Fast Rank-Adaptive CUR and Its Extension to Kernel Matrices

Gunnar Martinsson

Dept. of Mathematics & Oden Institute for Computational Sciences and Engineering

University of Texas at Austin

Collaborators: Robert van de Geijn, Abinand Gopal, Francisco Igual, Yuji Nakatsukasa, Nathaniel Pritchard, Gregorio Quintana-Ortí, Taejun Park, Vladimir Rokhlin, Joel Tropp, Mark Tygert, Heather Wilber.

Current and recent students/postdocs: Paul Beckman, Yunhui Cai, Chao Chen, Simon Dirckx, Yijun Dong, Michael Han, Joseph Kump, James Levitt, Kate Pearce, Anna Yesypenko.

Research support by:

Outline

Basic algorithmic pattern for low rank approximation:

- Randomized SVD. *[Brief review]*

Enhancements to the basic pattern:

- Fast a posteriori error estimation. *[Newish]*
- Adaptive rank determinination.
- Downdating of a small sketch.
- CUR and interpolatory decompositions.
- Putting all the enhancements together: Iterative CUR. *[Newish]*

Extension to kernel matrices and continuum problems. *[New] [time permitting . . .]*

- Suppose $\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{y}_j)$ for some kernel k and point sets $\{\boldsymbol{x}_i\}_{i=1}^m$ and $\{\boldsymbol{y}_j\}_{j=1}^n$. How do you compute a low rank approximation to $\mathbf{A}$ without forming the entire matrix?

Randomized low rank approximation: RSVD

Objective: Given an $m \times n$ matrix $\mathbf{A}$, find an approximate rank-k partial SVD:

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

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

(A) *Randomized sketching:*

A.1 Draw an $n \times k$ Gaussian random "test" matrix $\mathbf{\Omega}$. `Omega = randn(n,k)`

A.2 Form the $m \times k$ "sample" matrix $\mathbf{Y} = \mathbf{A\Omega}$. `Y = A * Omega`

A.3 Form an $m \times k$ orthonormal matrix $\mathbf{Q}$ such that $\text{ran}(\mathbf{Q}) = \text{ran}(\mathbf{Y})$. `[Q, ~] = qr(Y)`

(B) *Deterministic post-processing:*

B.1 Form the $k \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$. `B = Q' * A`

B.2 Form the full SVD of the small matrix $\mathbf{B}$: $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$. `[Uhat, Sigma, V] = svd(B,'econ')`

B.3 Form the matrix $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$. `U = Q * Uhat`

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

Stage B is exact: $\|\mathbf{A} - \mathbf{Q}\underbrace{\mathbf{Q}^*\mathbf{A}}_{=\mathbf{B}}\| = \|\mathbf{A} - \mathbf{Q}\underbrace{\mathbf{B}}_{=\hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*}\| = \|\mathbf{A} - \underbrace{\mathbf{Q}\hat{\mathbf{U}}}_{=\mathbf{U}}\mathbf{D}\mathbf{V}^*\| = \|\mathbf{A} - \mathbf{UDV}^*\|$.

Randomized low rank approximation: RSVD

Input: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 5$).	
Output: Rank-$(k+p)$ factors $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate SVD $\mathbf{A} \approx \mathbf{UDV}^*$.	
(1) Draw an $n \times (k+p)$ random matrix $\mathbf{\Omega}$.	(4) Form the small matrix $\mathbf{B} = \mathbf{Q}^* \mathbf{A}$.
(2) Form the $m \times (k+p)$ sample matrix $\mathbf{Y} = \mathbf{A\Omega}$.	(5) Factor the small matrix $\mathbf{B} = \hat{\mathbf{U}}\mathbf{DV}^*$.
(3) Compute an ON matrix $\mathbf{Q}$ s.t. $\mathbf{Y} = \mathbf{QR}$.	(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

Key points:

- High practical speed — interacts with $\mathbf{A}$ only through matrix-matrix multiplication.
- Communication efficient: Out-of-core, GPU, distributed memory, ...

Randomized low rank approximation: RSVD

Input: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 5$).	
Output: Rank-$(k+p)$ factors $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate SVD $\mathbf{A} \approx \mathbf{UDV}^*$.	
(1) Draw an $n \times (k+p)$ random matrix $\mathbf{\Omega}$.	(4) Form the small matrix $\mathbf{B} = \mathbf{Q}^* \mathbf{A}$.
(2) Form the $m \times (k+p)$ sample matrix $\mathbf{Y} = \mathbf{A\Omega}$.	(5) Factor the small matrix $\mathbf{B} = \hat{\mathbf{U}}\mathbf{DV}^*$.
(3) Compute an ON matrix $\mathbf{Q}$ s.t. $\mathbf{Y} = \mathbf{QR}$.	(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

Key points:

- High practical speed — interacts with $\mathbf{A}$ only through matrix-matrix multiplication.
- Communication efficient: Out-of-core, GPU, distributed memory, ...
- Consider the problem of computing the dominant k eigenvectors/eigenvalues of a dense matrix of size $m \times n$. Reduction in complexity from $O(mnk)$ to $O(mn)$.

  The key is to use a *Fast Johnson-Lindenstrauss transform*.
  - Randomized trigonometric transforms (FFT, Hadamard, etc). Cost is $O(mn \log(k))$.
  - Chains of Given's rotations ("Kac's random walk"). Cost is $O(mn \log(k))$.
  - "Sparse sign matrix". Place r random entries in each row of $\mathbf{\Omega}$. (Say $r = 2$ or $r = 4$.)
    Cost is now $O(mn)$!

  Practical acceleration is achieved at ordinary matrix sizes.

Randomized low rank approximation: RSVD

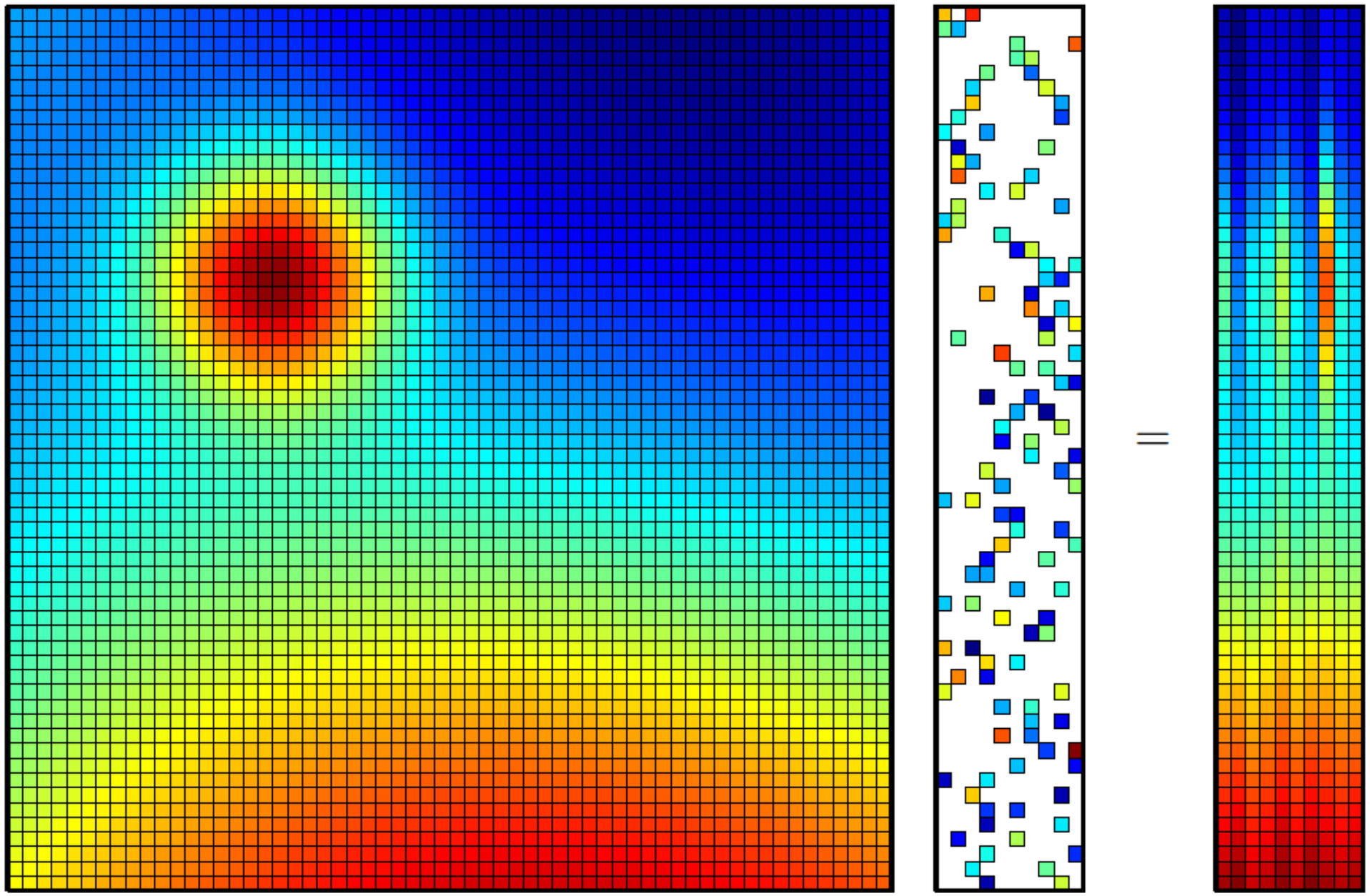
Input: An $m \times n$ matrix **A**, a target rank k, and an over-sampling parameter p (say $p = 5$).

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

(1) Draw a

(2) Form th

(3) Compu



Key points:

- High pra on.
- Commu
- Conside of a dense n

  The key

  - Rando
  - Chains
  - "Spars

    Cost is

Practical acceleration is achieved at ordinary matrix sizes.

Randomized low rank approximation: RSVD

Input: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 5$).	
Output: Rank-$(k+p)$ factors $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate SVD $\mathbf{A} \approx \mathbf{UDV}^*$.	
(1) Draw an $n \times (k+p)$ random matrix $\mathbf{\Omega}$.	(4) Form the small matrix $\mathbf{B} = \mathbf{Q}^* \mathbf{A}$.
(2) Form the $m \times (k+p)$ sample matrix $\mathbf{Y} = \mathbf{A\Omega}$.	(5) Factor the small matrix $\mathbf{B} = \hat{\mathbf{U}}\mathbf{DV}^*$.
(3) Compute an ON matrix $\mathbf{Q}$ s.t. $\mathbf{Y} = \mathbf{QR}$.	(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

Key points:

- High practical speed — interacts with $\mathbf{A}$ only through matrix-matrix multiplication.
- Communication efficient: Out-of-core, GPU, distributed memory, ...
- Single pass algorithms have been developed for *streaming environments.*

  The idea is that you are allowed to observe each matrix element only once.

  You cannot store the matrix. *Not possible with deterministic methods!*

Outline

Basic algorithmic pattern for low rank approximation:

- Randomized SVD. *[Brief review]*

Enhancements to the basic pattern:

- Fast a posteriori error estimation. *[Newish]*
- Adaptive rank determinination.
- Downdating of a small sketch.
- CUR and interpolatory decompositions.
- Putting all the enhancements together: Iterative CUR. *[Newish]*

Extension to kernel matrices and continuum problems. *[New]*

- Suppose $\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{y}_j)$ for some kernel k and point sets $\{\boldsymbol{x}_i\}_{i=1}^m$ and $\{\boldsymbol{y}_j\}_{j=1}^n$. How do you compute a low rank approximation to $\mathbf{A}$ without forming the entire matrix?

Recent work 1 (out of 5): A posteriori error estimation

Question: How estimate the error in a specific instantiation of a randomized algorithm?

Our framing is that we are given an $m \times n$ matrix $\mathbf{A}$, and use the randomized range finder to build an approximation:

- Draw an $n \times \ell$ random matrix $\mathbf{\Omega}$, for say $\ell = k + 10$ or $\ell = 2k$.
  Note: We may use a very fast, but possibly less reliable, class of random matrices.
- Form $\mathbf{Y} = \mathbf{A\Omega}$, and perform QR factorization $[\mathbf{Q}, \sim] = \mathtt{qr}(\mathbf{Y}, 0)$.
- Use $\mathbf{A}_{\text{approx}} = \mathbf{Q}\left(\mathbf{Q}^*\mathbf{A}\right)$ as the approximation.

Our objective is to estimate the error

$$e = \|\mathbf{A} - \mathbf{A}_{\text{approx}}\|$$

using only the information at hand!

Why?

- Enable the use of "risky" random dimension reducing maps.
  Cf. the "responsibly reckless" mode of computing, in Dongarra's terminology.
- The numerical rank of $\mathbf{A}$ may not be known in advance.
- Need bounds and estimates that involve *only quantities actually at hand.*

Recent work 1 (out of 5): A posteriori error estimation

Question: How estimate the error in a specific instantiation of a randomized algorithm?

Answer: You can very inexpensively compute a *certificate of accuracy*. To illustrate, consider the RSVD: We draw a random matrix $\mathbf{\Omega}$, set $\mathbf{Y} = \mathbf{A\Omega}$, and approximate $\mathbf{A}$ via

$$\mathbf{A}_{\text{approx}} = \mathbf{Q}\mathbf{Q}^*\mathbf{A},$$

where $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{A}, 0)$. We seek to estimate the error

$$e = \|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}.$$

The idea is to draw a thin slice of a Gaussian, $\mathbf{G} \in \mathbb{R}^{n\times q}$, for $q = 10$ say, that is *quarantined* from the construction of $\mathbf{A}_{\text{approx}}$ and is used purely to estimate the error. Evaluate $\mathbf{C} = \mathbf{AG}$. Then use that for any matrix $\mathbf{H}$, we have

$$\mathbb{E}\big[\|\mathbf{HG}\|_{\text{F}}^2\big] = q\|\mathbf{H}\|_{\text{F}}^2.$$

Setting $\mathbf{H} = \mathbf{A} - \mathbf{A}_{\text{approx}}$, we use the estimate

$$\|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}^2 \approx \frac{1}{q}\|(\mathbf{A} - \mathbf{A}_{\text{approx}})\mathbf{G}\|_{\text{F}}^2 = \frac{1}{q}\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^*)\mathbf{AG}\|_{\text{F}}^2 = \frac{1}{q}\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^*)\mathbf{C}\|_{\text{F}}^2.$$

Very reliable. Inexpensive. *Martinsson, Tropp, Acta Numerica 2020, Sec. 12.2.*

We can do better still. Observe that e does not depend on $\mathbf{Q}$, since $\mathbf{A}_{\text{approx}} = \mathbf{Y}\left(\mathbf{Y}^{\dagger}\mathbf{A}\right)$.

Recent work 1 (out of 5): A posteriori error estimation

Question: How estimate the error in a specific instantiation of a randomized algorithm?

Recall from previous slide: Draw a random matrix $\mathbf{\Omega}$, set $\mathbf{Y} = \mathbf{A\Omega}$, approximate $\mathbf{A}$ via $\mathbf{A}_{\text{approx}} = \mathbf{QQ}^*\mathbf{A}$, where $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y}, 0)$. We seek to estimate $e = \|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}$. To get a *certificate of accuracy*, we draw $\mathbf{G}$, set $\mathbf{C} = \mathbf{AG}$, and use $e^2 \approx \frac{1}{q}\|(\mathbf{I} - \mathbf{QQ}^*)\mathbf{C}\|_{\text{F}}^2$.

Recent work 1 (out of 5): A posteriori error estimation

Question: How estimate the error in a specific instantiation of a randomized algorithm?

Recall from previous slide: Draw a random matrix $\boldsymbol{\Omega}$, set $\mathbf{Y} = \mathbf{A}\boldsymbol{\Omega}$, approximate $\mathbf{A}$ via $\mathbf{A}_{\text{approx}} = \mathbf{Q}\mathbf{Q}^*\mathbf{A}$, where $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y}, 0)$. We seek to estimate $e = \|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}$. To get a *certificate of accuracy*, we draw $\mathbf{G}$, set $\mathbf{C} = \mathbf{A}\mathbf{G}$, and use $e^2 \approx \frac{1}{q}\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^*)\mathbf{C}\|_{\text{F}}^2$.

Acceleration via fast sketching: Observe that $\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^*)\mathbf{C}\|_{\text{F}}^2$ is the minimal error when seeking to fit $\mathbf{C}$ in $\text{ran}(\mathbf{Q}) = \text{ran}(\mathbf{A}\boldsymbol{\Omega}) = \text{ran}(\mathbf{Y})$. So

$$\|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}^2 \approx \frac{1}{q}\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^*)\mathbf{C}\|_{\text{F}}^2 = \frac{1}{q} \inf_{\mathbf{M} \in \mathbb{R}^{\ell \times q}} \|\mathbf{Y}\mathbf{M} - \mathbf{C}\|_{\text{F}}^2.$$

Next, we *sketch* the least squares problem. Draw FJLT $\boldsymbol{\Psi} \in \mathbb{R}^{m \times s}$, with $s = O(k)$, then

$$\|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}^2 \approx \frac{1}{q}\frac{m}{s} \inf_{\mathbf{M} \in \mathbb{R}^{\ell \times q}} \|\boldsymbol{\Psi}^*\big(\mathbf{Y}\mathbf{M} - \mathbf{C}\big)\|_{\text{F}}^2.$$

So in the end, all we need to do is to approximately solve the least squares problem

$$\underset{s \times \ell}{\big(\boldsymbol{\Psi}^*\mathbf{Y}\big)} \quad \underset{\ell \times q}{\mathbf{M}} \quad = \underset{s \times q}{\big(\boldsymbol{\Psi}^*\mathbf{C}\big)}$$

The total extra cost (beyond the cost of RSVD):

- q extra matvecs. (Think $q = 10$.) Can be done as a block.
- Sketching step – evaluate $\boldsymbol{\Psi}^*\mathbf{Y}$ and $\boldsymbol{\Psi}^*\mathbf{C}$. Cost $O(m\ell)$.
- Solve small least squares problem. Cost $O(s\,\ell^2) = O(k^3)$.

Recent work 1 (out of 5): A posteriori error estimation

Question: How estimate the error in a specific instantiation of a randomized algorithm?

Recall from previous slide: Draw a random matrix $\mathbf{\Omega}$, set $\mathbf{Y} = \mathbf{A\Omega}$, approximate $\mathbf{A}$ via $\mathbf{A}_{\text{approx}} = \mathbf{QQ}^*\mathbf{A}$, where $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y}, 0)$. We seek to estimate $e = \|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}$. To get a *certificate of accuracy*, we draw $\mathbf{G}$, set $\mathbf{C} = \mathbf{AG}$, and use $e^2 \approx \frac{1}{q}\|(\mathbf{I} - \mathbf{QQ}^*)\mathbf{C}\|_{\text{F}}^2$.

Acceleration via fast sketching: Observe that $\|(\mathbf{I} - \mathbf{QQ}^*)\mathbf{C}\|_{\text{F}}^2$ is the minimal error when seeking to fit $\mathbf{C}$ in $\text{ran}(\mathbf{Q}) = \text{ran}(\mathbf{A\Omega}) = \text{ran}(\mathbf{Y})$. So

$$\|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}^2 \approx \frac{1}{q}\|(\mathbf{I} - \mathbf{QQ}^*)\mathbf{C}\|_{\text{F}}^2 = \frac{1}{q} \inf_{\mathbf{M} \in \mathbb{R}^{\ell \times q}} \|\mathbf{YM} - \mathbf{C}\|_{\text{F}}^2.$$

Next, we *sketch* the least squares problem. Draw FJLT $\mathbf{\Psi} \in \mathbb{R}^{m \times s}$, with $s = O(k)$, then

$$\|\mathbf{A} - \mathbf{A}_{\text{approx}}\|_{\text{F}}^2 \approx \frac{1}{q}\frac{m}{s} \inf_{\mathbf{M} \in \mathbb{R}^{\ell \times q}} \|\mathbf{\Psi}^*\big(\mathbf{YM} - \mathbf{C}\big)\|_{\text{F}}^2.$$

So in the end, all we need to do is to approximately solve the least squares problem

$$\underset{s \times \ell}{\big(\mathbf{\Psi}^*\mathbf{Y}\big)} \quad \underset{\ell \times q}{\mathbf{M}} \quad = \underset{s \times q}{\big(\mathbf{\Psi}^*\mathbf{C}\big)}$$

- *Very* cheap error estimation. High accuracy.
- Involves only available data.
- No need to even form $\mathbf{Q}$! Can be done as soon as $\mathbf{A\Omega}$ is available.

