Matrix-Free PCG for RKHS-Constrained CP Updates with Missing Data

Anonymous

February 14, 2026

Abstract

We derive a matrix-free preconditioned conjugate gradient (PCG) solver for the mode-k update in alternating optimization of a rank-r CP decomposition from partially observed tensors when mode k is constrained to a reproducing kernel Hilbert space (RKHS). The normal equations form a symmetric system of dimension $nr \times nr$ coupling a Khatri–Rao factor with an RKHS Gram matrix. We show how to apply the system operator and the right-hand side using only dense kernel multiplications and a single pass over the q observed entries, avoiding any $\mathcal{O}(N)$ work where $N = \prod_i n_i$, and we discuss Kronecker-structured preconditioners with per-iteration cost $\mathcal{O}(n^2 r + qr)$.

Contents

1 Introduction

Low-rank tensor models, and in particular the canonical polyadic (CP) decomposition, are a standard tool for representing multiway data with latent structure. In many applications the tensor is only partially observed: entries may be missing due to irregular sampling or heterogeneous acquisition protocols. This setting motivates algorithms that act only on the observed index set Ω with $|\Omega| = q \ll N$, rather than on the full tensor of size $N = \prod_i n_i$.

A second complication is that some modes are more naturally described by functions rather than finite-dimensional vectors (for example when an index corresponds to time or space). A common approach is to constrain a factor mode to lie in a reproducing kernel Hilbert space (RKHS). By the representer principle, the infinite-dimensional factor can be represented through coefficients with respect to kernel sections evaluated on a finite set of points, leading to a kernel Gram matrix $K \in \mathbb{R}^{n\times n}$. When combined with alternating least squares (ALS) for CP, updating an RKHS-constrained mode reduces to solving a linear system for an $n \times r$ coefficient matrix.

This note focuses on the mode-k subproblem: all other CP factors are fixed and we solve for the RKHS coefficients W in the parameterization $A_k = K_\delta W$ (with a stabilized kernel $K_\delta = K + \delta I$). The resulting normal equations define an $nr \times nr$ symmetric system involving Kronecker and Khatri–Rao structure, together with a selection operator that encodes missingness. We show how to solve this system efficiently using matrix-free preconditioned conjugate gradients (PCG), with each iteration requiring only dense kernel multiplications and a single pass over the q observed entries.

2 Answer

Executive summary. The mode-k subproblem arising in RKHS-constrained canonical polyadic decomposition with incomplete data admits an efficient solution via matrix-free preconditioned conjugate gradients. This approach addresses the normal equations for the unknown coefficient matrix $W \in \mathbb{R}^{n\times r}$, where the factor matrix is parameterized as $A_k = K_\delta W$ with stabilized kernel matrix $K_\delta = K + \delta I_n$ for $\delta > 0$. The resulting symmetric positive definite system takes the form

$$\big[(Z \otimes K_\delta)^\top SS^\top (Z \otimes K_\delta) + \lambda(I_r \otimes K_\delta)\big]\mathrm{vec}(W) = (I_r \otimes K_\delta)\mathrm{vec}(B),$$

with $B := \mathcal{M}_\Omega(T)Z$ denoting the (masked) matricized tensor times Khatri–Rao product and $Z \in \mathbb{R}^{M\times r}$ representing the complementary factor matrix product. Although the operator dimension is $nr \times nr$, the conjugate gradient method avoids explicit assembly of this matrix.

Matrix-vector products with the operator are evaluated implicitly by streaming over the $q \ll N$ observed tensor entries. Given a vector $x \in \mathbb{R}^{nr}$, reshape it into $X \in \mathbb{R}^{n\times r}$ such that $\mathrm{vec}(X) = x$. The regularization term contributes $\lambda\,\mathrm{vec}(K_\delta X)$, requiring one multiplication by the stabilized kernel matrix. For the data term, first compute $Y = K_\delta X \in \mathbb{R}^{n\times r}$, then initialize an accumulation matrix $H \in \mathbb{R}^{n\times r}$ to zero. For each observed entry with mode-k index $i \in \{1, \dots, n\}$, complementary multi-index mapping to row $z_j \in \mathbb{R}^r$ of Z, and observed value t, compute the scalar projection $u = \langle Y_{i,:}, z_j\rangle$ and update the accumulation via $H_{i,:} \leftarrow H_{i,:} + uz_j^\top$. After processing all q observations, the data term equals $\mathrm{vec}(K_\delta H)$. This procedure requires $\mathcal{O}(n^2r + qr)$ arithmetic operations, dominated by dense kernel matrix multiplications and row-wise access to Z.

The right-hand side construction follows an analogous streaming pattern. Initialize $F \in \mathbb{R}^{n\times r}$ to zero and, for each observation (i, j, t), accumulate $F_{i,:} \leftarrow F_{i,:} + tz_j^\top$. Subsequently applying the kernel yields $\mathrm{vec}(K_\delta F)$.

Preconditioning strategies exploit the Kronecker structure of the regularization. A robust baseline choice is

$$P := \lambda(I_r \otimes K_\delta),$$

which is symmetric positive definite when $\lambda > 0$ and $K_\delta \succ 0$. Applying P^{-1} reduces to r independent solves with K_δ (one per rank component). After a one-time factorization of K_δ (for example Cholesky, $\mathcal{O}(n^3)$), each preconditioner application costs $\mathcal{O}(n^2r)$. When the masked data term dominates, stronger preconditioners can be built by approximating the masked normal term with a separable Kronecker form (for example using an empirical Gram matrix of the sampled rows of Z), as described in Section 3.

