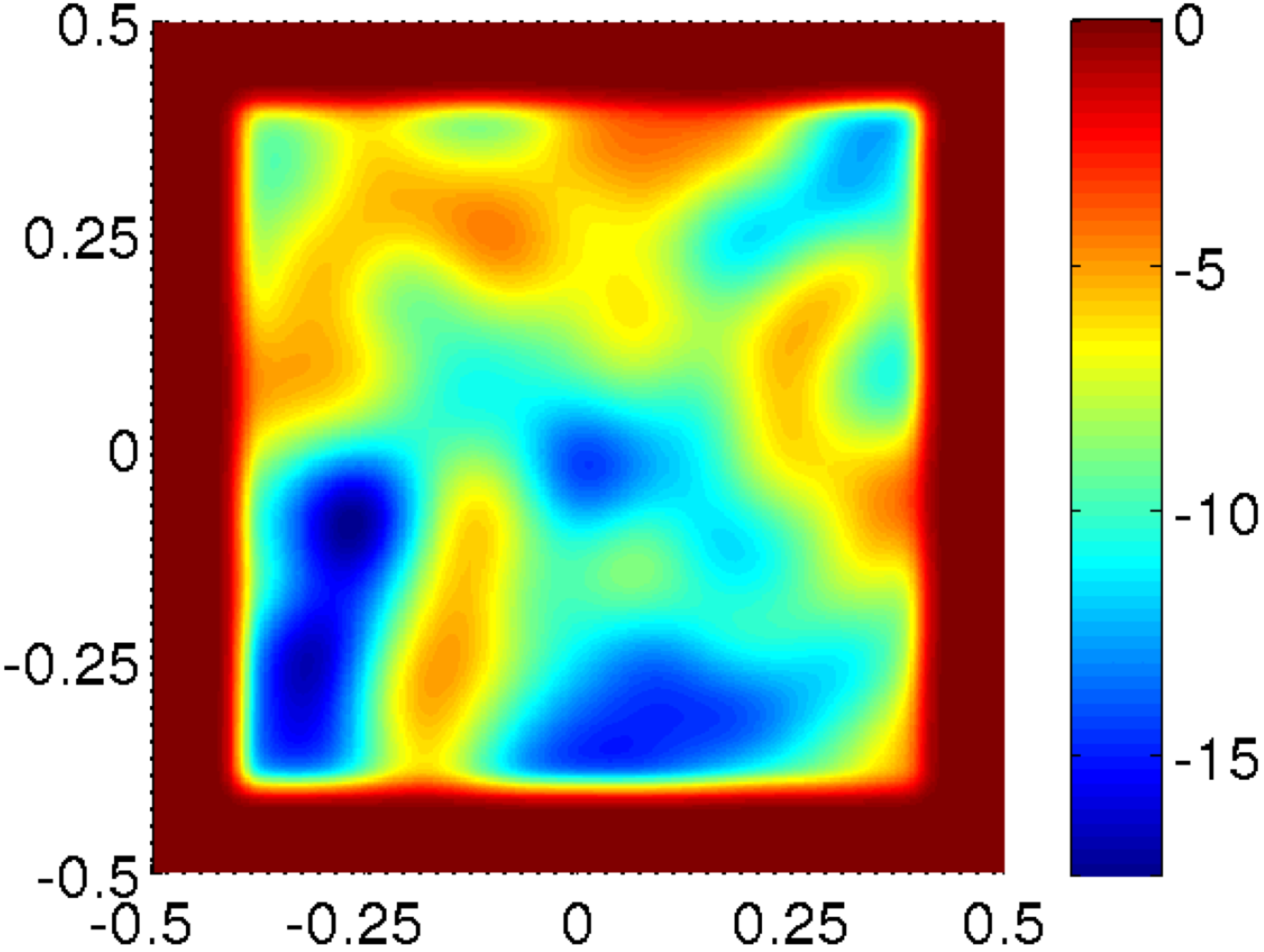
Example: Free space scattering $\begin{cases} -\Delta u_{\rm out}(\boldsymbol{x}) - \kappa^2\,(1 - b(\boldsymbol{x}))\,u_{\rm out}(\boldsymbol{x}) = -\kappa^2\,b(\boldsymbol{x})\,u_{\rm in}(\boldsymbol{x}) \\ \lim_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}\big(\partial_{|\boldsymbol{x}|}u_{\rm out}(\boldsymbol{x}) - i\kappa\,u_{\rm out}(\boldsymbol{x})\big) = 0 \end{cases}$

The scattering potential b — a graded "lens."



Example: Free space scattering
$$\begin{cases} -\Delta u_{\rm out}(\boldsymbol{x}) - \kappa^2\,(1 - b(\boldsymbol{x}))\,u_{\rm out}(\boldsymbol{x}) = -\kappa^2\,b(\boldsymbol{x})\,u_{\rm in}(\boldsymbol{x}) \\ \lim_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}\big(\partial_{|\boldsymbol{x}|}u_{\rm out}(\boldsymbol{x}) - i\kappa\,u_{\rm out}(\boldsymbol{x})\big) = 0 \end{cases}$$

The scattering potential b — a "bathroom glass".



Main themes

Randomized algorithms for low-rank approximation:

- Very simple to implement.
- Exploits parallelism through calls to `dgemm`, `dgeqrf`, etc.
- No need to specify rank in advance, just a tolerance.

Randomized UTV decomposition:

- Effective for *any* rank, including full rank.
- Very close to SVD accuracy when used for low-rank approximation.
- Easy to parallelize, execute out-of-core, etc. Perfect for GPUs.
- As fast as column pivoted QR, or faster. *Much* faster than SVD.

  Excellent general-purpose factorization.

The “Hierarchical Poincaré-Steklov” fast direct solver.

- Based on high-order local spectral discretization.
- Plays nicely with nested dissection type solvers. (Including $O(N)$ versions.)
- Enables parallel-in-time integration of hyperbolic PDEs.
- Enables “FEM-BEM coupling”, multiphysics, resonant wave problems, etc.

Tutorials, summer schools, etc:

- 2009: NIPS tutorial lecture, Vancouver, 2009. Online video available.
- 2014: CBMS summer school at Dartmouth College. 10 lectures on YouTube.
- 2016: Park City Math Institute (IAS): *The Mathematics of Data.*

Software packages:

- ID: `http://tygert.com/software.html`
- RSVDPACK: `https://github.com/sergeyvoronin` (expansions are in progress)
- Column pivoted QR: `https://github.com/flame/hqrrp` (much faster than LAPACK!)

Papers (see also `http://amath.colorado.edu/faculty/martinss/main_publications.html`):

- P.G. Martinsson, V. Rokhlin, and M. Tygert, “A randomized algorithm for the approximation of matrices”. 2007 report YALE-CS-1361; 2011 paper in ACHA.
- N. Halko, P.G. Martinsson, J. Tropp, “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions.” *SIAM Review*, 2011.
- E. Liberty, F. Woolfe, P.G. Martinsson, V. Rokhlin, and M. Tygert, “Randomized algorithms for the low-rank approximation of matrices”. *PNAS*, **104**(51), 2007.
- P.G. Martinsson, “A fast randomized algorithm for computing a Hierarchically Semi-Separable representation of a matrix”. *SIMAX*, **32**(4), 2011.
- P.G. Martinsson, “Compressing structured matrices via randomized sampling,” SISC **38**(4), 2016.
- P.G. Martinsson, G. Quintana-Ortí, N. Heaver, and R. van de Geijn, “Householder QR Factorization With Randomization for Column Pivoting.” To appear in SISC.