With Yuji Nakatsukasa. Forthcoming…

Recent work 1 (out of 5): A posteriori error estimation

Additional compelling techniques presented in:

E.N. Epperly, J. Tropp, “Efficient Error and Variance Estimation for Randomized Matrix Computations”, SISC. **46**(1), 2024.

“Leave-one-out” error estimator for randomized low-rank approximations.

“Jackknife resampling method” for estimating the variance of the output of a number of randomized matrix computations.

Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (gives correct asymptotic complexity):

Simply try with $k = 1$ first.

Use an a posteriori error estimator to check if the tolerance is met.

If not, then keep doubling, $k = 2, 4, 8, 16, \ldots$

Gives correct asymptotics. But quite wasteful and slow.

Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

For instance, fix a block size b (say $b = 25$), and a requested tolerance ε.

Build a rank-b factorization of **A** using the RSVD.

Estimate the error using an a posteriori error estimator.

If the tolerance is not met, then use RSVD again to build a rank-b factorization of *the residual* and add the new basis vectors to the b you got in the first step.

Etc.

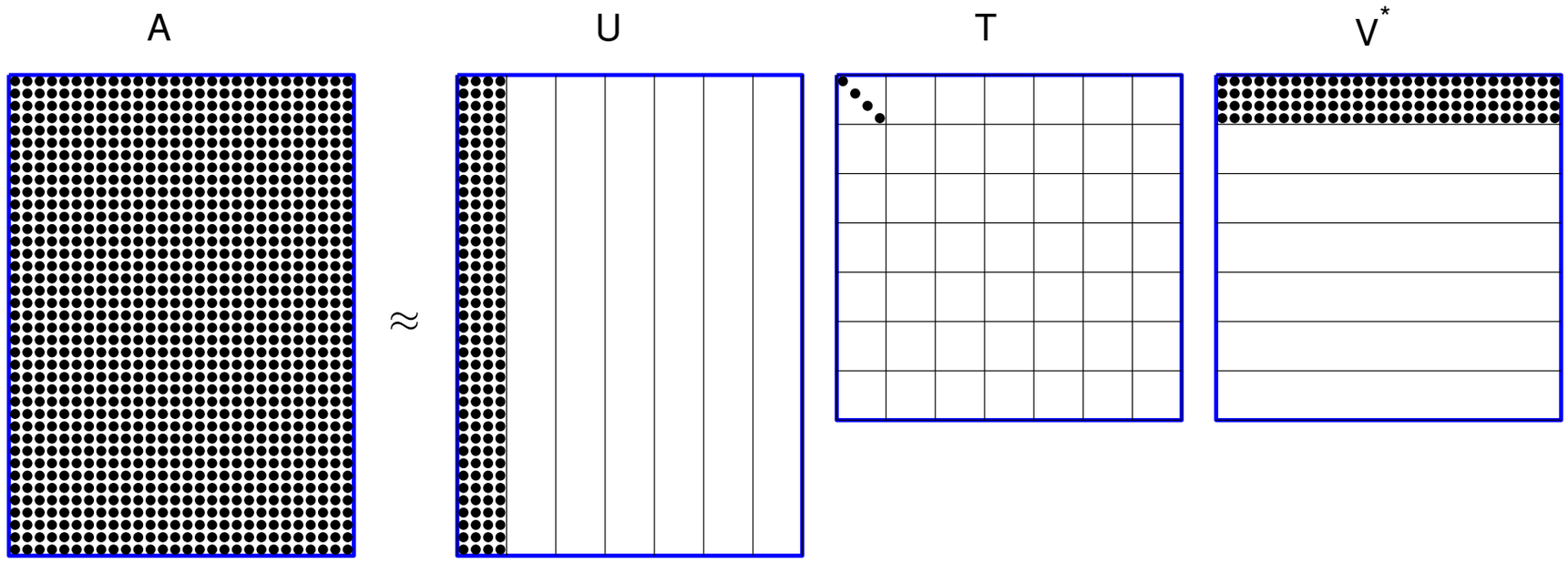
Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

Illustration for block size $b = 4$:



In the first step, compute a rank 4 approximation using the SVD.

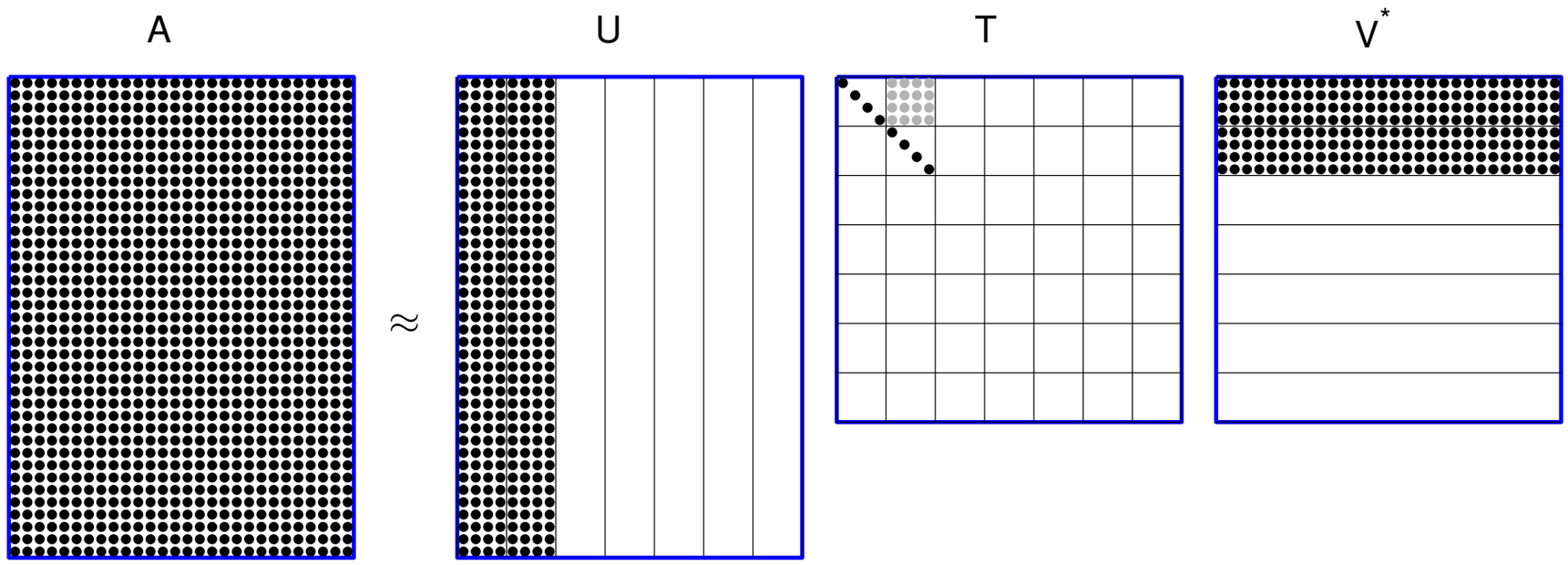
Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

Illustration for block size $b = 4$:



If the tolerance was not met, do a rank-4 RSVD on the residual to get rank-8 approximation.

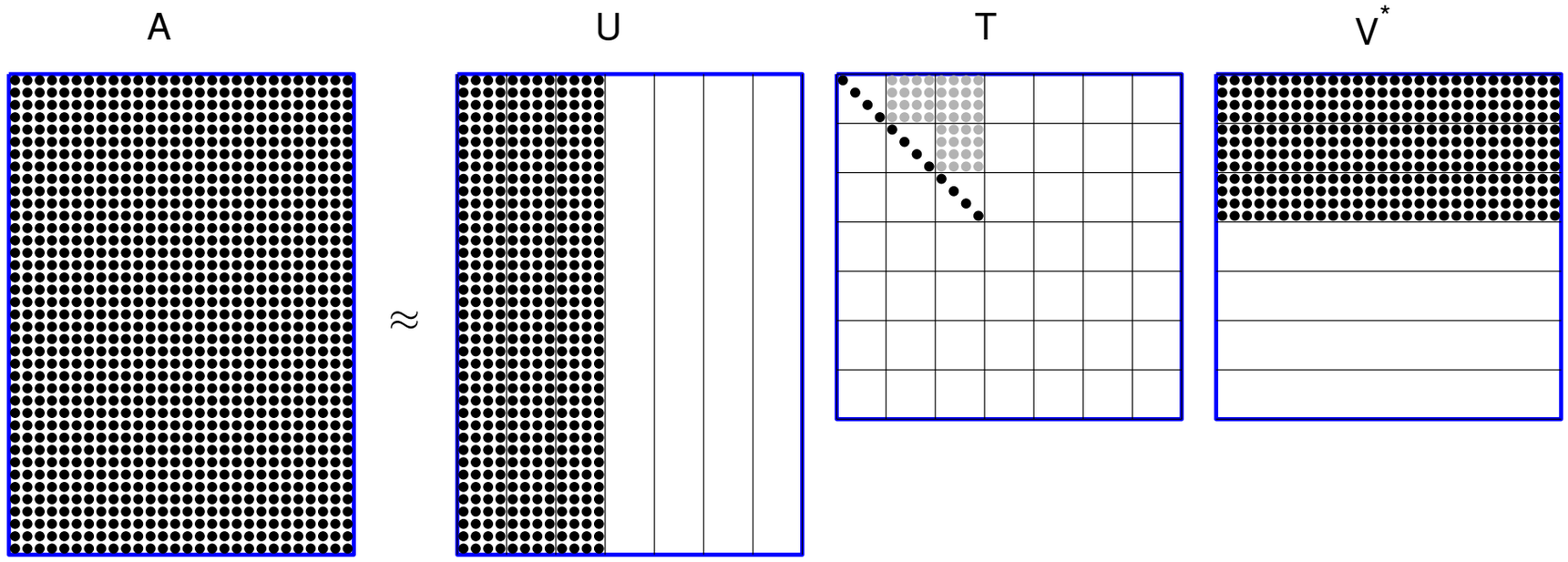
Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

Illustration for block size $b = 4$:



The rank-12 approximation.

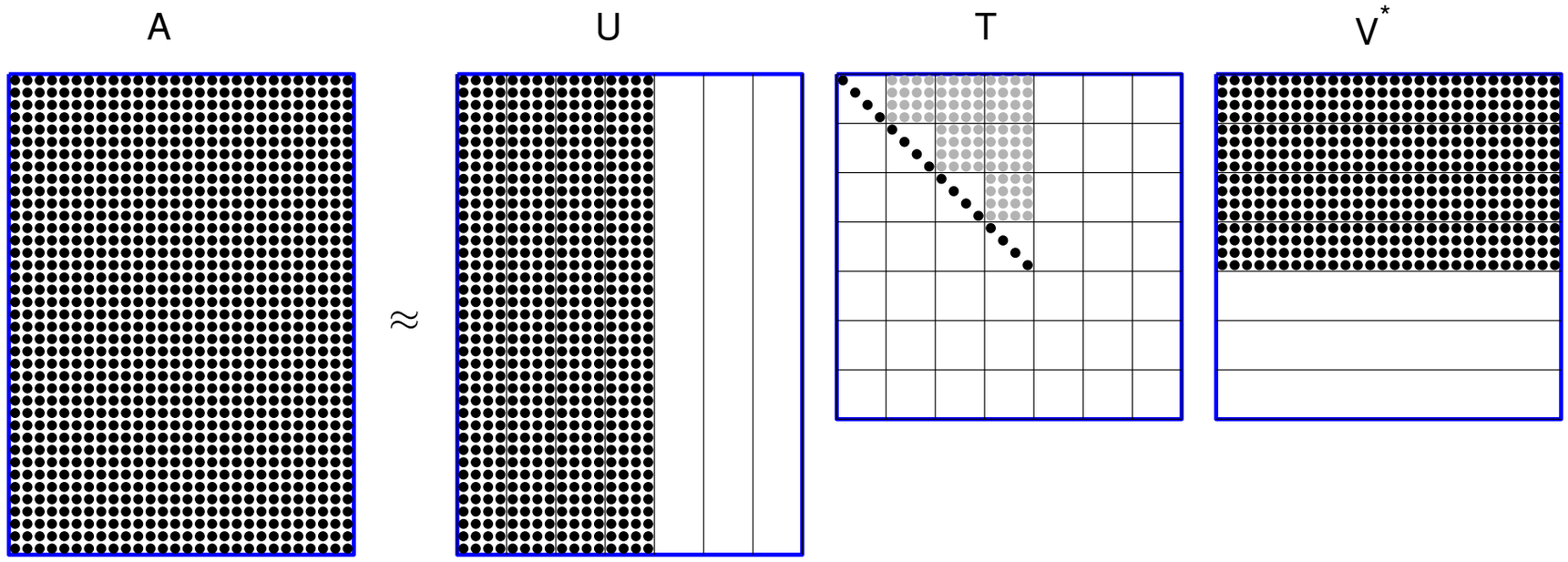
Recent work 2 (out of 5): Adaptive rank determination

Suppose that you seek to use the RSVD to compute a low rank approximation to a given matrix **A**, but you do not know its numerical rank in advance.

How do you proceed?

Answer (efficient in practice): Build the factorization iteratively.

Illustration for block size $b = 4$:



The rank-16 approximation.

Recent work 2 (out of 5): Adaptive rank determination

Numerical experiments illustrating how close the UTV is to the SVD

As a consequence of the fact that the super-diagonal elements of **T** are very small, the diagonal elements of **T** are *excellent* approximants to the singular values of **A**:

$$\mathbf{T}(j,j) \approx \sigma_j, \qquad j = 1,\, 2\, \ldots,\, \min(m,n).$$

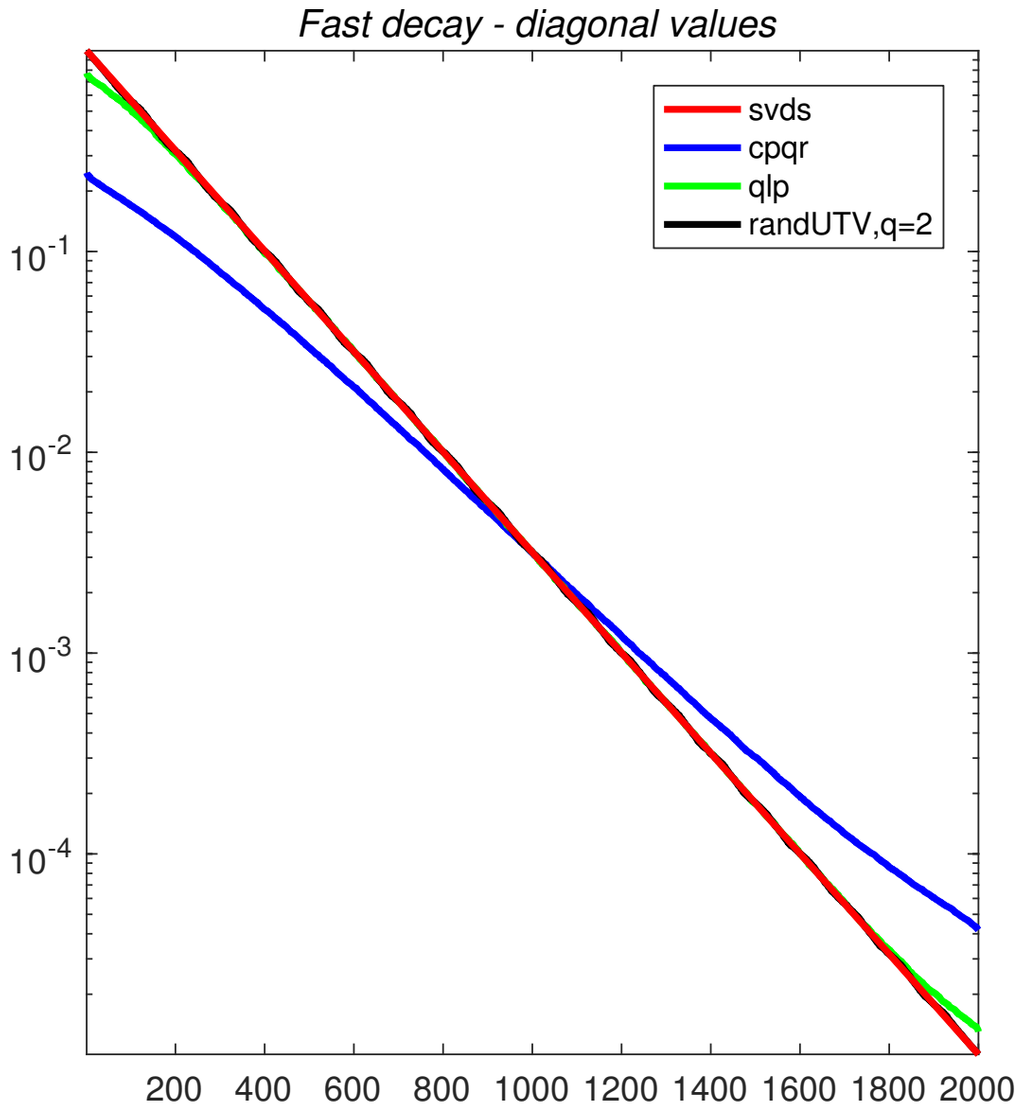
Recent work 2 (out of 5): Adaptive rank determination