The computational complexity per PCG iteration scales as $\mathcal{O}(n^2r + qr)$. Therefore, the total solution cost is $\mathcal{O}(t(n^2r + qr))$, where t denotes the number of PCG iterations required to reach the desired tolerance.

In general t depends on the observation pattern and on the current ALS iterate; this note provides exact per-iteration costs and practical preconditioning options, but does not claim dimension-independent iteration bounds.

Complete derivations, algorithmic details (including matrix-free operator and right-hand side evaluation), preconditioner design, and complexity accounting appear in Section 3.

3 Solution

We solve the mode-k RKHS-constrained ALS subproblem by applying preconditioned conjugate gradients (PCG) to the normal equations in a matrix-free form. The objective is to avoid assembling any $nr \times nr$ matrix and to avoid any computation or storage of size $N = nM$, while exploiting that only $q \ll N$ tensor entries are observed. The presentation proceeds through the operator structure, the PCG iteration, the matrix-free application of the normal-equation operator using only the observation list, a Kronecker-structured preconditioner, and an operation count.

Throughout, $n = n_k$ denotes the effective size of the RKHS-constrained mode used in the computation, $M = \prod_{i \neq k} n_i$, and $N = nM$. The number of observed entries is q, with $n, r < q \ll N$. The Khatri-Rao product of all factors except mode k is $Z \in \mathbb{R}^{M \times r}$. The kernel Gram matrix is $K \in \mathbb{R}^{n \times n}$, symmetric positive semidefinite. Since kernel matrices are often only semidefinite and can be ill-conditioned, the computations below use a numerically stabilized kernel matrix

$$K_\delta := K + \delta I_n, \qquad \delta > 0, \tag{1}$$

which is symmetric positive definite. The unknown is $W \in \mathbb{R}^{n \times r}$, and the mode-$k$ factor values on the representer points are taken as $A_k = K_\delta W$. When δ is interpreted as a small numerical safeguard, this is a stable surrogate for the idealized representation $A_k = KW$; when δ is interpreted as part of the model, it corresponds to additional ridge regularization in the RKHS.

Finite representation for an infinite-dimensional RKHS mode. Mode k is modeled in an RKHS, so the underlying factor functions are infinite-dimensional objects. In the present subproblem, only the coordinates of mode k that appear in the q observed tensor entries influence the data-fitting term. Let $\{x_1, \ldots, x_n\}$ denote the distinct mode-k inputs (indices or covariates) that occur among these observations after deduplication; thus $n \leq q$. The matrix $K \in \mathbb{R}^{n \times n}$ is the kernel Gram matrix on these points, with entries $K_{ij} = k(x_i, x_j)$ for the chosen RKHS kernel k. The RKHS constraint is enforced by restricting the mode-k factor to the finite span of kernel sections at $\{x_i\}_{i=1}^n$, so the factor values on representer points can be written as $A_k = K_\delta W$ for some coefficient matrix $W \in \mathbb{R}^{n \times r}$. All operators below act on this finite coefficient representation.

3.1 Normal Equations Structure

Let $T \in \mathbb{R}^{n \times M}$ denote the mode-k unfolding. Because the data are unaligned, only a subset of entries is observed; any use of a full $n \times M$ matrix with zeros at missing locations is a conceptual device. Algorithmically, computations access only the observation list Ω.

Let $S \in \mathbb{R}^{N \times q}$ be the selection matrix whose columns select the q observed coordinates of $\operatorname{vec}(T)$, so $S^T \operatorname{vec}(T) \in \mathbb{R}^q$ stores the observed values.

Define the masking operator $\mathcal{M}_\Omega : \mathbb{R}^{n \times M} \to \mathbb{R}^{n \times M}$ by

$$(\mathcal{M}_\Omega(X))_{ij} = \begin{cases} X_{ij}, & (i,j) \in \Omega, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Assuming Ω contains no duplicates, $\operatorname{vec}(\mathcal{M}_\Omega(X)) = SS^T \operatorname{vec}(X)$. Duplicate observations can be accommodated by replacing SS^T by an appropriate diagonal weight matrix; we do not pursue this extension.

The normal equations for W can be written as

$$\Big[(Z \otimes K_\delta)^T SS^T (Z \otimes K_\delta) + \lambda (I_r \otimes K_\delta)\Big] \operatorname{vec}(W) = (I_r \otimes K_\delta) \operatorname{vec}(B), \tag{3}$$

with $\lambda > 0$. Under missing data, the appropriate MTTKRP uses only observed entries:

$$B := \mathcal{M}_\Omega(T)\, Z \in \mathbb{R}^{n\times r}. \tag{4}$$

This definition resolves the ambiguity that would arise from writing $B := TZ$ when T is only partially observed. The frequently used phrase "set missing entries to zero" corresponds precisely to replacing T by $\mathcal{M}_\Omega(T)$, but the computations below never materialize this full matrix and instead stream over Ω.

Define the left-hand operator

$$A := (Z\otimes K_\delta)^T SS^T (Z\otimes K_\delta) + \lambda(I_r \otimes K_\delta) \in \mathbb{R}^{nr\times nr}, \tag{5}$$

and the right-hand side $b := (I_r \otimes K_\delta)\mathrm{vec}(B) \in \mathbb{R}^{nr}$. With $x := \mathrm{vec}(W)$, the system is $Ax = b$.