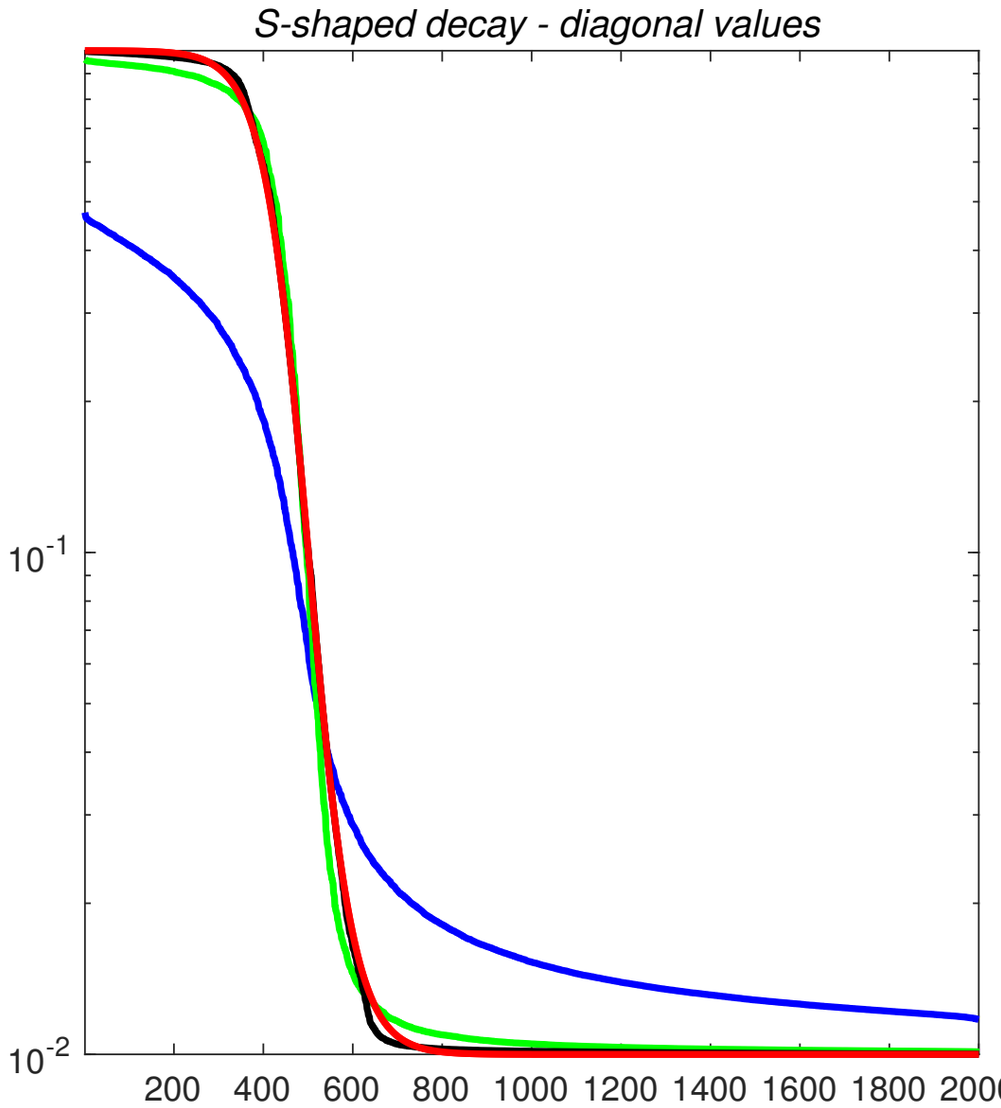
Numerical experiments illustrating how close the UTV is to the SVD

As a consequence of the fact that the super-diagonal elements of **T** are very small, the diagonal elements of **T** are *excellent* approximants to the singular values of **A**:

$$\mathbf{T}(j,j) \approx \sigma_j, \qquad j = 1,\, 2\, \ldots,\, \min(m,n).$$

Numerical illustration:





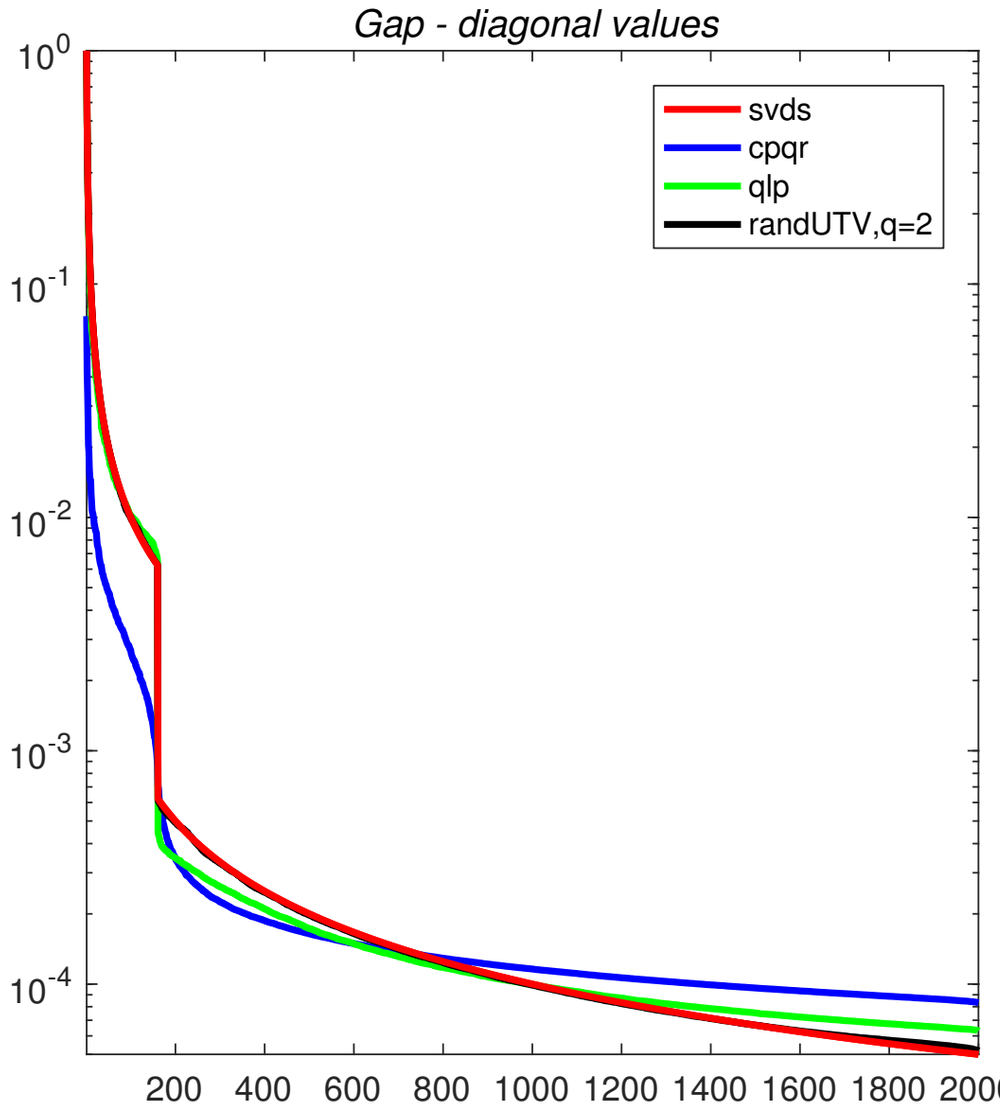
Recent work 2 (out of 5): Adaptive rank determination

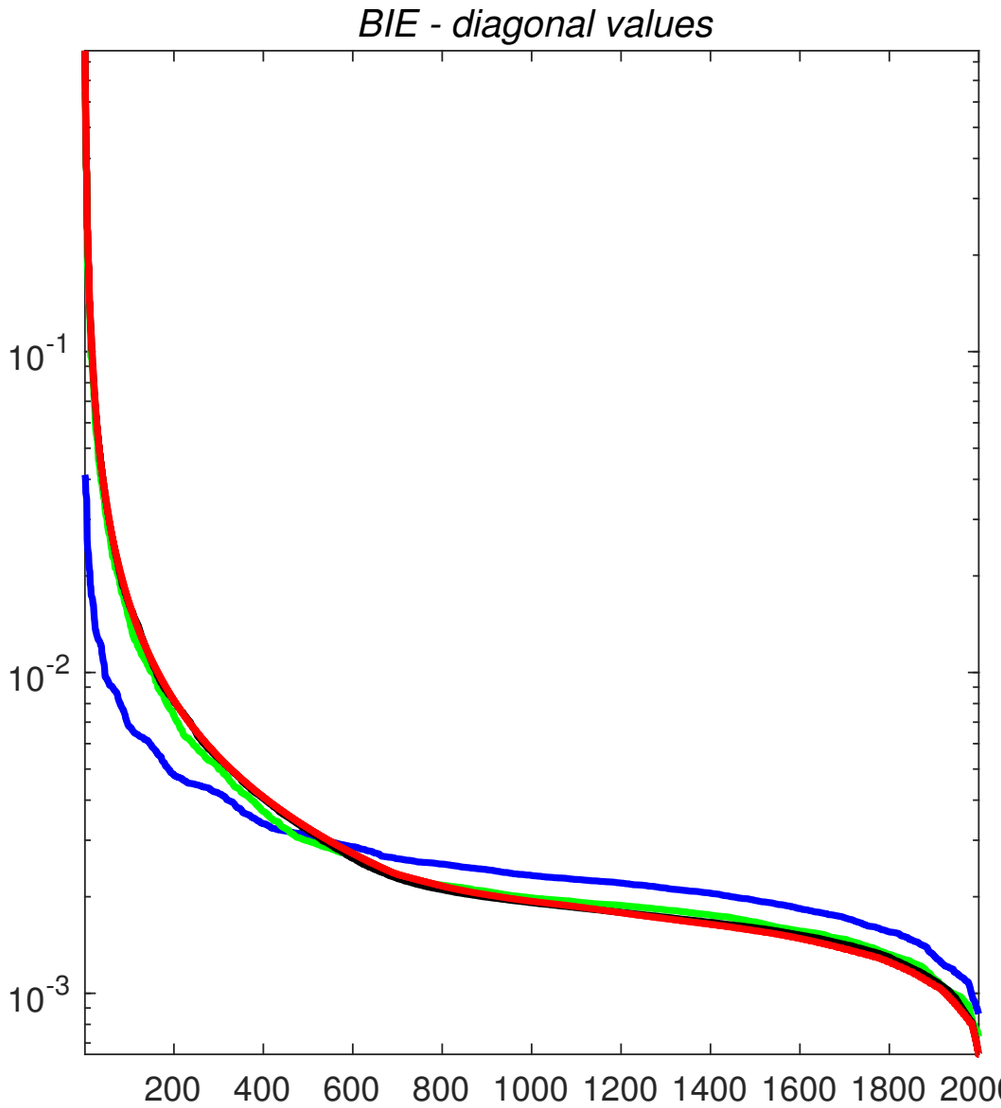
Numerical experiments illustrating how close the UTV is to the SVD

As a consequence of the fact that the super-diagonal elements of **T** are very small, the diagonal elements of **T** are *excellent* approximants to the singular values of **A**:

$$\mathbf{T}(j,j) \approx \sigma_j, \qquad j = 1, 2 \ldots, \min(m,n).$$

Numerical illustration:





Recent work 2 (out of 5): Adaptive rank determination

Numerical experiments illustrating how close the UTV is to the SVD

As a consequence of the fact that the super-diagonal elements of **T** are very small, the diagonal elements of **T** are *excellent* approximants to the singular values of **A**:

$$\mathbf{T}(j,j) \approx \sigma_j, \qquad j = 1,\, 2\, \ldots,\, \min(m,n).$$

Observation: You now have a basically perfect stopping criterion!

Note in particular that it is accurate with respect to *operator norm!* (Not just Frobenius.)

Recent work 3 (out of 5): Downdating of a small sketch

It turns out that you actually do not have to redraw a random sample from the matrix at every step — *you extract a single small sample, and then just recycle it!*

Recent work 3 (out of 5): Downdating of a small sketch

It turns out that you actually do not have to redraw a random sample from the matrix at every step — *you extract a single small sample, and then just recycle it!*

In randUTV, the first step is to draw a sample from $\mathbf{A}$:

$$\mathbf{Y}_1 = \mathbf{A}\Omega_1.$$

We use $\mathbf{Y}_1$ to build unitary $\mathbf{U}_1$ and $\mathbf{V}_1$ and form

$$\mathbf{A}^{(1)} := \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{0} & \mathbf{T}_{22} \end{bmatrix}.$$

We next want a sample from $\mathbf{A}^{(1)}$. Instead of generating a new draw Ω_2 from a Gaussian distribution and setting $\mathbf{Y}_2 = \mathbf{A}^{(1)}\Omega_2$, we proceed as follows:

$$\mathbf{Y}_2 := \mathbf{U}_1^* \mathbf{Y}_1 = \mathbf{U}_1^* \mathbf{A} \Omega_1 = \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 \mathbf{V}_1^* \Omega_1 = \mathbf{A}^{(1)} \Omega_2$$

where $\Omega_2 := \mathbf{V}_1^* \Omega_1$.

Question: Does Ω_2 have a Gaussian distribution?

Recent work 3 (out of 5): Downdating of a small sketch

It turns out that you actually do not have to redraw a random sample from the matrix at every step — *you extract a single small sample, and then just recycle it!*

In randUTV, the first step is to draw a sample from $\mathbf{A}$:

$$\mathbf{Y}_1 = \mathbf{A}\Omega_1.$$

We use $\mathbf{Y}_1$ to build unitary $\mathbf{U}_1$ and $\mathbf{V}_1$ and form

$$\mathbf{A}^{(1)} := \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{0} & \mathbf{T}_{22} \end{bmatrix}.$$

We next want a sample from $\mathbf{A}^{(1)}$. Instead of generating a new draw Ω_2 from a Gaussian distribution and setting $\mathbf{Y}_2 = \mathbf{A}^{(1)}\Omega_2$, we proceed as follows:

$$\mathbf{Y}_2 := \mathbf{U}_1^* \mathbf{Y}_1 = \mathbf{U}_1^* \mathbf{A} \Omega_1 = \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 \mathbf{V}_1^* \Omega_1 = \mathbf{A}^{(1)} \Omega_2$$

where $\Omega_2 := \mathbf{V}_1^* \Omega_1$.

Question: Does Ω_2 have a Gaussian distribution?

No. $\mathbf{V}_1$ is unitary, but it depends on the draw of Ω_1.
But in practice there is no discernible difference.

Recent work 4 (out of 5): Interpolatory and CUR decompositions.

Question: Suppose that $\mathbf{A}$ is an $m \times n$ matrix of rank k (exact, for now), and that you are given a matrix $\mathbf{Y}$ whose columns span the columns of $\mathbf{A}$. In other words,

$$\text{ran}(\mathbf{A}) \subseteq \text{ran}(\mathbf{Y}).$$

How do you efficiently use $\mathbf{Y}$ to build a rank-k approximation to $\mathbf{A}$?

Method 1: (Standard, we used this in construction of RSVD) Observe that

$$\mathbf{A} = \mathbf{Y}\mathbf{Y}^{\dagger}\mathbf{A} = \{\text{Do QR factorization } \mathbf{Y} = \mathbf{Q}\mathbf{R}\} = \mathbf{Q}\mathbf{Q}^{*}\mathbf{A} = \{\text{Set } \mathbf{B} := \mathbf{Q}^{*}\mathbf{A}\} = \mathbf{Q}\mathbf{B}.$$

Cost is $O(mk^2)$ to build $\mathbf{Q}$ and $O(mnk)$ to form $\mathbf{B}$.

Method 2: (Faster)

Recent work 4 (out of 5): Interpolatory and CUR decompositions.

Question: Suppose that $\mathbf{A}$ is an $m \times n$ matrix of rank k (exact, for now), and that you are given a matrix $\mathbf{Y}$ whose columns span the columns of $\mathbf{A}$. In other words,

$$\text{ran}(\mathbf{A}) \subseteq \text{ran}(\mathbf{Y}).$$

How do you efficiently use $\mathbf{Y}$ to build a rank-k approximation to $\mathbf{A}$?

Method 1: (Standard, we used this in construction of RSVD) Observe that

$$\mathbf{A} = \mathbf{Y}\mathbf{Y}^{\dagger}\mathbf{A} = \{\text{Do QR factorization } \mathbf{Y} = \mathbf{Q}\mathbf{R}\} = \mathbf{Q}\mathbf{Q}^{*}\mathbf{A} = \{\text{Set } \mathbf{B} := \mathbf{Q}^{*}\mathbf{A}\} = \mathbf{Q}\mathbf{B}.$$

Cost is $O(mk^2)$ to build $\mathbf{Q}$ and $O(mnk)$ to form $\mathbf{B}$.

Method 2: (Faster) Build an "interpolatory decomposition" of $\mathbf{Y}$:

$$\mathbf{Y} = \mathbf{X}\mathbf{Y}(I_{\text{S}}, :),$$

where I_{S} identifies a subset of the rows, and where $\mathbf{X}$ is well-conditioned. Then, *automatically*, we have

$$\mathbf{A} = \mathbf{X}\mathbf{A}(I_{\text{S}}, :).$$

Cost is $O(mk^2)$ to build $\mathbf{X}$ and $O(nk)$ to form $\mathbf{A}(I_{\text{S}}, :)$.

Building the ID can be done via Gram-Schmidt on the rows. (So CPQR on $\mathbf{Y}^{*}$.)

When the rank is approximate, then Method 2 can results in slightly larger errors.

Recent work 4 (out of 5): Interpolatory and CUR decompositions.

Note: Randomized ID is cheap. It does not need to revisit the entire matrix!

Randomized SVD:

1. Draw random matrix $\mathbf{\Omega}$ and form $\mathbf{Y} = \mathbf{A\Omega}$.
2. Form $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$.
3. Form $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.
4. Compute SVD of small matrix: $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd}(\mathbf{B})$.
5. Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

Then $\mathbf{A} \approx \mathbf{UDV}^*$.

Randomized ID:

1. Draw random matrix $\mathbf{\Omega}$ and form $\mathbf{Y} = \mathbf{A\Omega}$.
2. Form a row-ID of $\mathbf{Y}$ so that $\mathbf{Y} = \mathbf{XY}(I, :)$. *Use, e.g., CPQR. (Or LUPP!)*
3. Set $\mathbf{R} = \mathbf{A}(I, :)$.

Then *automatically* $\mathbf{A} \approx \mathbf{XR}$.

The only "postprocessing" is extracting the columns in $\mathbf{R}$.

Recent work 4 (out of 5): Interpolatory and CUR decompositions.

Note: Randomized ID is cheap. It does not need to revisit the entire matrix!

Randomized SVD:

1. Draw random matrix $\mathbf{\Omega}$ and form $\mathbf{Y} = \mathbf{A\Omega}$.
2. Form $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$.
3. Form $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.
4. Compute SVD of small matrix: $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd}(\mathbf{B})$.
5. Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

Then $\mathbf{A} \approx \mathbf{UDV}^*$.

Randomized CUR:

1. Draw random matrix $\mathbf{\Omega}$ and form $\mathbf{Y} = \mathbf{A\Omega}$.
2. Form a row-ID of $\mathbf{Y}$ so that $\mathbf{Y} = \mathbf{XY}(I, :)$. *Use, e.g., CPQR. (Or LUPP!)*
3. Set $\mathbf{R} = \mathbf{A}(I, :)$.
4. Form a column-ID of $\mathbf{R}$ so that $\mathbf{R} = \mathbf{R}(:, J)\mathbf{Z}$. *Use, e.g., CPQR. (Or LUPP!)*
5. Set $\mathbf{U} = (\mathbf{A}(I, J))^{\dagger}_{\epsilon}$. *Some care required...*

Then $\mathbf{A} \approx \mathbf{CUR}$.

The only "postprocessing" is extracting the submatrices $\mathbf{C}$ *and* $\mathbf{R}$.

Recent work 5 (out of 5): Iterative CUR

Finally, we describe the algorithm `IterativeCUR` that given a tolerance ε incrementally builds a CUR decomposition

$$\mathbf{A} \approx \mathbf{CUR}$$

such that $\|\mathbf{A} - \mathbf{CUR}\|_{\mathrm{F}} \leq \varepsilon$.

The method has the following characteristics:

- The method is incremental, and "blocked" for computational efficiency.
- After each step of the iteration is completed, an a posteriori error estimation is used to provide a highly reliable stopping criterion.
- The "new" indices that are added at each step of the iteration are chosen using a sketch of the present residual $\mathbf{A} - \mathbf{CUR}$. LUPP is used $\rightarrow$ high speed.
- Downdating of sketch is used to ensure that only $O(1)$ random samples from the matrix are required. (Say $b = 10$ or $b = 25$ samples.)
- After the initial sketch, *we never again visit the entire matrix*; instead, we only extract the chosen columns and rows.
- Overall complexity is $O(mn + (m+n)k^2)$ where k denotes the computed rank.

With N. Pritchard, T. Park, Y. Nakatsukasa, arxiv:2509.21963

Recent work 5 (out of 5): Iterative CUR

Generate a sketch matrix $\mathbf{\Omega} \in \mathbb{R}^{1.1b \times n}$.

$\mathbf{I} = [\,]$; $\mathbf{I} = [\,]$; $\mathbf{C} = [\,]$; $\mathbf{U} = [\,]$; $\mathbf{R} = [\,]$;

$\mathbf{Y}_0^{\mathrm{row}} = \mathbf{\Omega A}$.

$\rho_0 = \|\mathbf{Y}_0^{\mathrm{row}}\|_{\mathrm{F}} / \|\mathbf{A}\|_{\mathrm{F}}$.

for $k = 1, 2, 3, \ldots$

if $[\rho_{k-1} \leq \varepsilon]$ **then stop**

Compute column indices J_k using the sketch $\mathbf{Y}_k^{\mathrm{row}}$. *[LUPP recommended.]*

$\mathbf{Y}_k^{\mathrm{col}} = \mathbf{A}(:, J_k) - \mathbf{CUR}(:, J_k)$.

Compute row indices I_k using the sketch $\mathbf{Y}_k^{\mathrm{col}}$. *[CPQR recommended.]*

$J = [J, J_k]$ and $I = [I, I_k]$.

$\mathbf{C} = [\mathbf{C}, \mathbf{A}(:, J_k)]$ and $\mathbf{R} = [\mathbf{R};\ \mathbf{A}(I_k, :)]$.

Compute $\mathbf{U}_k = \mathbf{A}(I, J)^{\dagger}$. *[Cheap Woodbury like update formula exists.]*

Compute $\mathbf{Y}_k^{\mathrm{row}} = \mathbf{\Omega A} - \mathbf{\Omega C}_k \mathbf{U}_k \mathbf{R}_k$. *[Downdating!]*

$\rho_k = \|\mathbf{Y}_k^{\mathrm{row}}\|_{\mathrm{F}} / \|\mathbf{A}\|_{\mathrm{F}}$.

end for

Recent work 5 (out of 5): Iterative CUR

The stopping criterion is accurate and reliable in practice.

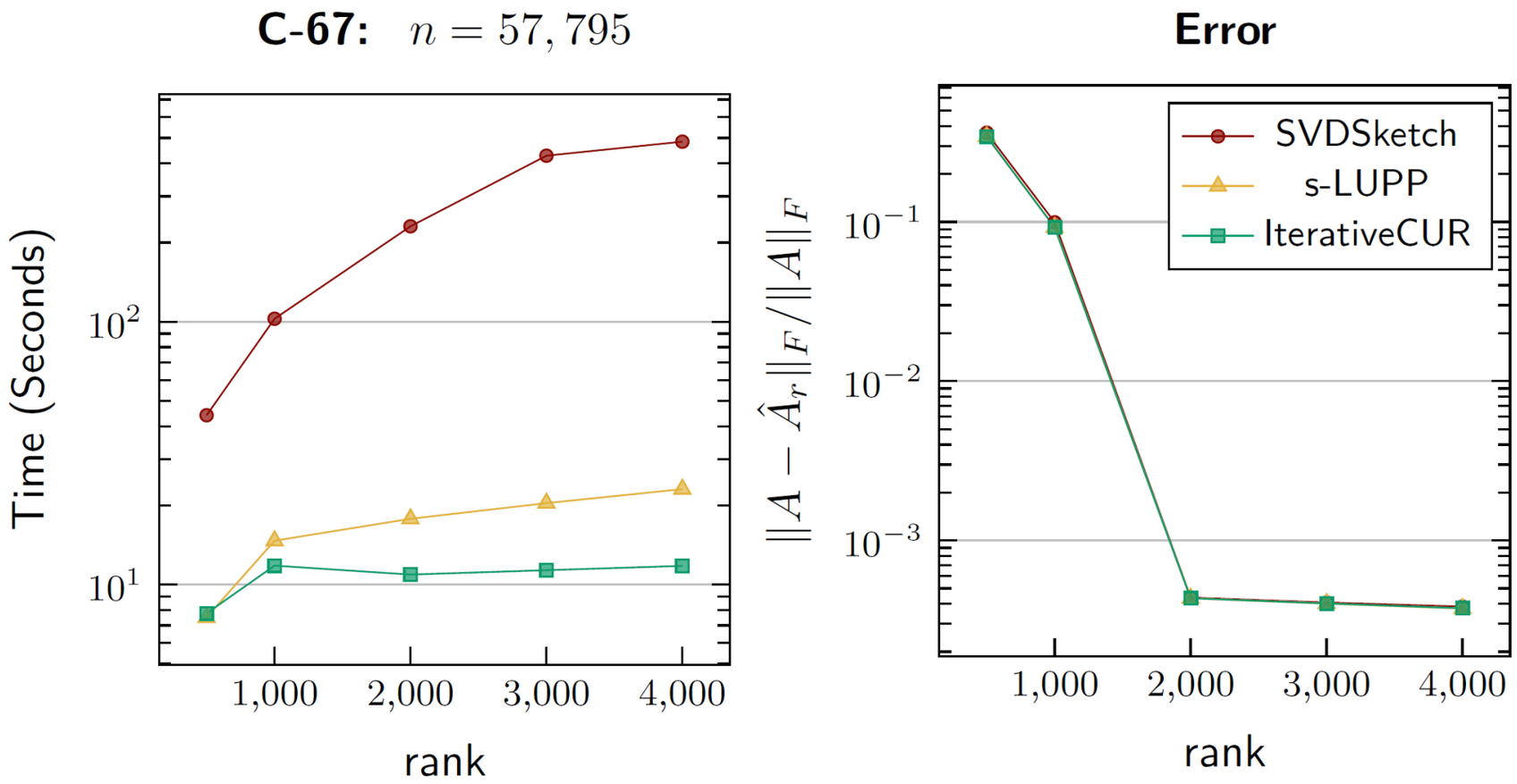
Due to the downdating, it does not rest on solid theoretical guarantees.

However, the final output can be "certified" using a small additional sketch.

Method:	RSVD	randUTV	Accelerated RSVD	IterativeCUR
Arithmetic work:	$\sim mnk$	$\sim mnk$	$\sim mn + k^2(m+n)$	$\sim mn + k^2(m+n)$
Rank adaptive:	No	Yes	No	Yes
Precise theory:	Yes	Yes	So-so	?
Certificate of accuracy:	Yes	Excellent!!	Yes	Yes

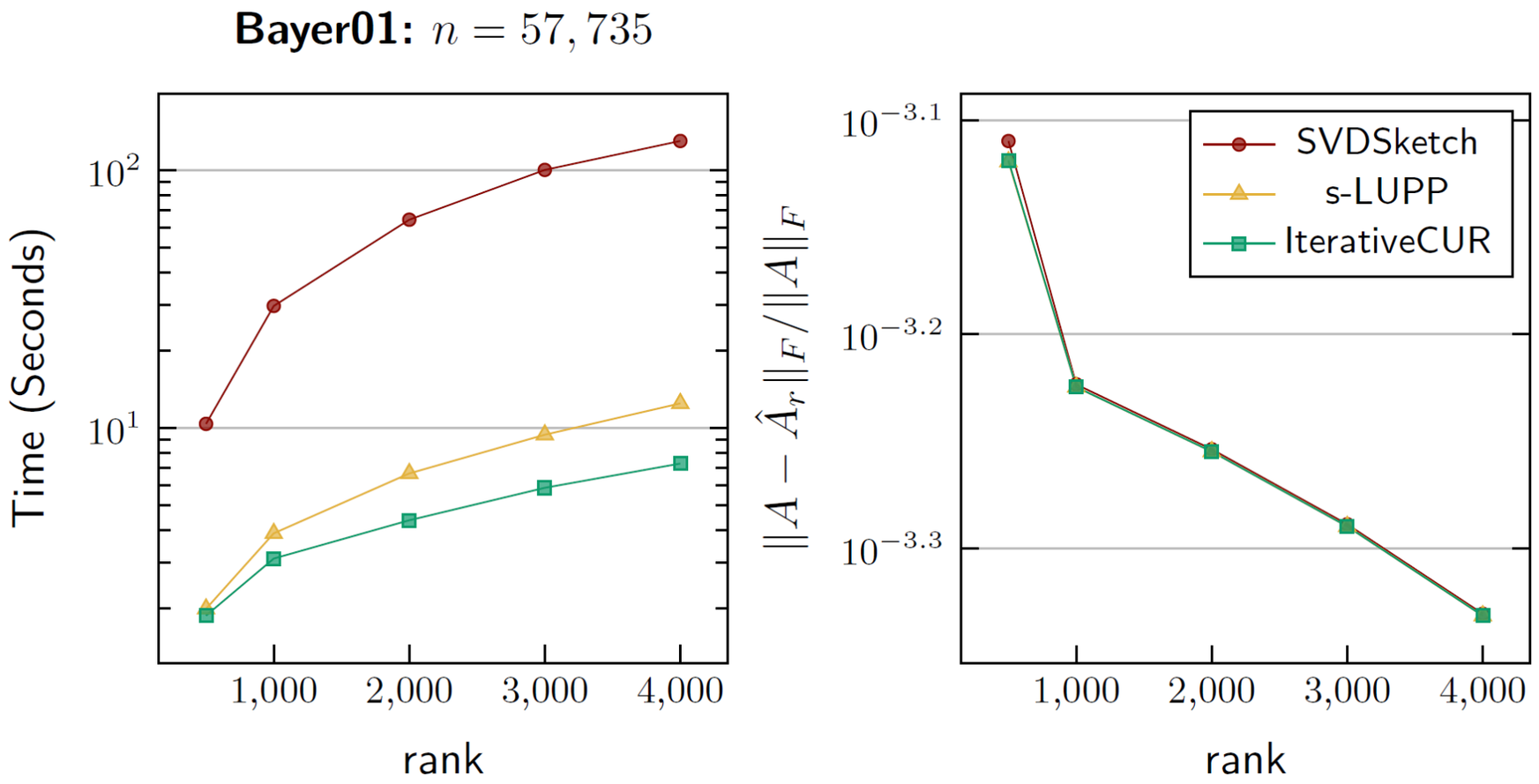
With N. Pritchard, T. Park, Y. Nakatsukasa, arxiv:2509.21963

Recent work 5 (out of 5): Iterative CUR



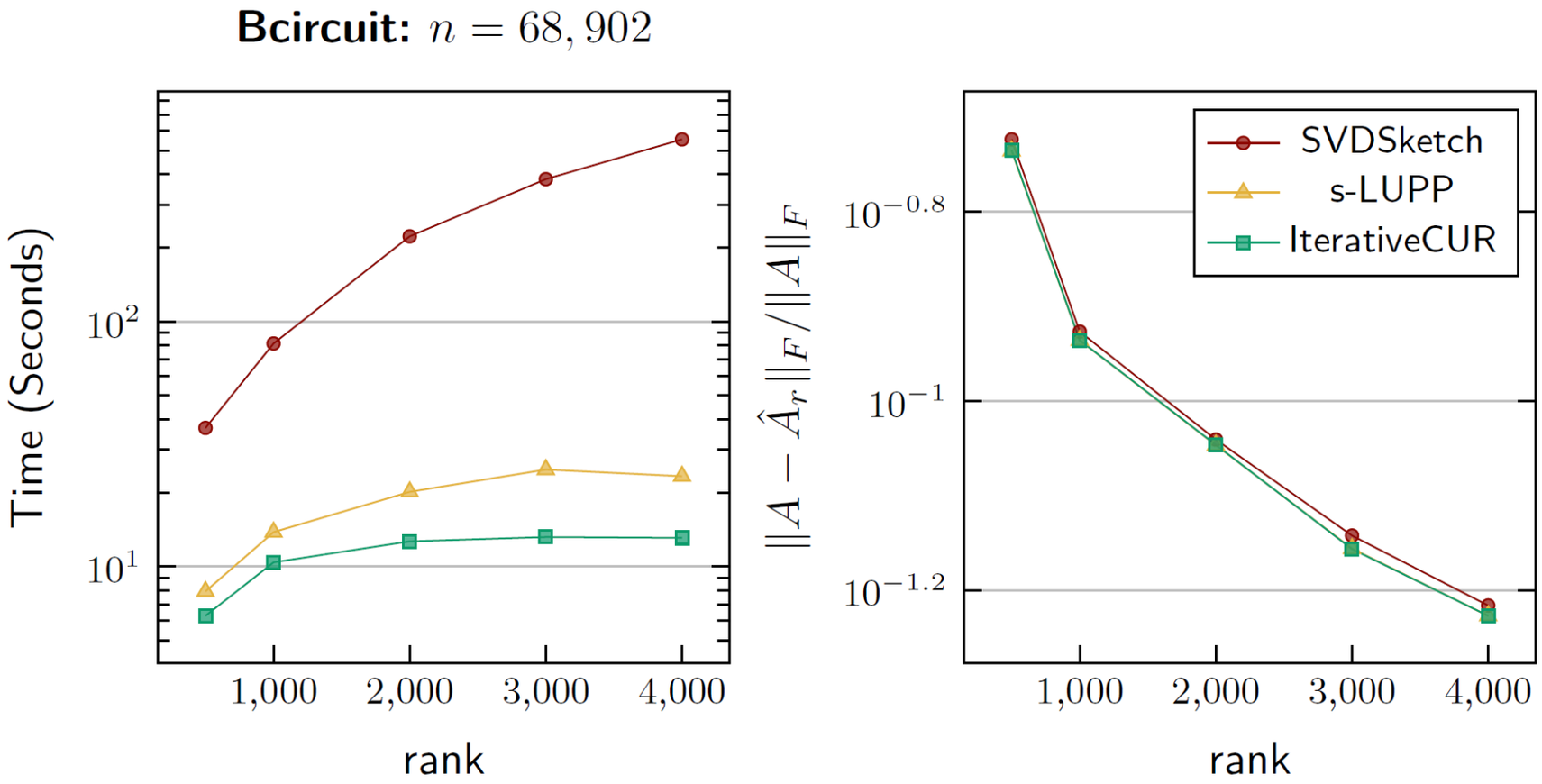
- SVDsketch: MATLAB's built-in [Yu-Yu-Li 18]
  - Roughly 'iterative RSVD' drawing new sketch each time
  - Cannot handle accuracy $\leq \epsilon_{\text{mach}}^{1/2} \approx 10^{-8}$
- s-LUPP: big sketch+LUPP
  - to test effect of reusing small sketch *G*
- IterativeCUR-LUPP: this work

Recent work 5 (out of 5): Iterative CUR



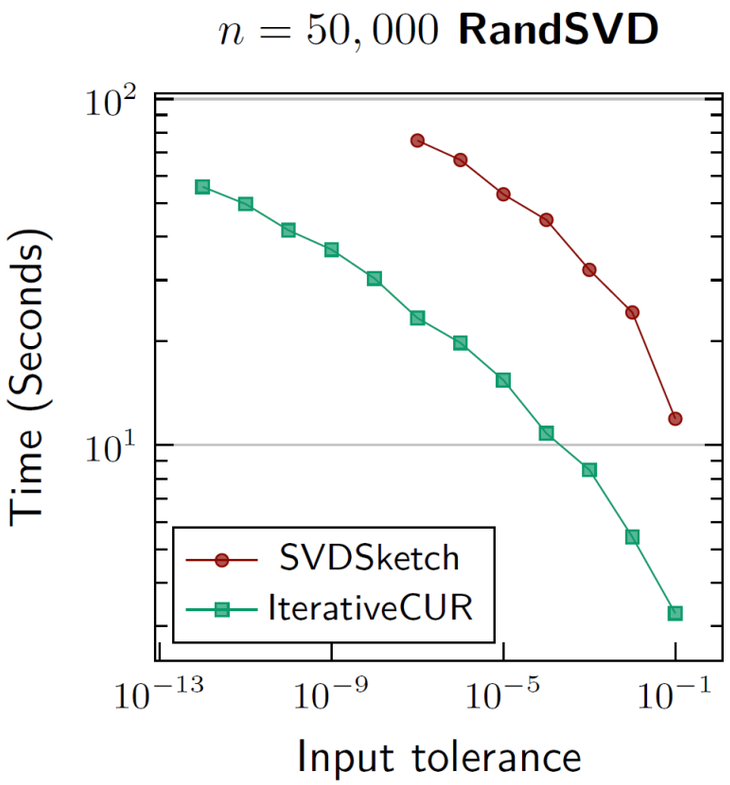
- SVDsketch: MATLAB's built-in [Yu-Yu-Li 18]
  - Roughly 'iterative RSVD' drawing new sketch each time
  - Cannot handle accuracy $\leq \epsilon_{\text{mach}}^{1/2} \approx 10^{-8}$
- s-LUPP: big sketch+LUPP
  - to test effect of reusing small sketch *G*
- IterativeCUR-LUPP: this work

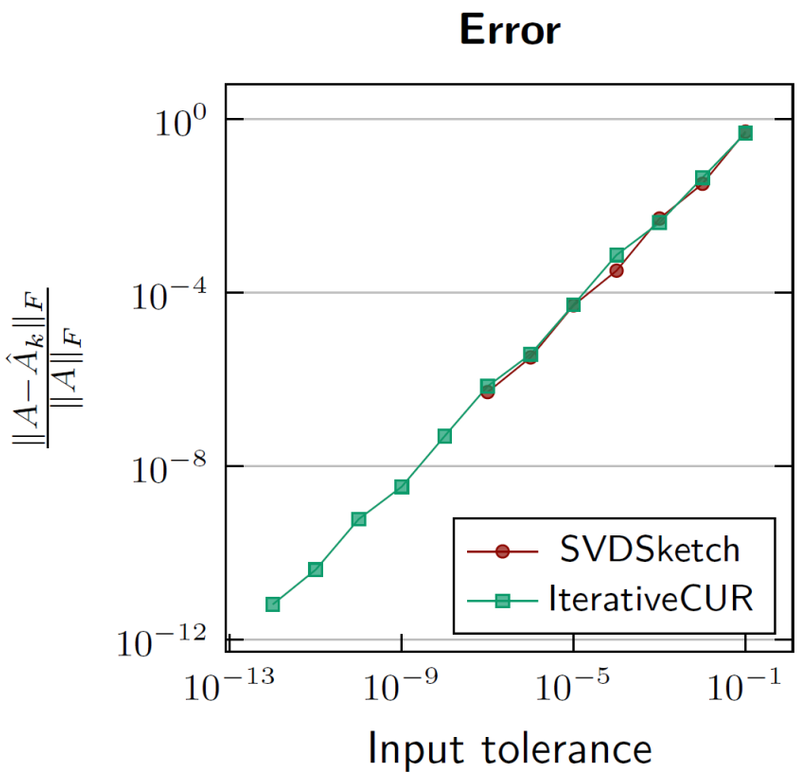
Recent work 5 (out of 5): Iterative CUR

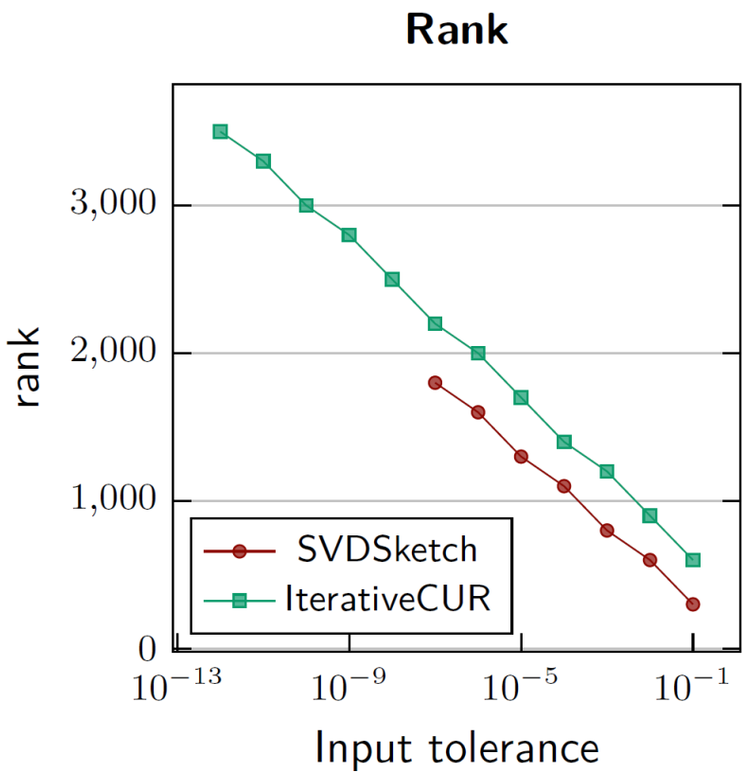


- SVDsketch: MATLAB's built-in [Yu-Yu-Li 18]
  - Roughly 'iterative RSVD' drawing new sketch each time
  - Cannot handle accuracy $\leq \epsilon_{\text{mach}}^{1/2} \approx 10^{-8}$
- s-LUPP: big sketch+LUPP
  - to test effect of reusing small sketch *G*
- IterativeCUR-LUPP: this work