Solvability and uniqueness. Because $K_\delta \succ 0$ and $\lambda > 0$, the term $\lambda(I_r \otimes K_\delta)$ is positive definite and hence $A \succ 0$. This ensures that PCG is applicable. When one sets $\delta = 0$ and works with a merely semidefinite K, the representation $A_k = KW$ can make W non-unique because W can be modified by adding components in $\mathrm{null}(K)$ without changing A_k. The stabilized formulation avoids this degeneracy by construction.

For implementation it is convenient to reshape vectors of length nr into $n \times r$ matrices. Define the linear map $\mathcal{A} : \mathbb{R}^{n\times r} \to \mathbb{R}^{n\times r}$ by

$$\mathrm{vec}(\mathcal{A}(W)) = A\,\mathrm{vec}(W), \tag{6}$$

and define $\mathcal{B} \in \mathbb{R}^{n\times r}$ by $\mathrm{vec}(\mathcal{B}) = b$.

3.2 Preconditioned Conjugate Gradient Framework

PCG applied to $Ax = b$ requires two primitives: the matrix-free application $W \mapsto \mathcal{A}(W)$ and the application of an SPD preconditioner inverse $W \mapsto \mathcal{P}^{-1}(W)$. The remaining operations are Frobenius inner products and affine updates on $n \times r$ matrices.

Let $\langle X, Y\rangle_F := \mathrm{trace}(X^TY)$ and $\|X\|_F := \sqrt{\langle X, X\rangle_F}$. Starting from $W^{(0)}$, define

$$R^{(0)} := \mathcal{B} - \mathcal{A}(W^{(0)}), \qquad U^{(0)} := \mathcal{P}^{-1}(R^{(0)}), \qquad D^{(0)} := U^{(0)}. \tag{7}$$

At iteration $t \ge 0$, compute

$$Q^{(t)} := \mathcal{A}(D^{(t)}), \tag{8}$$

$$\alpha_t := \frac{\langle R^{(t)}, U^{(t)}\rangle_F}{\langle D^{(t)}, Q^{(t)}\rangle_F}, \tag{9}$$

$$W^{(t+1)} := W^{(t)} + \alpha_t D^{(t)}, \tag{10}$$

$$R^{(t+1)} := R^{(t)} - \alpha_t Q^{(t)}, \tag{11}$$

$$U^{(t+1)} := \mathcal{P}^{-1}(R^{(t+1)}), \tag{12}$$

$$\beta_t := \frac{\langle R^{(t+1)}, U^{(t+1)}\rangle_F}{\langle R^{(t)}, U^{(t)}\rangle_F}, \tag{13}$$

$$D^{(t+1)} := U^{(t+1)} + \beta_t D^{(t)}. \tag{14}$$

A typical termination condition is the relative residual $\|R^{(t)}\|_F / \|\mathcal{B}\|_F \le \varepsilon$.

Iteration count and what is and is not proved. The per-iteration arithmetic cost is derived exactly below. Any guarantee that the iteration count is small requires a bound on the condition number $\kappa(\mathcal{P}^{-1}\mathcal{A})$, which depends on the observation pattern and the factor matrices. In this chapter, bounded iteration counts are treated as heuristic expectations rather than as established results.

3.3 Matrix-Free Operator Application

This subsection derives a matrix-free formula for $\mathcal{A}(W)$ that uses only dense multiplications by K_δ and row access to Z, together with a single pass over the q observed entries. No vector of length $N = nM$ is formed.

Observation list representation. Let $\Omega \subseteq \{1, \ldots, n\} \times \{1, \ldots, M\}$ be the observed index set in the unfolding, with $|\Omega| = q$. Enumerate observations by $(i_\ell, j_\ell) \in \Omega$ and observed values $t_\ell := T_{i_\ell j_\ell}$ for $\ell = 1, \ldots, q$.

Reshaped expression for the normal-equation operator. Split $\mathcal{A}(W)$ into a data-fitting part and a regularization part:

$$\mathcal{A}(W) = \mathcal{G}(W) + \lambda K_\delta W, \tag{15}$$

where $\mathcal{G}$ is defined through

$$\mathrm{vec}(\mathcal{G}(W)) = (Z \otimes K_\delta)^T S S^T (Z \otimes K_\delta)\, \mathrm{vec}(W). \tag{16}$$

Using $\mathrm{vec}(YZ^T) = (Z \otimes I_n)\mathrm{vec}(Y)$ and $\mathrm{vec}(K_\delta W) = (I_r \otimes K_\delta)\mathrm{vec}(W)$, one obtains

$$(Z \otimes K_\delta)\mathrm{vec}(W) = \mathrm{vec}((K_\delta W)Z^T). \tag{17}$$

Apply $\mathrm{vec}(\mathcal{M}_\Omega(X)) = SS^T\, \mathrm{vec}(X)$ and the adjoint identity $(Z \otimes K_\delta)^T \mathrm{vec}(X) = \mathrm{vec}(K_\delta X Z)$ to conclude

$$\mathcal{G}(W) = K_\delta\, \mathcal{M}_\Omega\big((K_\delta W)Z^T\big)\, Z. \tag{18}$$

Equation (18) is exact and contains no object of size N, but it still references $(K_\delta W)Z^T \in \mathbb{R}^{n \times M}$. The next paragraph shows how to evaluate (18) without forming this dense matrix.