Recent work 5 (out of 5): Iterative CUR







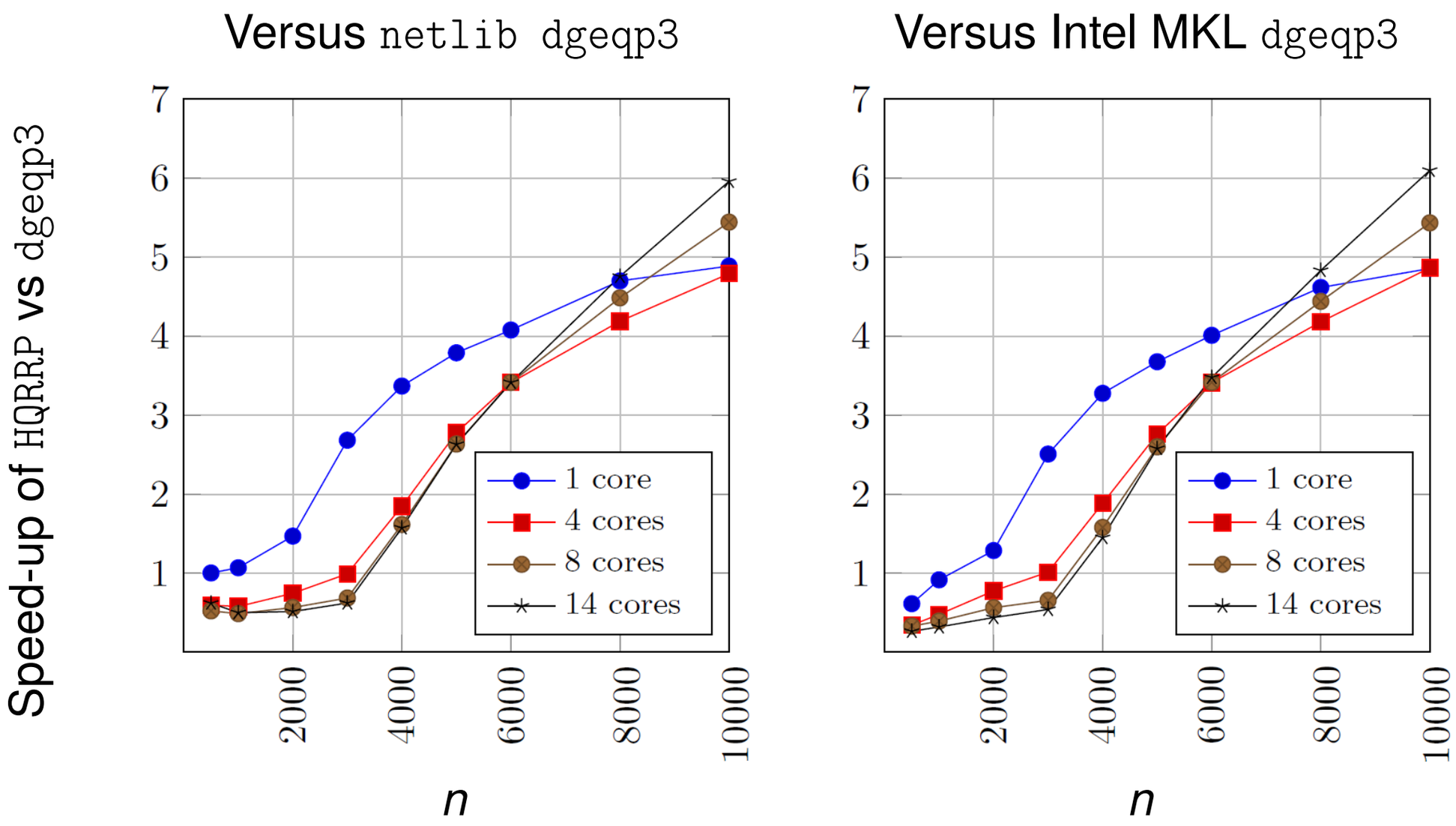
- SVDSketch fails for tolerance $\leq 10^{-8}$.
- Observe speedup attained by Iterative CUR.

RNLA in standard software: some examples

Environment	Documented routine	Notes
MATLAB	`svdsketch`	Built-in function: computes the SVD of a low-rank matrix sketch. Introduced in R2020b.
scikit-learn	`randomized_svd`	Library routine: computes an approximate truncated SVD using randomization.
scikit-learn PCA	`PCA(svd_solver="randomized")`	Documented PCA solver path using randomized truncated SVD; selected automatically for some large problems with few requested components.
scikit-learn TruncatedSVD	`TruncatedSVD(algorithm="randomized")`	Randomized SVD is the default algorithm; standard sparse/LSA-style dimensionality-reduction path.
Dask Array	`dask.array.linalg.svd_compressed`	Distributed-array routine for a compressed approximate thin SVD; used when arrays are chunked in both dimensions.

Takeaway: RSVD/sketching is exposed as documented, user-facing functionality in mainstream scientific-computing software.

FLAME implementation of *full* CPQR: The randomized pivot selection technique can also be used to compute a column pivoted QR decomposition (CPQR). Resolves long standing issue of how to move flops from BLAS2 to BLAS3 in CPQR.



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/

Outline

Basic algorithmic pattern for low rank approximation:

- Randomized SVD. *[Brief review]*

Enhancements to the basic pattern:

- Fast a posteriori error estimation. *[Newish]*
- Adaptive rank determinination.
- Downdating of a small sketch.
- CUR and interpolatory decompositions.
- Putting all the enhancements together: Iterative CUR. *[Newish]*

Extension to kernel matrices and continuum problems. *[New]*

- Suppose $\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{y}_j)$ for some kernel k and point sets $\{\boldsymbol{x}_i\}_{i=1}^m$ and $\{\boldsymbol{y}_j\}_{j=1}^n$. How do you compute a low rank approximation to $\mathbf{A}$ without forming the entire matrix?

Extension to kernel matrices and continuum problems

Objective: Build a rank-adaptive CUR approximation to a kernel matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$:

$$\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{y}_j), \qquad \{\boldsymbol{x}_i\}_{i=1}^{m} \in \Omega_{\mathrm{t}}, \quad \{\boldsymbol{y}_j\}_{j=1}^{n} \in \Omega_{\mathrm{s}}.$$

Kernel matrices arise when pairwise interactions are mediated by a continuum kernel k.

- **PDE solvers and integral equations:** boundary element methods, Lippmann–Schwinger equations, Schur complements, and fast direct solvers.
- **Hierarchical matrices:** $\mathcal{H}$-, $\mathcal{H}^2$-, HSS/HBS/HODLR formats require repeated compression of off-diagonal kernel blocks.
- **Kernel methods in data analysis:** Gaussian processes, kernel ridge regression, radial basis function interpolation, and covariance models.
- **Many-query settings:** compression must often be reliable, adaptive, and cheap enough to apply block by block.

Common feature: Separated source and target sets often give low numerical rank,

$$r \ll \min(m,n),$$

but forming all mn entries is too expensive.

Often **A** *is an off-diagonal block of a larger matrix.*

Extension to kernel matrices and continuum problems

Objective: Build a rank-adaptive CUR approximation to a kernel matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

$$\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{y}_j), \qquad \{\boldsymbol{x}_i\}_{i=1}^{m} \in \Omega_{\mathrm{t}}, \quad \{\boldsymbol{y}_j\}_{j=1}^{n} \in \Omega_{\mathrm{s}}.$$

Access model:

- We can evaluate individual entries $k(\boldsymbol{x}_i, \boldsymbol{y}_j)$.
- We cannot afford to form all of $\mathbf{A}$.
- We do *not* assume matrix-vector products with $\mathbf{A}$.

Adaptive cross approximation (ACA) – the "standard" existing tool

- Builds a cross approximation, and hence a CUR-type approximation, by repeatedly selecting pivot rows and columns from the current residual. ACA can be viewed as an incomplete Gaussian-elimination procedure, with pivoting heuristics in the spirit of partial or rook style pivot searches.
- Uses only selected matrix entries; often attains close to $O(r(m+n))$ kernel evaluations in practice. And $O(r^2(m+n))$ arithmetic work.
- Popular in hierarchical matrix codes and boundary element computations.
- The pivoting and stopping rules are *heuristic*. ACA is often very fast, but it can fail on adversarial or poorly scaled examples, and the stopping criterion is not reliable in the way that a sketch-based certificate is.

Standard ACA: one row and one column at a time

Let

$$\mathbf{S}_k = \mathbf{A} - \sum_{t=1}^{k} \mathbf{c}_t \mathbf{r}_t$$

be the current residual. A typical partially pivoted ACA step is:

1. Start from a pivot row i_k.
2. Evaluate the residual row $\mathbf{S}_k(i_k, :)$ and choose
$$j_k = \arg\max_j |\mathbf{S}_k(i_k, j)|.$$
3. Evaluate the residual column $\mathbf{S}_k(:, j_k)$ and set
$$\mathbf{r}_{k+1} = \mathbf{S}_k(i_k, :), \qquad \mathbf{c}_{k+1} = \frac{\mathbf{S}_k(:, j_k)}{\mathbf{S}_k(i_k, j_k)}.$$
4. Update
$$\mathbf{S}_{k+1} = \mathbf{S}_k - \mathbf{c}_{k+1}\mathbf{r}_{k+1}.$$
5. Choose the next pivot row from the residual column just evaluated.
$$i_{k+1} = \arg\max_i |\mathbf{S}_k(i, j_k)|,$$
with minor safeguards against reusing rows or tiny pivots.

Cost: One residual row and one residual column per step, hence roughly $O(r(m+n))$ entry evaluations in favorable cases.

Caveat: The pivot row and stopping rule are heuristic; the full residual is never seen.

Exploiting analytic knowledge

To avoid forming the full matrix, we have to exploit a priori analytic knowledge about

BOTH the kernel AND the geometry.

To quantify things, let us introduce a set $\mathcal{W}$ of all "possible" functions on Ω_{t}:

$$f \in \mathcal{W} \qquad \Leftrightarrow \qquad f(x) = k(x, y), \qquad \text{for some } y \in \Omega_{\mathrm{s}}.$$

We assume that the set $\mathcal{W}$ can be well-approximated by some R-dimensional space V, in the sense that any $f \in \mathcal{W}$ can be approximated to relative precision ε by some $v \in V$. For instance, we may assume that

$$\forall\, f \in \mathcal{W}: \qquad \inf_{v \in V} \frac{\|v - f\|}{\|f\|} \leq \varepsilon$$

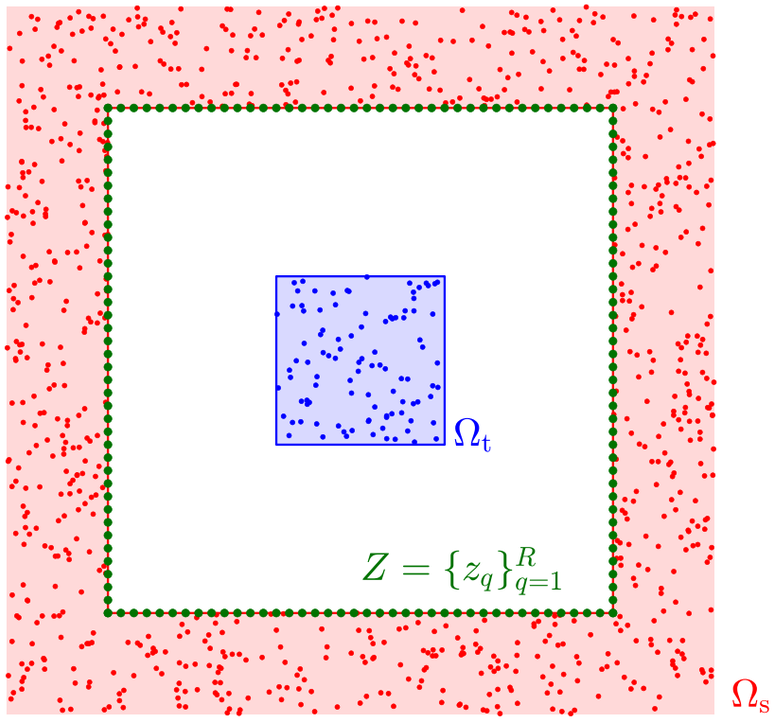
where $\|\cdot\|$ is some suitable norm on Ω_{t}. Say max norm.

We will describe two methods for how to exploit this rank deficiency:

- ***Proxy points:*** A well established "surrogate sources" method.
- ***Interpolation points:*** A "surrogate targets" method that is less explored. This is our focus today, as it interacts particularly well with IterativeCUR.

Proxy point method

Introduce artificial "proxy source locations" $Z = \{z_q\}_{q=1}^R$:



For kernels from differential and integral equation contexts, you can often find proxy points such that any $v \in V$ can be generated by equivalent sources on the proxy points.

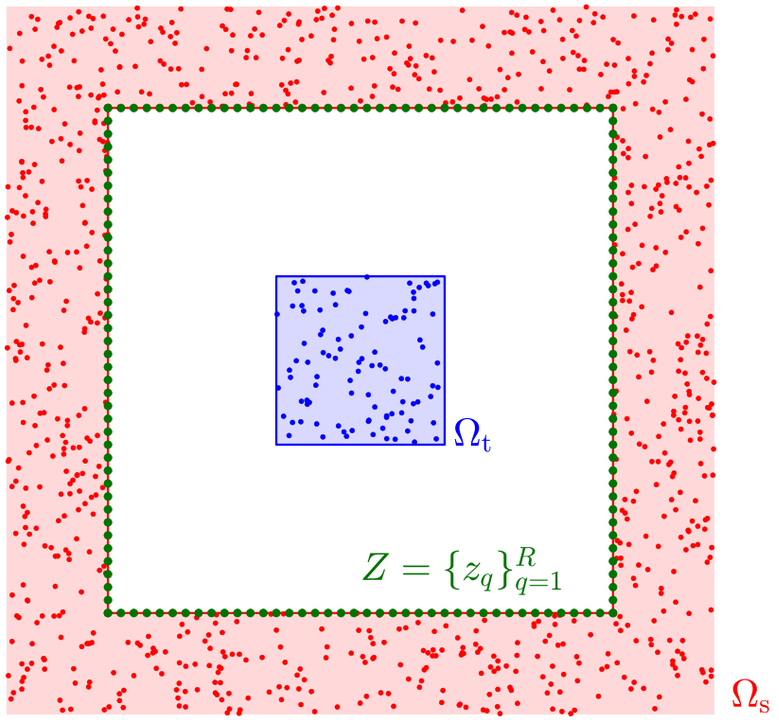
Mathematically justified through integral representations. E.g., if $k(\cdot, y)$ is analytic, then a Cauchy representation would give (with Γ the square that holds the proxy points Z)

$$k(x,y) = \frac{1}{2\pi i} \int_\Gamma \frac{k(z,y)}{z-x}\, dz, \qquad y \in \Omega_s, \qquad x \in \Omega_t.$$

If k is the fundamental solution to an elliptic PDE (Laplace, Helmholtz, time-harmonic Maxwell, etc), then "Green representation formulas" do the equivalent work.

Proxy point method

Introduce artificial "proxy source locations" $Z = \{z_q\}_{q=1}^{R}$:



For kernels from differential and integral equation contexts, you can often find proxy points such that any $v \in V$ can be generated by equivalent sources on the proxy points.

Form the surrogate matrix $\hat{\mathbf{A}} = k(X, Z) \in \mathbb{R}^{m \times R}$. Then col($\mathbf{A}$) $\subseteq$ col($\hat{\mathbf{A}}$).

You can form a basis for the column space of $\mathbf{A}$ by processing the small surrogate $\hat{\mathbf{A}}$.

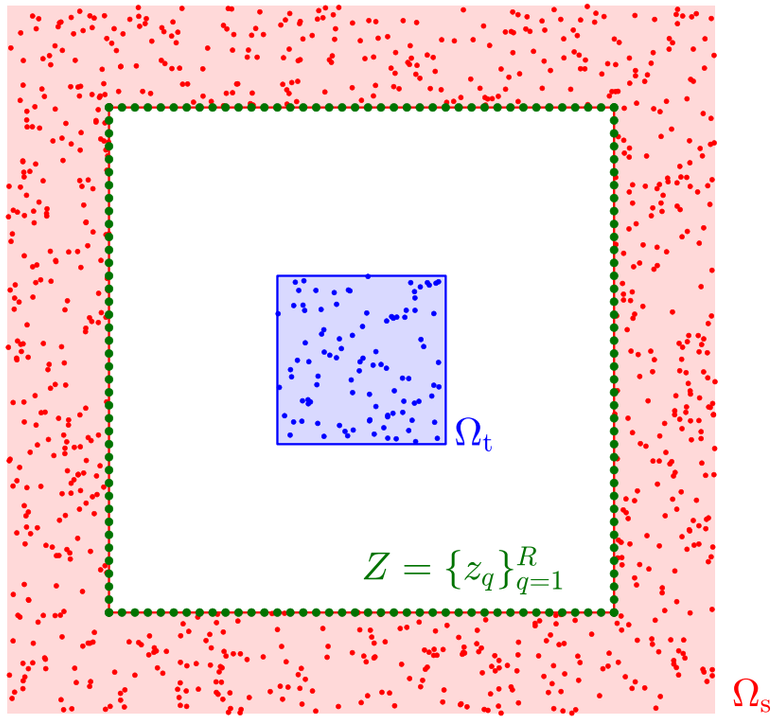
Better, we can form an ID of the surrogate: $\hat{\mathbf{A}} = \mathbf{X}\hat{\mathbf{A}}(I, :)$.
Then automatically, $\mathbf{A} = \mathbf{X}\mathbf{A}(I, :)$. Then detect the true rank of $\mathbf{A}$ by performing a rank-revealing factorization of $\mathbf{A}(I, :)$.

Proxy point method

When a kernel function admits a Green's function representation, the proxy point method is rock solid reliable.