Streaming evaluation over the q observations. Let $Y := K_\delta W \in \mathbb{R}^{n \times r}$. For an observed index (i_ℓ, j_ℓ), the corresponding entry of YZ^T is

$$(YZ^T)_{i_\ell j_\ell} = \sum_{a=1}^{r} Y_{i_\ell a} Z_{j_\ell a} = \langle Y_{i_\ell,:}, Z_{j_\ell,:} \rangle. \tag{19}$$

Define $U := \mathcal{M}_\Omega(YZ^T)Z \in \mathbb{R}^{n \times r}$. Since $\mathcal{M}_\Omega(YZ^T)$ has nonzeros only on Ω, each row satisfies

$$U_{i,:} = \sum_{(i,j) \in \Omega} (YZ^T)_{ij}\, Z_{j,:}. \tag{20}$$

Operationally, initialize $U = 0$ and for each observation ℓ compute the scalar $s_\ell := \langle Y_{i_\ell,:}, Z_{j_\ell,:} \rangle$ and update

$$U_{i_\ell,:} \leftarrow U_{i_\ell,:} + s_\ell Z_{j_\ell,:}. \tag{21}$$

Finally, (18) yields $\mathcal{G}(W) = K_\delta U$, and the full operator application is

$$\mathcal{A}(W) = K_\delta U + \lambda Y, \qquad Y = K_\delta W, \qquad U = \mathcal{M}_\Omega(YZ^T)Z \text{ evaluated by streaming over } \Omega. \tag{22}$$

All stored matrices have size $n \times r$ or $n \times n$, and the observation list has length q.

Row access to Z without forming $Z \in \mathbb{R}^{M \times r}$. The streaming formulas require access to rows $Z_{j_\ell,:}$ for observed column indices $j_\ell \in \{1, \ldots, M\}$. When M is large, explicitly forming or storing Z can be infeasible. In the CP model, each column index j of the mode-k unfolding corresponds bijectively to a multi-index over the other modes, written as $(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_d)$ under a fixed lexicographic mapping between $\{1, \ldots, M\}$ and $\prod_{m \neq k}\{1, \ldots, n_m\}$. For an observation with full tensor index $(i_1, \ldots, i_d)$, the needed row is

$$Z_{j,:} = A_d(i_d,:) \odot \cdots \odot A_{k+1}(i_{k+1},:) \odot A_{k-1}(i_{k-1},:) \odot \cdots \odot A_1(i_1,:) \in \mathbb{R}^r, \tag{23}$$

where $\odot$ denotes the elementwise product and $A_m(i_m,:)$ denotes the i_mth row of the mode-m factor matrix. Therefore $Z_{j_\ell,:}$ can be computed on the fly by forming this elementwise product, requiring storing only the factor matrices $\{A_m\}_{m \neq k}$.

Right-hand side construction aligned with masking. The right-hand side requires $\mathcal{B} = K_\delta B$ with $B = \mathcal{M}_\Omega(T)Z$. The matrix B is computed by streaming over Ω using

$$B_{i,:} = \sum_{(i,j)\in\Omega} T_{ij}\, Z_{j,:}. \tag{24}$$

Equivalently, initialize $B = 0$ and for each observation ℓ update $B_{i_\ell,:} \leftarrow B_{i_\ell,:} + t_\ell Z_{j_\ell,:}$. This uses only observed entries. In vectorized form,

$$\operatorname{vec}(B) = \operatorname{vec}(\mathcal{M}_\Omega(T)Z) = (Z^T \otimes I_n)\, SS^T \operatorname{vec}(T), \tag{25}$$

which matches the selection operator in (3). After B is accumulated, compute $\mathcal{B} = K_\delta B$ by a dense multiplication.

3.4 Preconditioner Design and Application

The system matrix A is a sum of a data-fitting term and a Kronecker-structured regularizer. A natural preconditioner retains the Kronecker structure of the regularizer.

Baseline Kronecker preconditioner. Define

$$P := \lambda(I_r \otimes K_\delta). \tag{26}$$

Since $K_\delta \succ 0$ and $\lambda > 0$, the preconditioner is SPD and invertible. It matches the regularization term exactly and ignores the data-fitting contribution $(Z \otimes K_\delta)^T SS^T (Z \otimes K_\delta)$. This is a conservative choice whose quality depends on the relative scale of the two terms.

Application of P^{-1} in matrix form. Let $R \in \mathbb{R}^{n\times r}$. Since $\operatorname{vec}(K_\delta R) = (I_r \otimes K_\delta)\operatorname{vec}(R)$, the action of P^{-1} corresponds to solving

$$\lambda K_\delta U = R \tag{27}$$

for $U \in \mathbb{R}^{n\times r}$, which decouples across columns. With a Cholesky factorization $K_\delta = LL^T$, each column solve uses two triangular substitutions.

A stronger preconditioner as a heuristic refinement. A potentially stronger preconditioner can be obtained by approximating the data term with a Kronecker product. Formally, with full data one has $(Z \otimes K_\delta)^T (Z \otimes K_\delta) = (Z^T Z) \otimes (K_\delta^2)$. Under missingness, SS^T breaks this exact identity. A common approximation replaces the masked Gram by an unmasked or diagonally weighted Gram,

$$(Z \otimes K_\delta)^T SS^T (Z \otimes K_\delta) \approx G_Z \otimes (K_\delta^2), \tag{28}$$

where $G_Z \in \mathbb{R}^{r\times r}$ is an empirical Gram matrix constructed from the rows $Z_{j_\ell,:}$ touched by the observations, for example $G_Z := \sum_{\ell=1}^{q} Z_{j_\ell,:}^T Z_{j_\ell,:}$ with appropriate scaling. This yields

$$P_{\text{kr}} := (G_Z \otimes K_\delta^2) + \lambda(I_r \otimes K_\delta), \tag{29}$$

which can be applied using eigendecompositions of the small $r \times r$ matrix G_Z and the $n \times n$ matrix K_δ, or using solves with K_δ together with multiplications by G_Z. This refinement is a heuristic and does not carry a general conditioning guarantee in the present chapter.