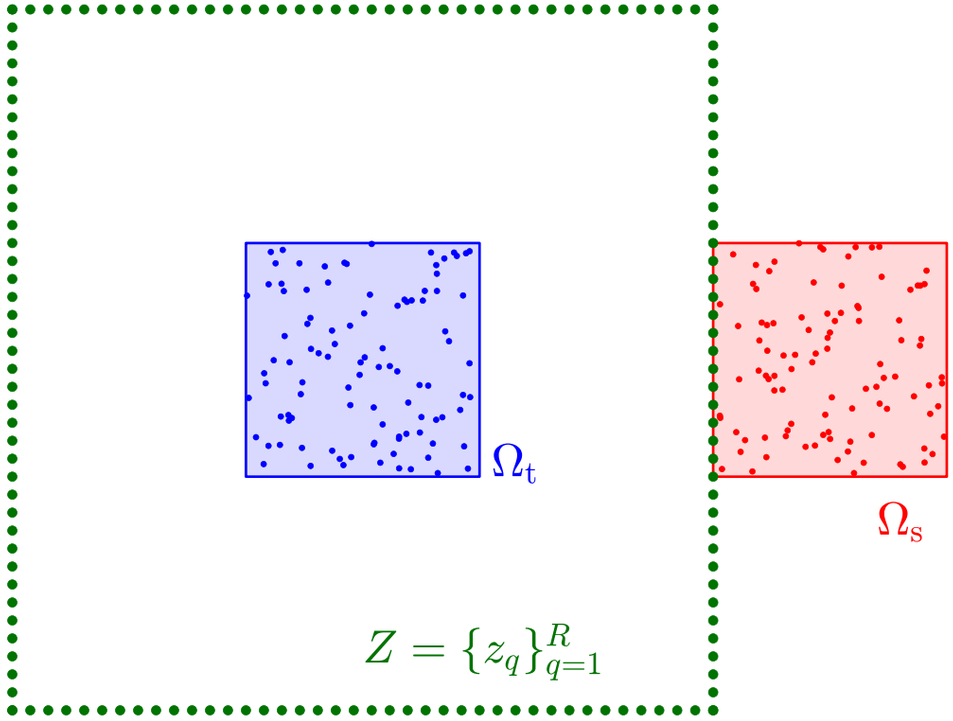
However, it can be more or less optimal as a computational tool.

Proxy sources work well



$r \approx R$

Proxy sources suboptimal



$r \ll R$

(Recall, r is the true rank, while R is the number of proxy points.)

To be precise, the proxy point method cannot take advantage when the source points are clustered, or otherwise have exploitable geometry.

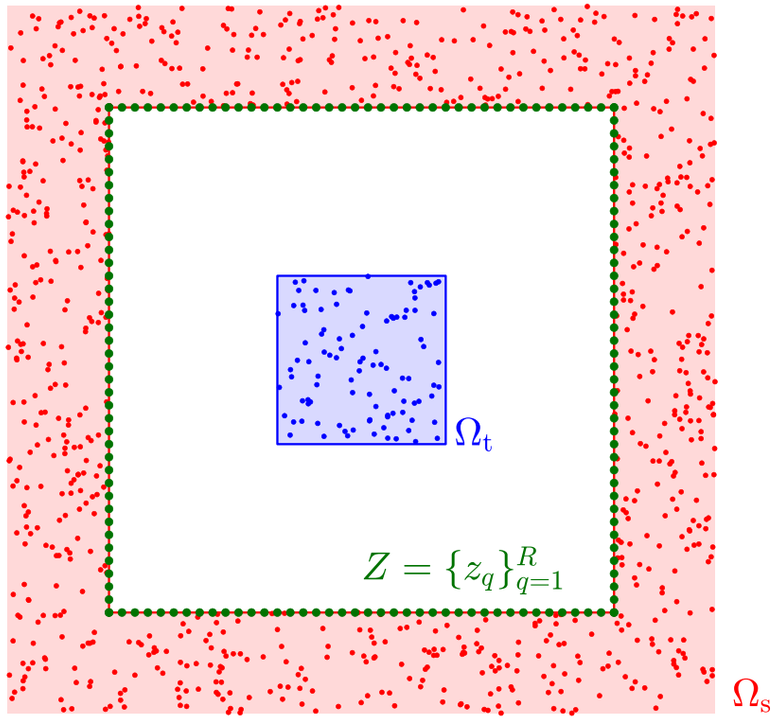
(Well, it *can*, but that requires constructing bespoke proxy point distributions . . .)

Proxy point method

When a kernel function admits a Green's function representation, the proxy point method is rock solid reliable.

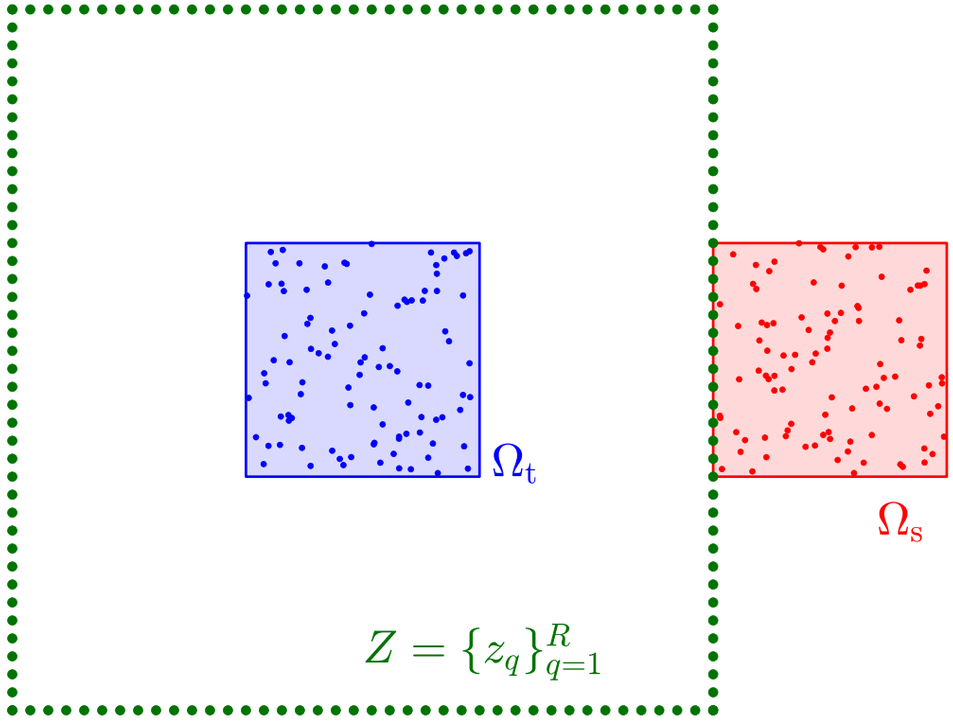
However, it can be more or less optimal as a computational tool.

Proxy sources work well



$r \approx R$

Proxy sources suboptimal



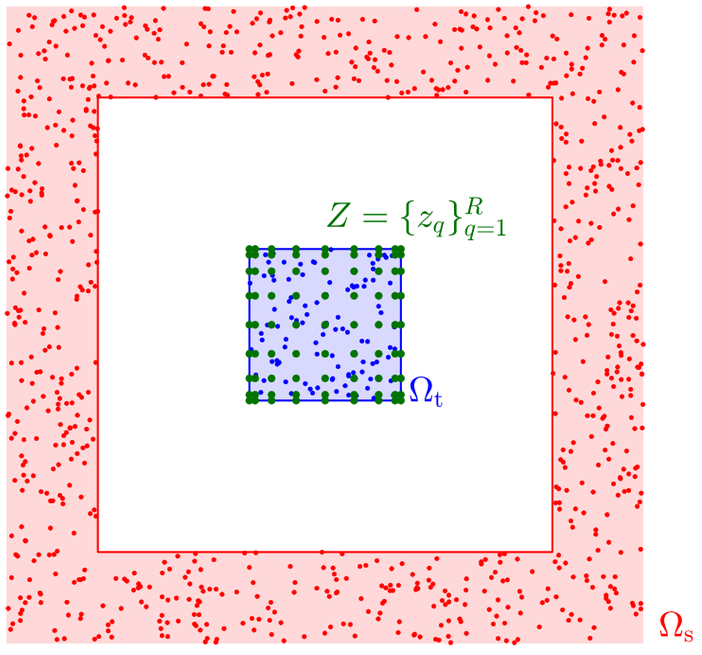
$r \ll R$

(Recall, r is the true rank, while R is the number of proxy points.)

Personal opinion (contested!): In the case of $\mathcal{H}^2$ and uniform $\mathcal{H}$-matrices, this is a much better and safer method than ACA.

Interpolation point method

Introduce interpolatory target locations $Z = \{z_q\}_{q=1}^{R}$:



The idea is that any function $v \in V$ can be stably interpolated from its values at the points in Z. In particular, there is a map $\mathbf{F}_{X,Z}$ such that $v(X) = \mathbf{F}_{X,Z} v(Z)$ for any $v \in V$, with $\|\mathbf{F}_{X,Z} v(Z)\| \leq C\|v(Z)\|$ for some suitable norms.

Form the surrogate matrix $\hat{\mathbf{A}} = k(Z, Y) \in \mathbb{R}^{R \times n}$. Then row($\mathbf{A}$) $\subseteq$ row($\hat{\mathbf{A}}$).

You can form a basis for the row space of $\mathbf{A}$ by processing the small surrogate $\hat{\mathbf{A}}$.

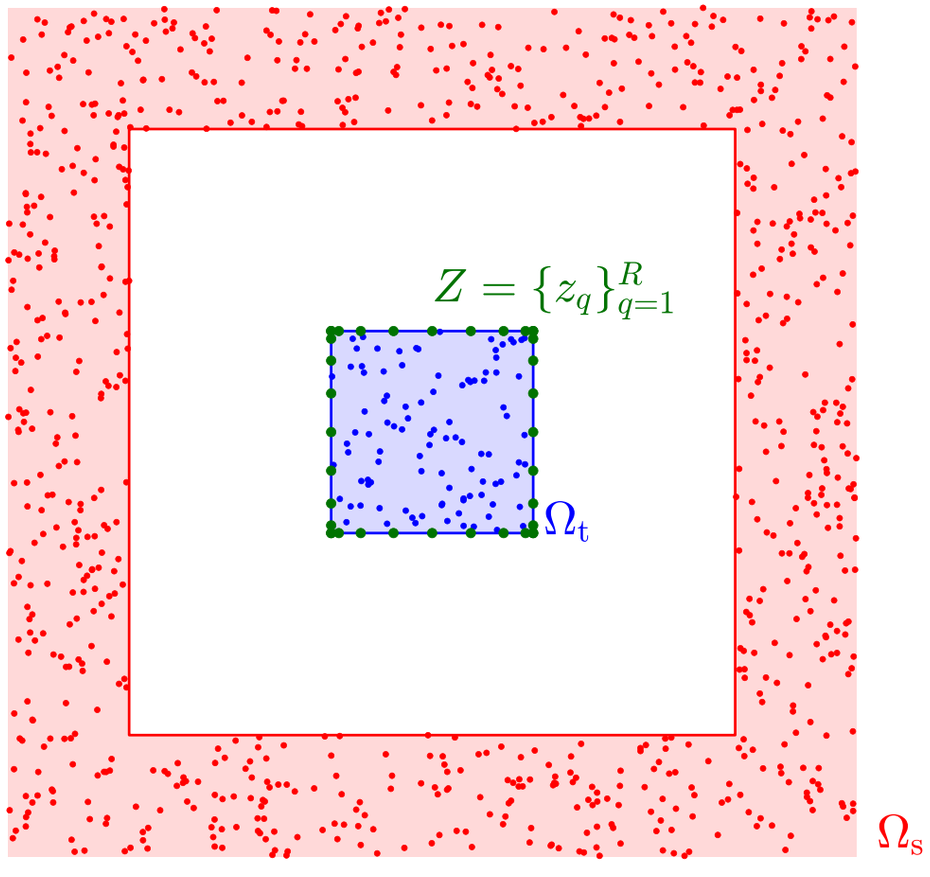
Better, we can form an ID of the surrogate: $\hat{\mathbf{A}} = \hat{\mathbf{A}}(:, J)\mathbf{Y}$.
Then automatically, $\mathbf{A} = \mathbf{A}(:, J)\mathbf{Y}$.
Then detect the true rank of $\mathbf{A}$ by performing a rank-revealing factorization of $\mathbf{A}(:, J)$.

Interpolation point method

Examples of how to build interpolation points in different environments:

- *Analytic in 2D:* Sample on an enclosing circle; Fourier representation.
- *Harmonic in 3D:* Sample on an enclosing sphere; spherical harmonics.
- *Bandlimited model:* Uniform grid; FFT/NUFFT style reconstruction.
- *Local smooth model:* Composite Legendre/Chebyshev grids; fast local interpolation.
- *Helmholtz:* Boundary traces, two-layer stabilization, or phase/directional models.

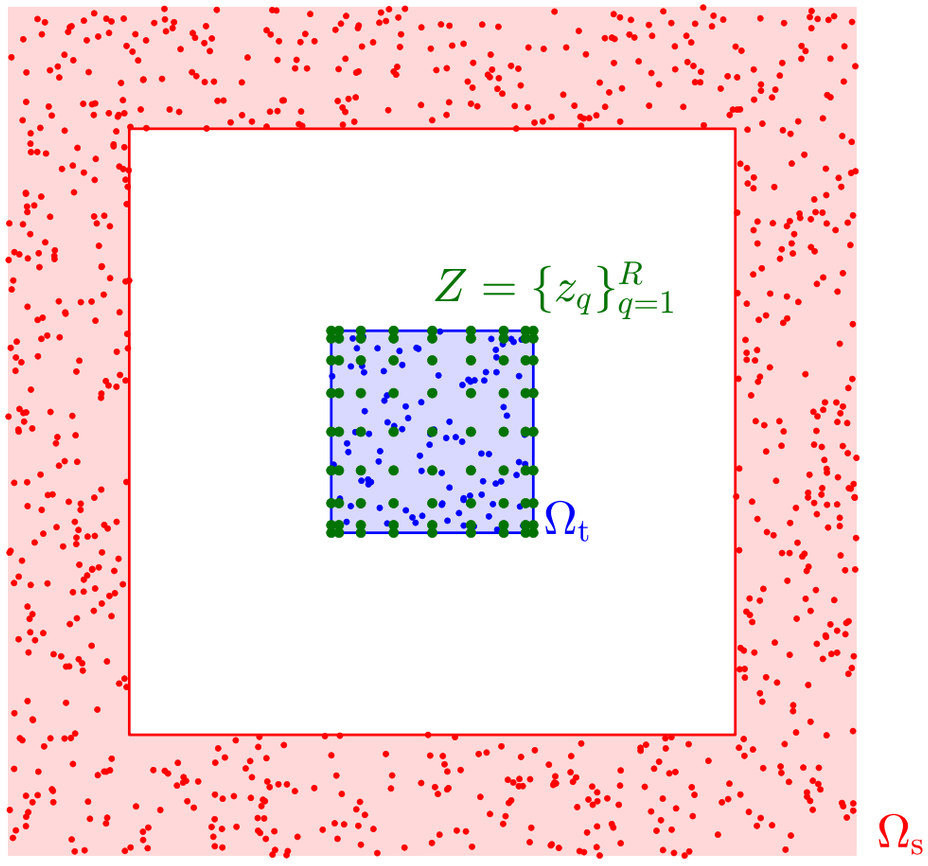
Note: For PDE models, we often pick the interpolation points *on the boundary.*



Interpolation point method

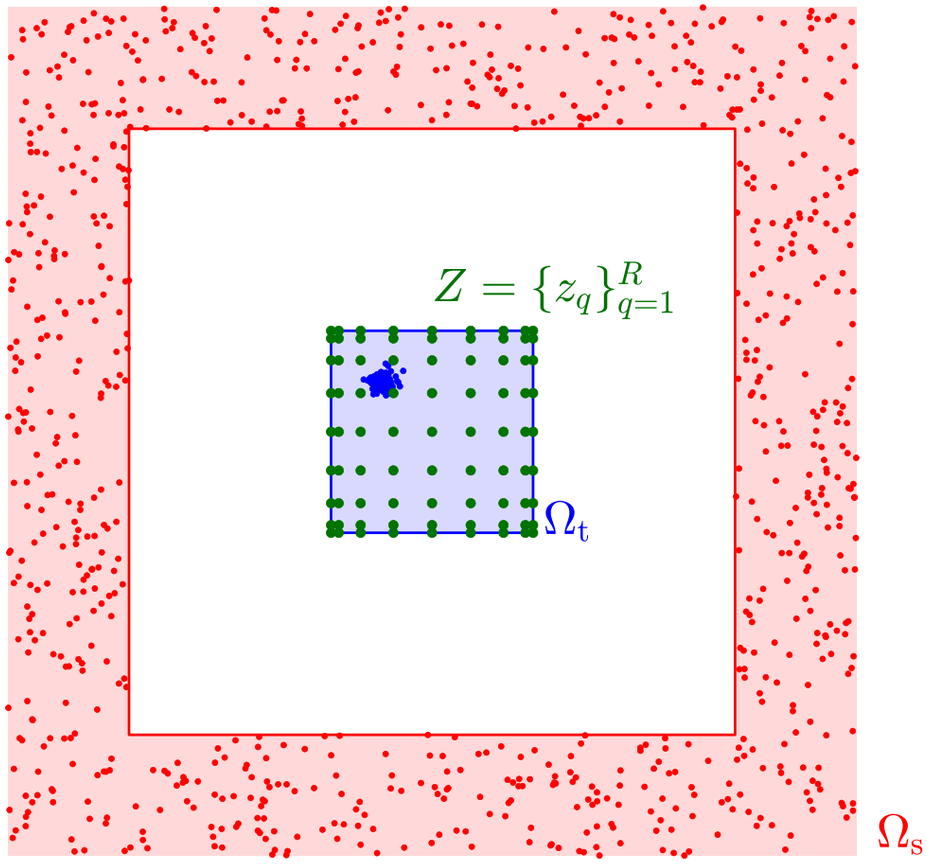
Like the proxy point method, the interpolation point method can be more or less optimal.

Interpolation points work well



$r \approx R$

Interpolation points suboptimal



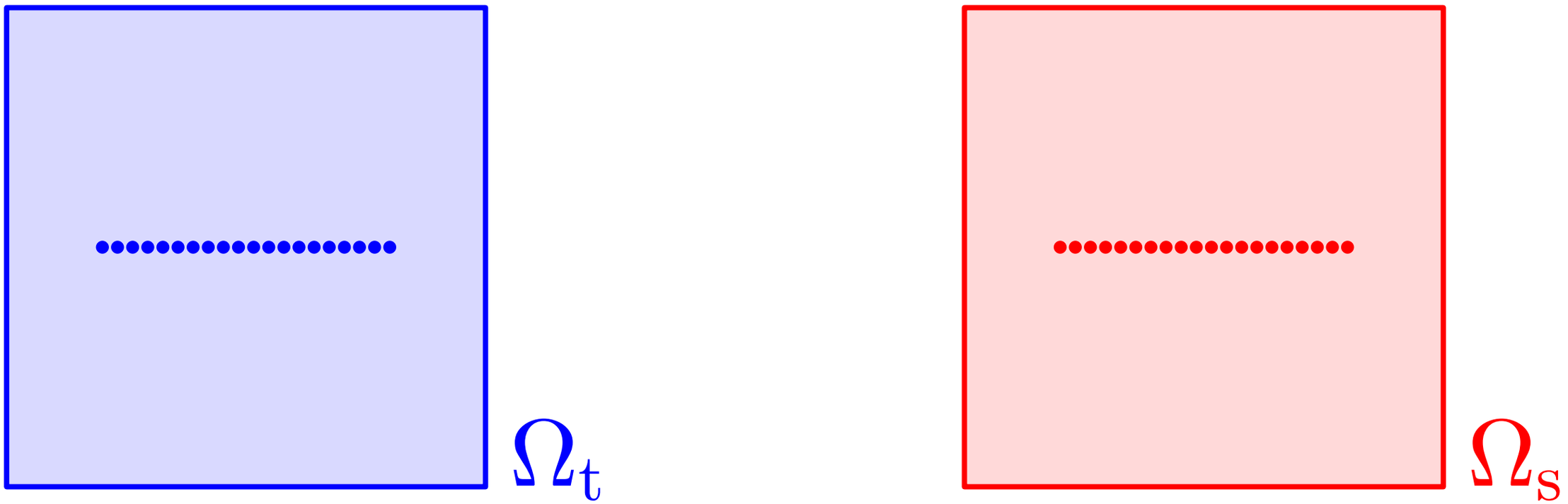
$r \ll R$

(Recall, r is the true rank, while R is the number of interpolation points.)