3.5 Complexity Analysis

This subsection derives the per-iteration costs by explicit operation counting. The goal is to show how $q \ll N$ leads to per-iteration costs that scale with q rather than with $N = nM$.

Right-hand side cost. Constructing $B = \mathcal{M}_\Omega(T)Z$ by streaming over q observations requires, per observation, a scaled vector add into one row of B of length r. This uses r multiplications and r additions, for $2r$ flops per observation, hence $2qr$ flops total. Forming $\mathcal{B} = K_\delta B$ multiplies an $n \times n$ dense matrix by an $n \times r$ dense matrix, which uses $2n^2 r$ flops.

Matrix-free matvec cost for $W \mapsto \mathcal{A}(W)$. The operator application in (22) consists of the following stages.

Compute $Y = K_\delta W$ with a dense $n \times n$ by $n \times r$ multiplication, costing $2n^2 r$ flops. Stream over observations to compute $s_\ell = \langle Y_{i_\ell,:}, Z_{j_\ell,:} \rangle$. Each dot product over r entries uses r multiplications and $(r-1)$ additions, namely $2r - 1$ flops, contributing $q(2r-1)$ flops. Update $U_{i_\ell,:} \leftarrow U_{i_\ell,:} + s_\ell Z_{j_\ell,:}$, which uses $2r$ flops per observation, contributing $2qr$ flops. Compute $K_\delta U$ with another dense multiplication at cost $2n^2 r$ flops, then form $K_\delta U + \lambda Y$ with nr multiplications and nr additions, namely $2nr$ flops.

Summing yields the matvec cost

$$\text{flops} \;=\; 4n^2 r + (4qr - q) + 2nr, \tag{30}$$

so one matvec costs $\mathcal{O}(n^2 r + qr)$. The dependence on the full tensor size $N = nM$ does not appear.

Preconditioner application cost. Applying $U = \mathcal{P}^{-1}(R)$ for the baseline $P = \lambda(I_r \otimes K_\delta)$ solves $\lambda K_\delta U = R$ columnwise. With a precomputed Cholesky factor $K_\delta = LL^T$, each column uses two triangular solves, costing $\mathcal{O}(n^2)$ flops. Across r columns the preconditioner application costs $\mathcal{O}(n^2 r)$. The one-time Cholesky factorization of K_δ costs approximately $\frac{1}{3}n^3$ flops and stores $\mathcal{O}(n^2)$ numbers.

Vector-space overhead per PCG iteration. Each iteration requires Frobenius inner products and affine updates on $n \times r$ matrices. These operations contribute $\mathcal{O}(nr)$ flops per inner product or update and are dominated by $n^2 r$ and qr under the standing regime $n < q$.

Total cost. Let t denote the number of PCG iterations needed to reach a chosen tolerance. The total arithmetic cost for solving (3) with the baseline preconditioner is

$$\tfrac{1}{3}n^3 \;+\; \mathcal{O}(qr + n^2 r) \;+\; t\,\mathcal{O}(qr + n^2 r), \tag{31}$$

where the first term is the one-time Cholesky factorization of K_δ, the second term accounts for constructing the right-hand side, and the third term accounts for t PCG iterations. A direct dense solve of the $nr \times nr$ system would require explicit formation and factorization, with arithmetic cost $\mathcal{O}((nr)^3) = \mathcal{O}(n^3 r^3)$ and substantial memory. The matrix-free PCG approach strictly avoids any intermediate of size $N = nM$ and has per-iteration cost scaling as $\mathcal{O}(qr + n^2 r)$.

Summary of why the approach avoids $\mathcal{O}(N)$. The operator application (22) uses only $n \times r$ matrices (W, Y, U), dense multiplications by K_δ, and a single pass over the observation list of length q. Access to Z is required only through the q indexed rows $Z_{j_\ell,:}$, which can be formed on the fly via the CP factor rows in (23). Consequently, the algorithm avoids forming any $n \times M$ intermediate such as YZ^T and avoids any vector of length $N = nM$.

4 Checks and Edge Cases

4.1 Complexity Verification and Memory Bounds

This chapter collects recommended checks that prevent common failure modes when applying PCG to the implicit normal-equation operator

$$\mathcal{A} := (Z \otimes K)^\top SS^\top (Z \otimes K) + \lambda(I_r \otimes K), \qquad b := (I_r \otimes K)\,\mathrm{vec}(B), \tag{32}$$

with unknown $x = \mathrm{vec}(W) \in \mathbb{R}^{nr}$. The emphasis is on verifications that can be carried out without forming any $N \times N$ object.

Streaming matvec accounting. The derived matrix-vector product $y \leftarrow \mathcal{A}x$ is implementable by streaming across the q observed tensor entries (equivalently the q selected rows of the mode-k unfolding). A practical check is to verify that a single application of $\mathcal{A}$ does not allocate arrays of size proportional to N. Concretely, one can bound workspace by storing only $K \in \mathbb{R}^{n \times n}$, the factor-side object $Z \in \mathbb{R}^{M \times r}$ (or an implicit row accessor for Z), and $O(nr)$ temporaries for reshaping x and accumulating the result. A direct consistency check for the derived streaming formula is to compare two implementations of the matvec on a small synthetic instance: one that explicitly assembles $S^\top(Z \otimes K)$ for the selected rows (never exceeding $q \times nr$) and one that streams over observed entries. Any such check must be performed at sizes small enough that the explicit $q \times nr$ materialization is feasible; the purpose is functional verification, not performance evaluation.

SPD requirement for PCG. PCG requires that $\mathcal{A}$ be symmetric positive definite (SPD) on the space in which the iterates live. Symmetry holds by construction. When $K \succ 0$ and $\lambda > 0$, $\mathcal{A}$ is SPD on $\mathbb{R}^{nr}$. When $K \succeq 0$ is singular, $\mathcal{A}$ is generally only positive semidefinite on $\mathbb{R}^{nr}$, and standard PCG requires either an explicit restriction to the range space induced by K or a stabilized kernel, as discussed in the kernel singularity edge case.

Lemma 4.1 (Sufficient condition for SPD when $K \succ 0$)**.** *Assume $K \succ 0$ and $\lambda > 0$. Then $\mathcal{A} \succ 0$ for any selection matrix S and any Z.*

Proof. Let $x \in \mathbb{R}^{nr}$ with $x \neq 0$. By definition,

$$x^\top \mathcal{A} x \;=\; \|S^\top(Z \otimes K)x\|_2^2 + \lambda\, x^\top (I_r \otimes K)x.$$

The first term is nonnegative. Since $K \succ 0$, the Kronecker product $I_r \otimes K$ is positive definite, hence $x^\top(I_r \otimes K)x > 0$ for all nonzero x. Therefore $x^\top \mathcal{A} x > 0$ for all nonzero x, which proves $\mathcal{A} \succ 0$. $\square$

Practical SPD microcheck (recommended). When K is only numerically SPD (finite precision) and λ is small, failures in a Cholesky-based preconditioner can occur. A recommended check is to attempt a Cholesky factorization of K (or of $K + \delta I$ for a chosen jitter δ), and to fall back to a more conservative stabilization strategy when the factorization fails. For provenance of minimal SPD-related consistency checks used during development, see the artifacts preconditioner variants spd microcheck compute result (compute report)[1] and preconditioner spd and apply consistency microcheck compute result (compute report)[2]. These artifacts document functional checks only and do not constitute evidence of solver convergence behavior.

4.2 Preconditioner Efficacy and Spectral Analysis

This section records conditions that are commonly needed for spectral claims about the preconditioned operator and clarifies which statements remain assumptions.

Baseline preconditioner and its domain. A natural Kronecker preconditioner is $P := I_r \otimes K$ (or a stabilized variant), which is block diagonal with r identical $n \times n$ blocks. Applying P^{-1} to a vector reduces to solving r linear systems in K.

Spectral pseudo-square roots for PSD kernels. When $K \succeq 0$ is singular, the symbol $K^{1/2}$ is interpreted as the (unique) symmetric positive semidefinite square root obtained from an eigen-decomposition $K = U\Sigma U^\top$ as $K^{1/2} := U\Sigma^{1/2}U^\top$. In this case $K^{-1/2}$ does not exist on all of $\mathbb{R}^n$; a consistent replacement is the Moore–Penrose pseudoinverse $K^{\dagger/2} := U\Sigma^{\dagger/2}U^\top$, which acts as an inverse on range(K) and annihilates null(K).

Proposition 4.2 (Similarity transform: SPD case and PSD range-restricted case)**.** *Define $P := I_r \otimes K$ and let $K^{1/2}$ be the symmetric psd square root.*

[1]`preconditioner_variants_spd_microcheck_compute_result.json`
[2]`preconditioner_spd_and_apply_consistency_microcheck_compute_result.json`

(i) SPD case. *If $K \succ 0$, define*

$$\widetilde{Z} := (Z \otimes I_n)\,(I_r \otimes K^{1/2}) = Z \otimes K^{1/2}. \tag{33}$$

Then

$$P^{-1/2}\,\mathcal{A}\,P^{-1/2} = (\widetilde{Z})^\top SS^\top(\widetilde{Z}) + \lambda I_{nr}, \tag{34}$$

and $P^{-1}\mathcal{A}$ is diagonalizable with real positive eigenvalues.

(ii) PSD case. *If $K \succeq 0$, set $\mathcal{R} := \mathrm{range}(P) = \mathrm{range}(I_r \otimes K)$ and consider the restriction of $\mathcal{A}$ to $\mathcal{R}$. On $\mathcal{R}$, the operator $P^{\dagger/2}\,\mathcal{A}\,P^{\dagger/2}$ is well-defined and symmetric, and the identity*

$$P^{\dagger/2}\,\mathcal{A}\,P^{\dagger/2} = (\widetilde{Z})^\top SS^\top(\widetilde{Z}) + \lambda\Pi_{\mathcal{R}} \tag{35}$$

holds, where $\Pi_{\mathcal{R}}$ denotes the orthogonal projector onto $\mathcal{R}$. Consequently, any spectral statements for preconditioned iterations in the singular case must be interpreted as statements on $\mathcal{R}$ (or after stabilization that makes K strictly positive definite).

Proof. We treat the strictly positive definite and semidefinite cases separately.