To be precise, the interpolation point method initially over-estimate the rank when the *target* points are clustered, or otherwise have exploitable geometry.

There are situations where *both* methods initially over-estimate the rank

Consider a high frequency Helmholtz problem on the following geometry:



Here,

$$k(x, y) = H_0^{(1)}(\kappa|x - y|)$$

for a large wave-number κ. (Complex valued, but this is no problem.)

In order to correctly detect that this interaction is very low rank, you need to know about BOTH the source AND the target geometry (and the exact nature of the kernel).

Both methods would initially over-estimate the rank in this case. That is, $r \ll R$.
This is where IterativeCUR comes in handy!

Comparison of costs

Let s denote the rank of the surrogate matrix $\hat{\mathbf{A}}$. Observe that $s \leq R$.

	Proxy points	Interpolation points	ACA (well ...)
Kernel evaluations:	$Rm + sn + rm$	$Rn + sm + rn$	$r(m+n)$
Flops:	$O(mR + (m+R)s^2 + nr^2)$	$O(nR + (n+R)s^2 + mr^2)$	$O(r^2(m+n))$

The costs of the proxy point and the interpolation point methods represent modest improvements upon what was previously reported in the literature.

These gains were achieved by applying IterativeCUR to factorize first the surrogate matrix, and then the identified submatrix of $\mathbf{A}$.

The interpolation points method has two distinct advantages:

(1) It works for a broader class of kernels than the proxy point method.

(2) *It often allows fast application of* $\mathbf{A}$ *itself* $\rightarrow$ *IterativeCUR applies!*

Interpolation point method

Important: The interpolation method comes with an important additional benefit in that we can often build a well conditioned interpolation matrix $\mathbf{F}_{X,Z}$ such that for any $v \in V$, we have $v(X) = \mathbf{F}_{X,Z} v(Z)$. In particular

$$\mathbf{A} = k(X, Y) = \mathbf{F}_{X,Z} k(Z, Y) = \mathbf{F}_{X,Z} \hat{\mathbf{A}}.$$

Deterministic method: Do a rank revealing factorization of $\mathbf{F}_{X,Z}$ so that $\mathbf{F}_{X,Z} = \mathbf{EF}$ for a well-conditioned $\mathbf{E}$. Then $\mathbf{A} = \mathbf{EF}\hat{\mathbf{A}}$, and we can determine the true rank of $\mathbf{A}$ by factorizing $\mathbf{F}\hat{\mathbf{A}}$.

This works particularly well if you do an interpolatory decomposition, since then $\mathbf{F}$ is just subsampling, and you do not even need to form the full surrogate matrix!

Randomized method: Draw a thin Gaussian random matrix $\mathbf{G} \in \mathbb{R}^{n \times \ell}$. Then form

$$\mathbf{S} = \mathbf{AG} = \mathbf{F}_{X,Z} \hat{\mathbf{A}} \mathbf{G}.$$

In practice, form $\mathbf{S}$ by computing $\hat{\mathbf{A}}$ and then evaluating $\mathbf{S}' = \hat{\mathbf{A}}\mathbf{G}$. Then use fast interpolation to interpolate the ℓ vectors in $\mathbf{S}'$ to the full column samples in $\mathbf{S}$.

At this point, you can just execute the “row dual” of IterativeCUR, using the column samples in $\mathbf{S}$ to guide your row pivot selections, and to provide a stopping criterion.

Interpolation point method – now randomized with IterativeCUR

The cost of the randomized method works out as:

$$\begin{aligned} \text{Kernel evaluations:} &\quad Rn + r(m+n) \\ \text{Floating point work:} &\quad O\big(Rn\ell + T_X(\ell) + (m+n)r^2 + m\ell r + r^3\big) \\ \text{Memory:} &\quad O\big(m\ell + r(m+n) + r^2\big) \end{aligned}$$

Here

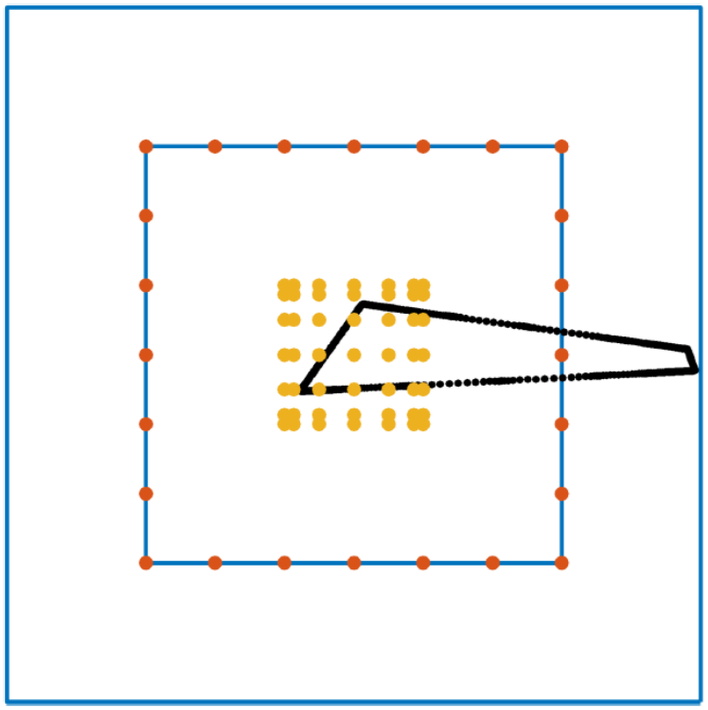
$$\ell = b + p$$

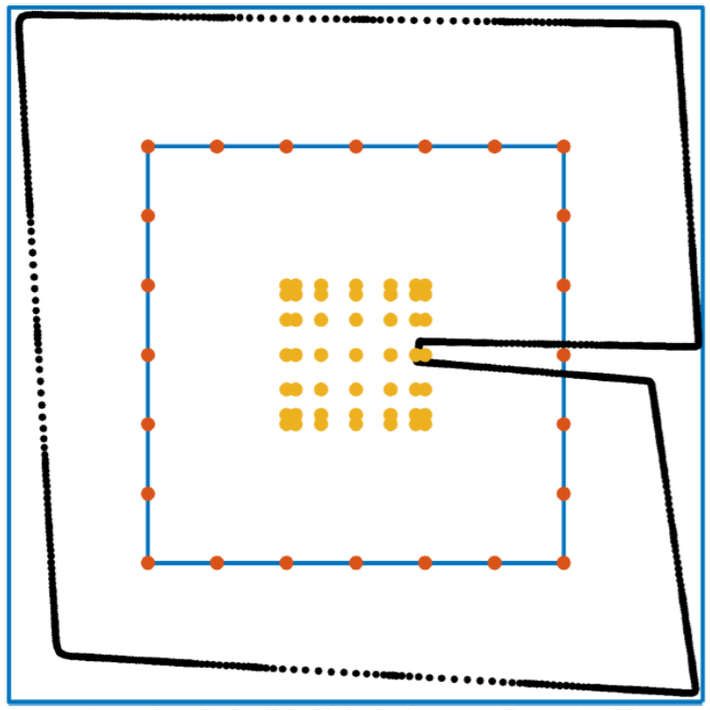
where b is the block size and p is a small oversampling parameter.

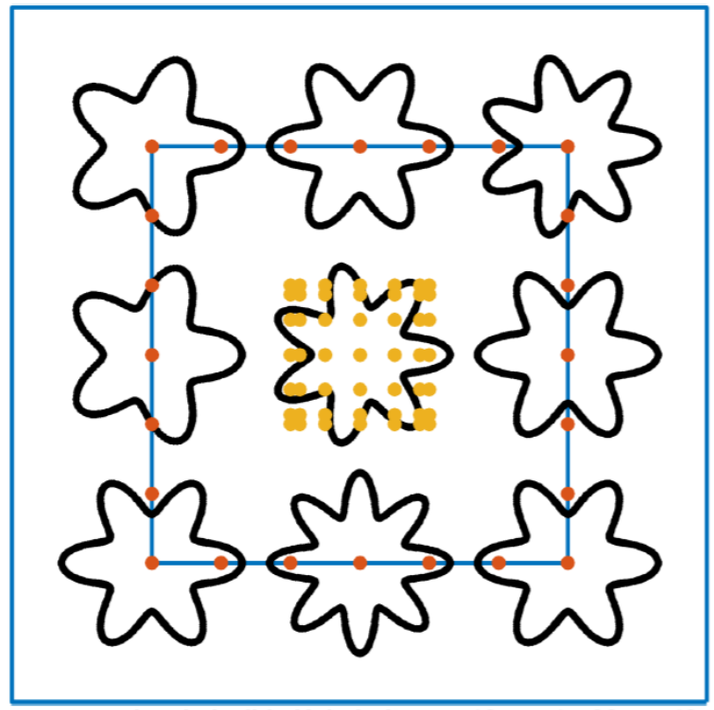
Important point: ℓ is small and fixed; it is independent of r and R.

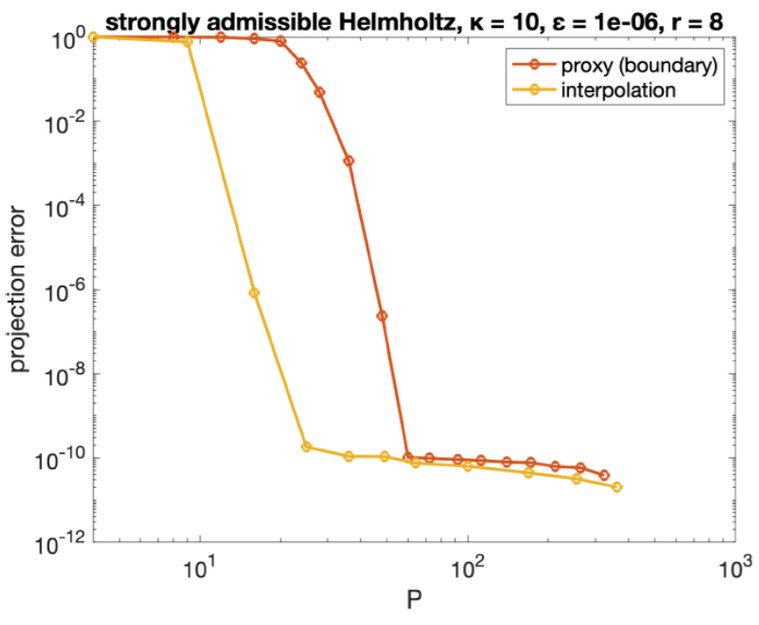
- R is the analytic upper bound from the surrogate model.
- r is the numerical rank discovered by the adaptive CUR iteration.
- The method is attractive when $r \ll R \ll \min(m, n)$.

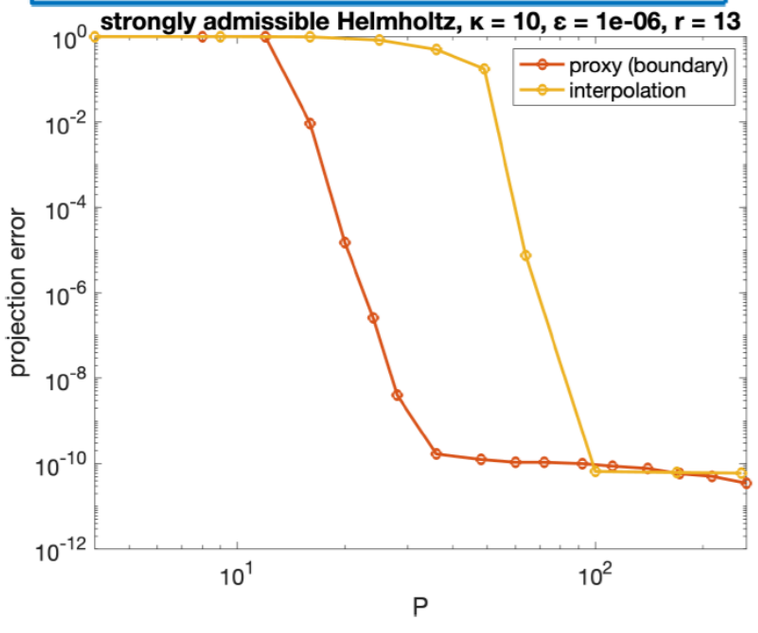
Numerical results: Proxy point method versus interpolation point method











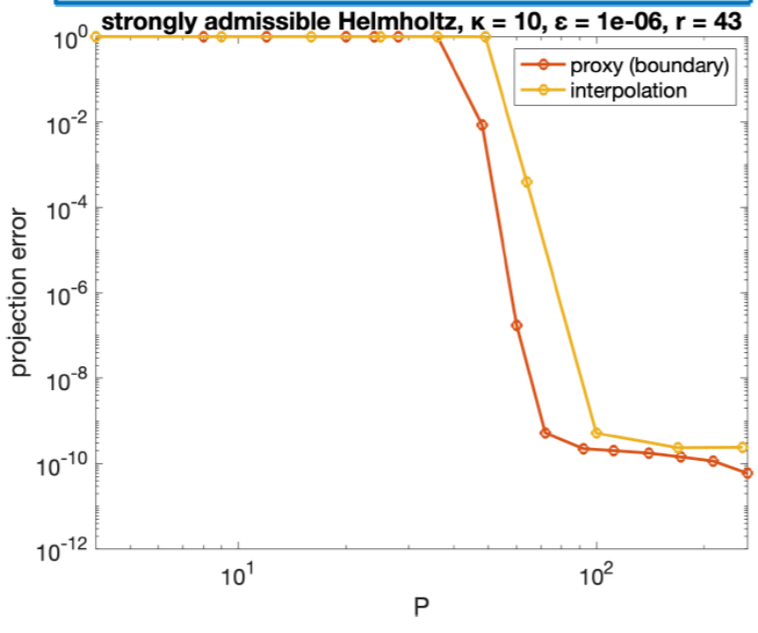


Figure 2: Projection error $\|\mathbf{U}_\varepsilon - \mathbf{Q}\mathbf{Q}^*\mathbf{U}_\varepsilon\|_F$ where $\mathbf{U}_\varepsilon$ is the left (for proxy) or right (for interpolation) singular vector basis of $\mathbf{A}$ to tolerance ε, and $\mathbf{Q}$ is the basis for the column space or row space of the surrogate matrix $\mathbf{B}$ using P proxy or interpolation points.

Proxy and interpolation surrogates are complementary; which one is better depends on the geometry.

Numerical results: Timing of ACA versus IterativeCUR (preliminary!!)

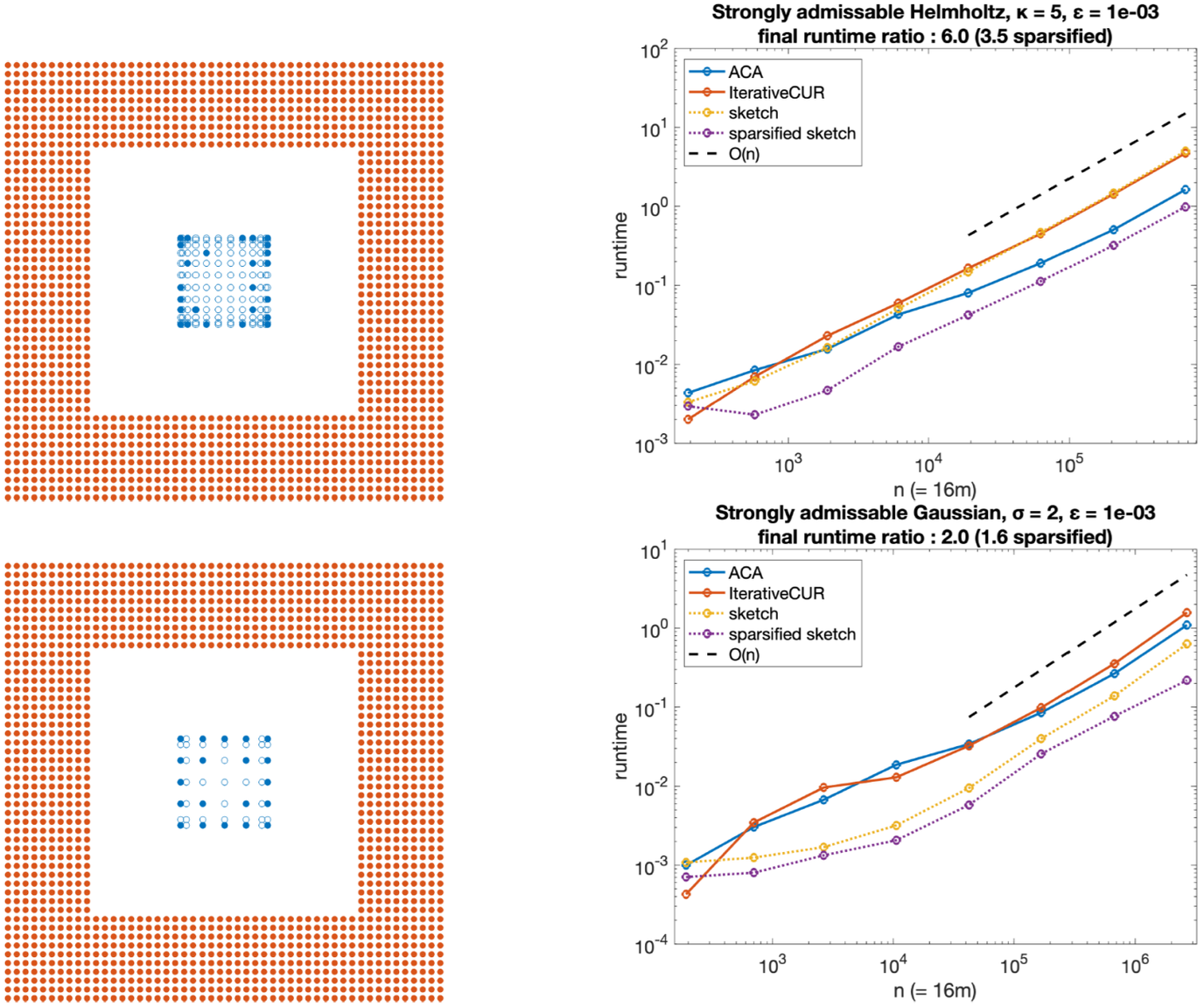


Figure 3: Runtime comparison of ACA and Interp + IterativeCUR for a Helmholtz single-layer kernel (top row) and a Gaussian (bottom row). The left column shows the equispaced source points in orange and the interpolation points in blue, with the sparsified points filled. The right column shows the runtimes including the cost of sketching with the original interpolation points (yellow) and the sparsified interpolation points (purple), of running IterativeCUR after the sketch has been computed (orange) and of running ACA. The titles give the ratio of the total runtime of IterativeCUR plus sketching to the total runtime of ACA.