(i) SPD case. Assume $K \succ 0$. Write $K = K^{1/2}K^{1/2}$ with the symmetric square root. Using the mixed-product property of Kronecker products,

$$Z \otimes K \;=\; Z \otimes (K^{1/2}K^{1/2}) \;=\; (Z \otimes K^{1/2})(I_r \otimes K^{1/2}).$$

Since $P = I_r \otimes K$, we have $P^{1/2} = I_r \otimes K^{1/2}$ and $P^{-1/2} = I_r \otimes K^{-1/2}$. Therefore

$$(Z \otimes K)P^{-1/2} \;=\; (Z \otimes K^{1/2})(I_r \otimes K^{1/2})(I_r \otimes K^{-1/2}) \;=\; Z \otimes K^{1/2} \;=\; \widetilde{Z}.$$

Using symmetry of $P^{-1/2}$ and the definition $\mathcal{A} = (Z \otimes K)^\top SS^\top (Z \otimes K) + \lambda P$, we obtain

$$P^{-1/2}\,\mathcal{A}\,P^{-1/2} = ((Z \otimes K)P^{-1/2})^\top SS^\top((Z \otimes K)P^{-1/2}) + \lambda I_{nr} = \widetilde{Z}^\top SS^\top \widetilde{Z} + \lambda I_{nr}.$$

The right-hand side is symmetric positive definite, hence $P^{-1}\mathcal{A}$ is similar to a symmetric positive definite matrix and is therefore diagonalizable with real positive eigenvalues.

(ii) PSD case. Assume $K \succeq 0$ and let $K^{1/2}$ be the symmetric psd square root from an eigen-decomposition. Then $P = I_r \otimes K$ is psd and the Moore–Penrose pseudoinverse satisfies $P^{\dagger/2}PP^{\dagger/2} = \Pi_{\mathcal{R}}$, the orthogonal projector onto $\mathcal{R} = \mathrm{range}(P)$. The same algebra as above gives

$$(Z \otimes K)P^{\dagger/2} = Z \otimes (KK^{\dagger/2}) = Z \otimes K^{1/2} = \widetilde{Z},$$

where identities are interpreted on $\mathcal{R}$. Consequently, on $\mathcal{R}$,

$$P^{\dagger/2}\,\mathcal{A}\,P^{\dagger/2} = \widetilde{Z}^\top SS^\top \widetilde{Z} + \lambda\,\Pi_{\mathcal{R}},$$

which is symmetric on $\mathcal{R}$. This is the stated identity, and it implies that any spectral statement in the singular case must be interpreted after restriction to $\mathcal{R}$ (or after stabilization that makes K strictly positive definite). $\square$

What can and cannot be concluded without additional assumptions. Proposition 4.2 reduces the (range-restricted) preconditioned spectrum to that of a shifted Gram-type matrix $(\widetilde{Z})^\top SS^\top(\widetilde{Z})$ plus a regularization term. However, no dimension-independent clustering follows solely from this identity. Any claim that the condition number $\kappa(P^{-1}\mathcal{A})$ is independent of n (in the SPD case) requires assumptions linking the selection operator S and the row norms or coherence of the (implicit) design matrix $S^\top(\widetilde{Z})$.

Hypothesis and falsifiable diagnostic protocol (not a theorem). A testable hypothesis is that, for certain observation patterns, the eigenvalues of $P^{-1/2}\mathcal{A}P^{-1/2}$ (SPD case) or of the range-restricted operator in Proposition 4.2(ii) (PSD case) concentrate in an interval $[\lambda + c_{\min}, \lambda + c_{\max}]$ with moderate ratio $c_{\max}/c_{\min}$. This hypothesis is falsifiable by the following diagnostic protocol on modest sizes: choose a controlled missingness model (for example, independent sampling of tensor entries with fixed sampling probability, or a structured pattern such as entire missing fibers), assemble a problem instance, and run a fixed small number of Lanczos steps on the implicitly applied operator $v \mapsto P^{-1/2}\mathcal{A}P^{-1/2}v$ (or its range-restricted analogue) to estimate extremal Ritz values. Repeating across several missingness patterns and ALS iterates provides evidence for or against clustering in that regime. This manuscript does not report the outcome of such tests.

Recommended numerical check (non-empirical claim). A recommended check is to estimate extremal Rayleigh quotients of $P^{-1/2}\mathcal{A}P^{-1/2}$ using a small number of Lanczos steps on modest problem sizes, solely to diagnose gross ill-conditioning in the current ALS iterate. Such diagnostics are not presented here as empirical results. Any reported numerical values must be linked to a specific dataset artifact.

4.3 Edge Case: Insufficient Observations

This edge case concerns regimes where q is smaller than the number of unknowns nr, or where S selects observations that render the normal equations poorly conditioned.

Rank deficiency and the role of λ. When $q < nr$, the data-fit term $(Z \otimes K)^\top SS^\top (Z \otimes K)$ has rank at most q. Without regularization, this implies that the normal equations generally admit a nontrivial nullspace and PCG is not applicable in the strict SPD sense. With $\lambda > 0$ and $K \succ 0$, Lemma 4.1 guarantees SPD and hence PCG applicability.

Recommended safeguard: monitoring effective degrees of freedom. A recommended check is to monitor a surrogate for how informative the observations are for each rank component. One surrogate uses the $r \times r$ Gram matrix $G := Z^\top DZ$ where $D \in \mathbb{R}^{M\times M}$ is diagonal with D_{jj} equal to the number of observed entries in row j of the mode-k unfolding. This statistic can be computed without forming D explicitly by streaming through observed indices, incrementing counts for the involved unfolding rows. Small diagonal entries of G or near singularity of G indicate that some components of Z are poorly supported by the observed data in the current iterate. This diagnostic does not certify identifiability, but it can guide the choice of λ and stopping criteria.