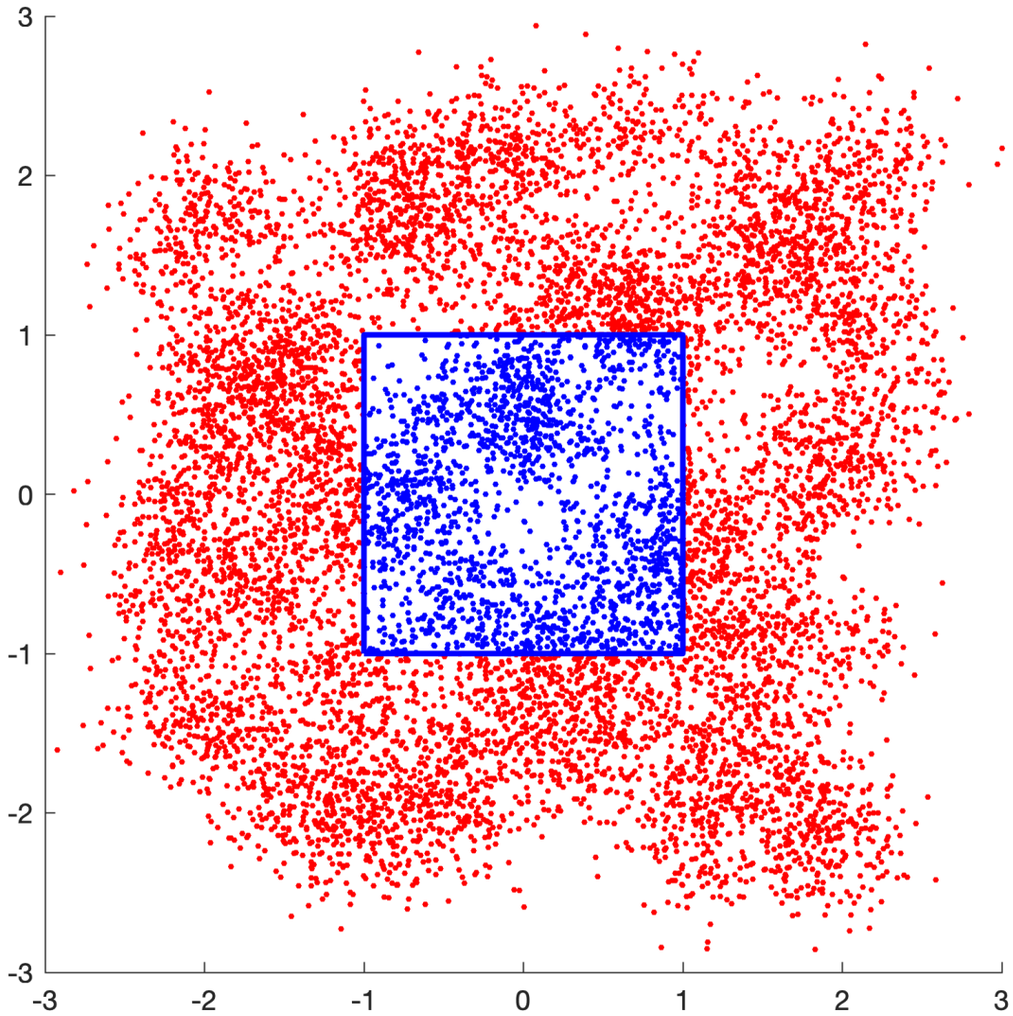
ACA is faster on these benign examples, but interpolation + IterativeCUR is within a modest constant factor and retains sketch-based error control.

Numerical results: Errors in ACA versus IterativeCUR

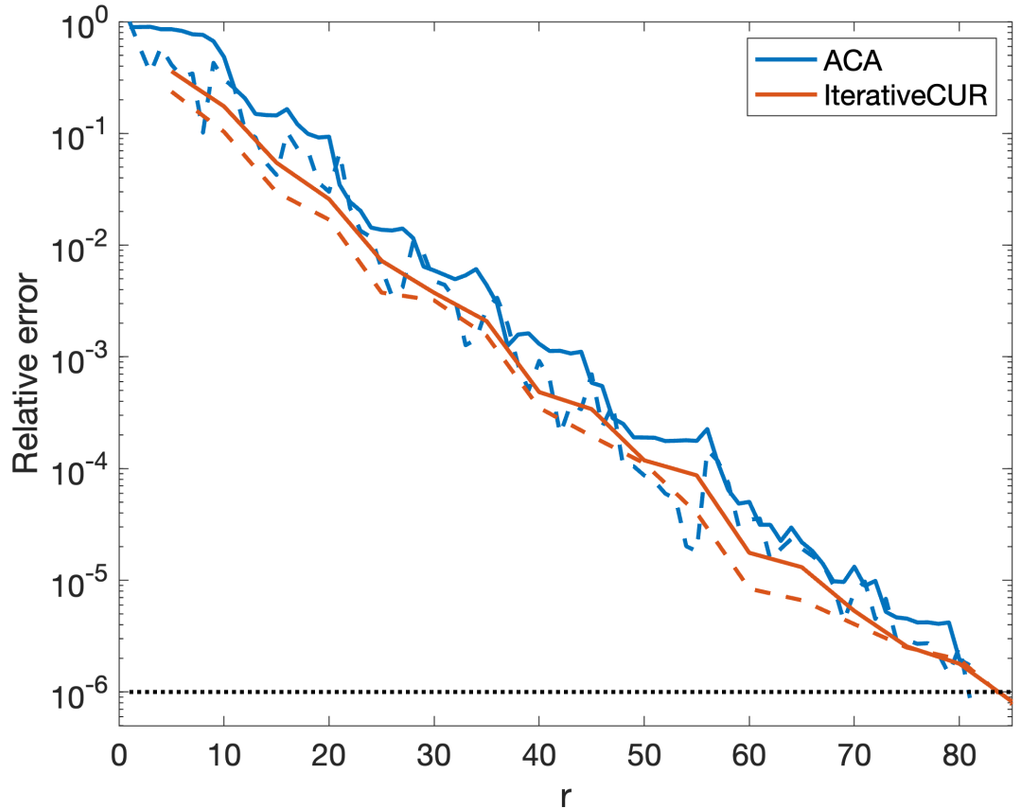
The kernel is a Gaussian, $k(x, y) = e^{-\|x-y\|^2/\sigma^2}$ for $\sigma = 1$.

Solid lines = true errors (computed only for diagnostics during testing).

Dashed lines = estimated error (IterativeCUR) / heuristic stopping indicator (ACA).



The points $\{\boldsymbol{x}_i\}_{i=1}^m$ in blue and $\{\boldsymbol{y}_j\}_{j=1}^n$ in red.



Corresponding errors and error estimators.

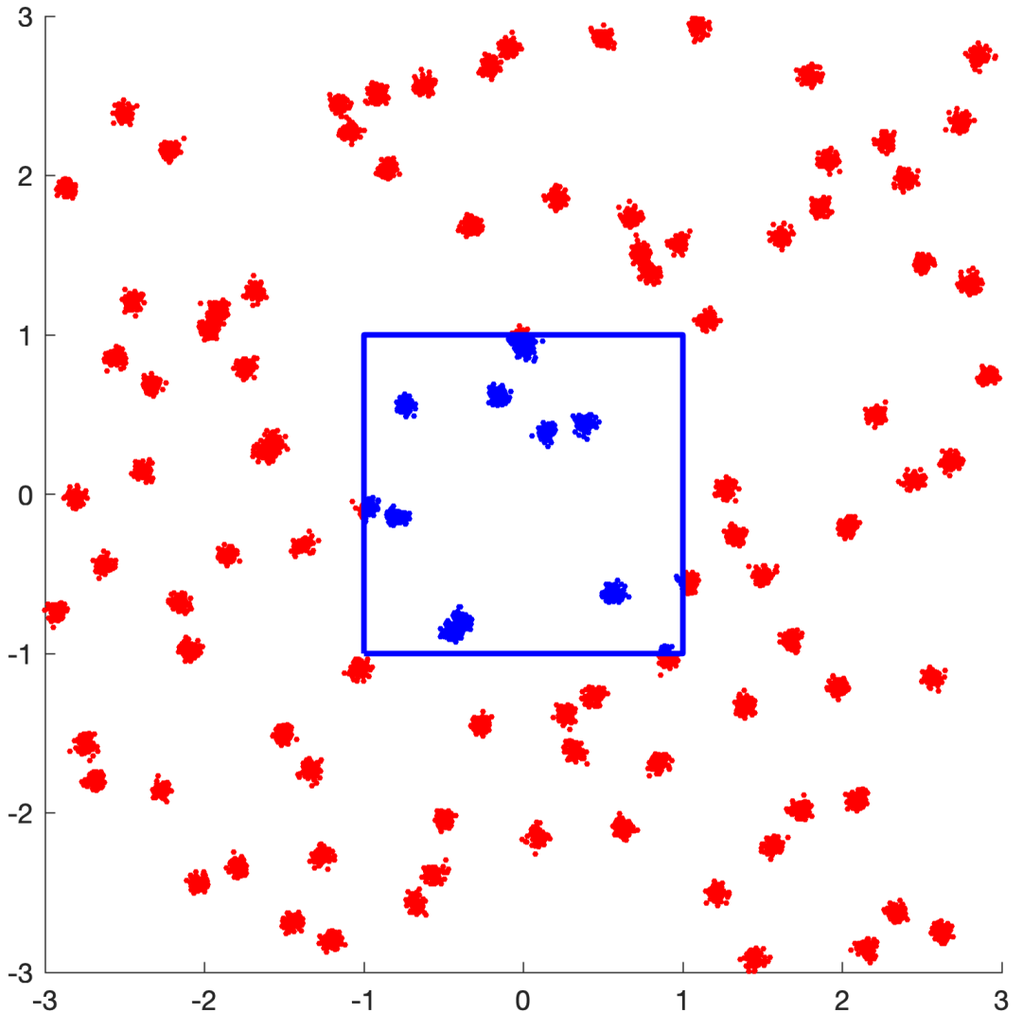
Both methods work well, but IterativeCUR more accurately reveals the true errors.

Numerical results: Errors in ACA versus IterativeCUR

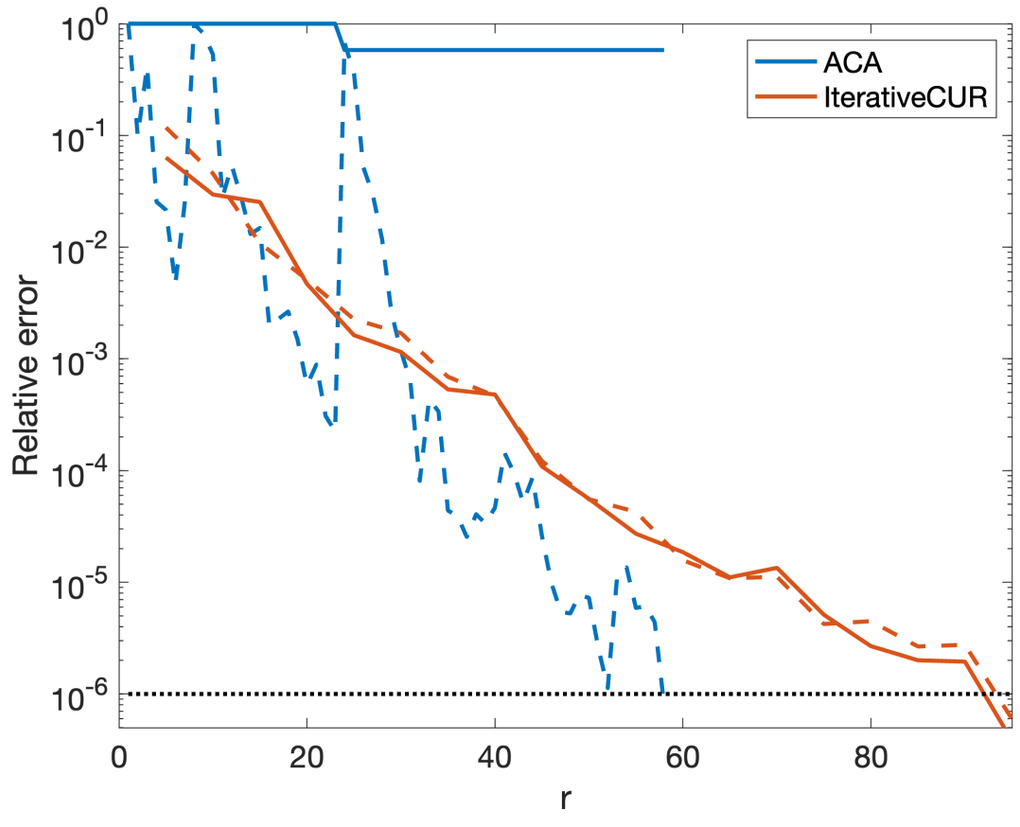
The kernel is a Gaussian, $k(x, y) = e^{-\|x-y\|^2/\sigma^2}$ for $\sigma = 0.1$.

Solid lines = true errors (computed only for diagnostics during testing).

Dashed lines = estimated error (IterativeCUR) / heuristic stopping indicator (ACA).



The points $\{\boldsymbol{x}_i\}_{i=1}^m$ in blue and $\{\boldsymbol{y}_j\}_{j=1}^n$ in red.



Corresponding errors and error estimators.

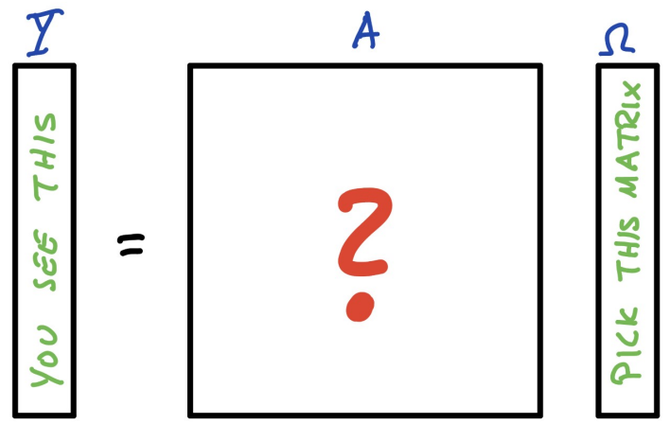
In this clustered example, ACA's stopping indicator is overly optimistic, while IterativeCUR tracks the true error and reaches the requested tolerance.

What I REALLY am interested in: Compression of a rank-structured matrices

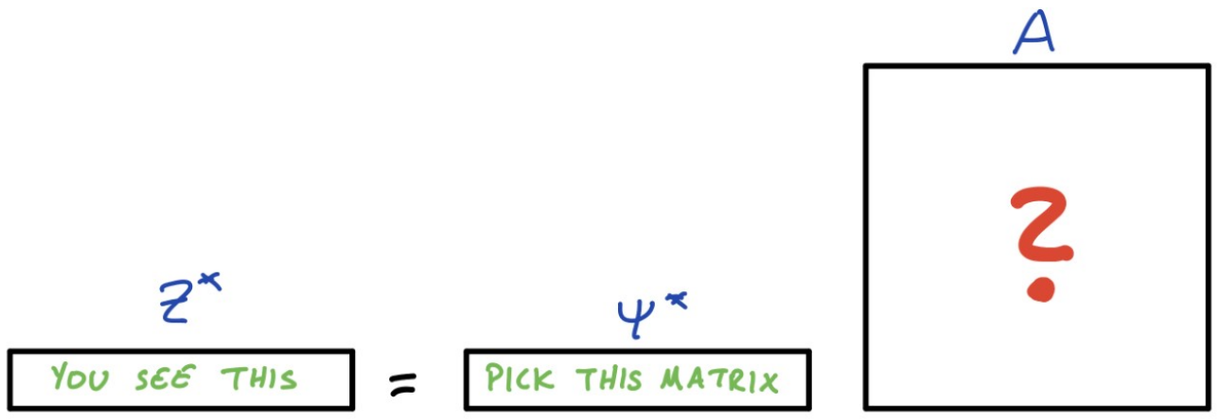
Environment: We are given a rank-structured matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$. We assume that we can evaluate $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ and $\mathbf{x} \mapsto \mathbf{A}^*\mathbf{x}$ fast.

Objective: Construct thin matrices $\mathbf{\Omega}$ and $\mathbf{\Psi}$ such that $\mathbf{A}$ can be completely reconstructed in $O(N)$ work from the set $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$ where $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$ and $\mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}$?

Sample the column space of the matrix:



If $\mathbf{A} \neq \mathbf{A}^*$, then sample the row space too:



In the present context, the application of $\mathbf{A}$ to $\mathbf{\Omega}$ typically involves running a legacy PDE solver, or applying some known fast method for the matrix-vector multiplication.

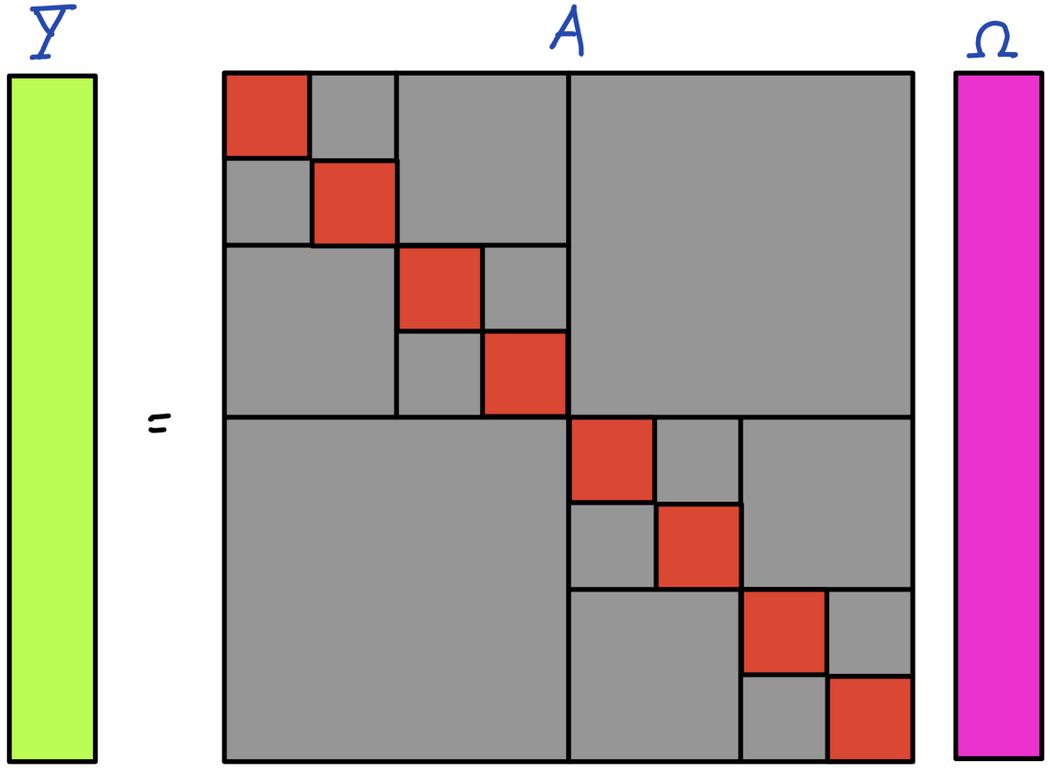
What I REALLY am interested in: Compression of a rank-structured matrices

Environment: We are given a rank-structured matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$. We assume that we can evaluate $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ and $\mathbf{x} \mapsto \mathbf{A}^*\mathbf{x}$ fast.

Objective: Construct thin matrices $\mathbf{\Omega}$ and $\mathbf{\Psi}$ such that $\mathbf{A}$ can be completely reconstructed in $O(N)$ work from the set $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$ where $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$ and $\mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}$?

The Randomized SVD solves the problem when $\mathbf{A}$ has globally low rank.

The challenge in the rank structured case is that each part of the matrix $\mathbf{Y}$ contains a mix of components from many off-diagonal blocks, and from some dense blocks too.



How do you disentangle these components?

Key points

We described a base algorithm for low rank approximation: Randomized SVD
Then various enhancements:

1. A posteriori error estimation.
2. Adaptive rank determination.
3. “Downdating” or “recycling” of a small sketch.
4. CUR/ID: Cheap since there is no need to revisit the target matrix.

Combining all four bells and whistles: Iterative CUR

- Very fast. Only interaction with the whole matrix is through *small* sketch.
- Determines the rank adaptively.
- Rock solid reliability.

Finally: Extension to kernel matrices
New randomized method exploits analytical knowledge in a systematic way to extend IterativeCUR to the case of continuum problems where the matvec is not available.
Preliminary experiments: ACA remains a strong speed baseline, but interpolation + IterativeCUR gives a reliable stopping mechanism and cleaner mathematical foundation.

Main ongoing effort: Randomized compression of *rank structured matrices.*