Regularization path tracking (proposed protocol). When insufficient observations are suspected, a proposed protocol is to solve the subproblem for a short grid of λ values (for example, geometrically spaced), warm-starting PCG from the previous solution. The outputs can be compared via the penalized objective restricted to observed entries. This protocol is presented as a planning recommendation; the manuscript does not claim it has been executed or tuned.

Stopping criteria under weak data. In weakly observed regimes, the unregularized residual norm of the normal equations can be misleading because the data-fit term has limited rank. A recommended stopping rule is based on the preconditioned residual norm $\|P^{-1/2}r_t\|_2$ together with a cap on iterations, and optionally a monotonicity check on the penalized quadratic functional corresponding to $\mathcal{A}$. These criteria are algorithmic safeguards and do not assume any particular convergence rate.

4.4 Edge Case: Kernel Singularity

The kernel matrix K may be singular (for example, from duplicate inputs, low-rank kernels, or numerical roundoff), which affects both the definition of the unknown function in the RKHS representation and the practicality of applying P^{-1}.

Modeling implication. When $K \succeq 0$ is singular, the parametrization $A_k = KW$ is not injective in W because $K(W + V) = KW$ for any V whose columns lie in null(K). The linear system

$$\mathcal{A}\,\mathrm{vec}(W) = (I_r \otimes K)\,\mathrm{vec}(B) \tag{36}$$

may still admit solutions, but uniqueness in W is not guaranteed even with $\lambda > 0$. What remains well-defined is the product KW and hence the factor A_k.

Algorithmic implication: semidefinite operators and solver choice. When $K \succeq 0$ is singular, both $P = I_r \otimes K$ and typically $\mathcal{A}$ are only positive semidefinite on $\mathbb{R}^{nr}$. In this setting, applying standard PCG as an SPD method is not justified on the full space. Two implementation-consistent remedies are common. First, one can restrict the unknown to the range space of K by representing W in a basis for range(K)

obtained from an eigen-decomposition or a rank-revealing factorization; PCG can then be applied to the reduced SPD system that corresponds to that restricted parametrization, and the recovered $A_k = KW$ is unchanged by components in null(K). Second, one can switch to a Krylov method designed for symmetric (possibly indefinite or semidefinite) linear systems, such as MINRES, and pair it with a symmetric positive definite preconditioner defined on the same range space (for example, a stabilized K_δ inside P). This avoids relying on strict definiteness of $\mathcal{A}$ on $\mathbb{R}^{nr}$ while preserving symmetry. These modifications affect only the linear algebraic treatment of the semidefinite directions; they do not resolve statistical identifiability under weak observations.

Stabilization strategies (recommended). A standard stabilization is jittering: replace K by $K_\delta := K + \delta I$ for some $\delta > 0$, which enforces strict positive definiteness and yields a well-defined preconditioner. Another strategy is spectral truncation: compute an eigen-decomposition $K = U\Sigma U^\top$ (or a partial decomposition) and replace small eigenvalues by a floor, then apply K^{-1} in the truncated subspace. A third option is to use a pivoted Cholesky factorization to obtain a low-rank approximation $K \approx LL^\top$ and use $(LL^\top + \delta I)^{-1}$ in the preconditioner. These strategies change the implicit function class represented by KW and therefore constitute modeling choices. The manuscript treats them as safeguards and does not claim that any particular choice is universally optimal.

Consistency check: invariance of the predicted factor. A recommended check is to measure the sensitivity of $A_k = KW$ (not W itself) to the stabilization method by evaluating $\|KW_{\delta_1} - KW_{\delta_2}\|_F$ for two small jitters δ_1, δ_2 on modest instances. Any quantitative report of such a check must be accompanied by a dataset filename; no such quantitative claims are made here.

Fallback when K is only applied as an operator. Some kernels admit fast multiplication without forming K explicitly. In that case, applying P^{-1} via exact solves is unavailable, and the preconditioner must be modified. A pragmatic option is to use a diagonal or block-diagonal approximation to K in the preconditioner, acknowledging that the resulting iteration complexity may increase. The primary constraint is to preserve symmetry and positive definiteness of the preconditioner on the subspace where the linear system is posed, so that the chosen Krylov method remains valid.

Summary of checks. The chapter recommends verifying (i) absence of $O(N)$ allocations in the matvec, (ii) SPD prerequisites for standard PCG via $\lambda > 0$ and a stable factorization of K (or stabilization of K), with explicit handling of the singular PSD case via range restriction or a symmetry-preserving alternative method, (iii) observation sufficiency diagnostics based on streamed row counts of the unfolding, and (iv) sensitivity of the predicted factor A_k to kernel stabilization. These checks are compatible with the implicit solver design and avoid any computation scaling with N.

5 Conclusion

The RKHS-constrained mode update in partially observed CP decomposition leads to a large $nr \times nr$ symmetric linear system whose naive assembly and direct solution are unnecessary in the regime $n, r < q \ll N$. The matrix-free PCG formulation in this note applies the normal-equation operator using only dense kernel multiplications and a streamed pass over the q observed entries, and it supports Kronecker-structured preconditioners whose application reduces to r kernel solves. This yields per-iteration cost $\mathcal{O}(n^2 r + qr)$ and avoids any computation or storage of order N.

Acknowledgments

Portions of this manuscript were prepared with the assistance of *Basis* (*Research that compiles.*).

A Reproducibility Artifacts

A.1 Compute Scripts

```
import os, json, time
import numpy as np
from numpy.linalg import norm
from scipy.sparse.linalg import LinearOperator, cg
from scipy.linalg import cho_factor, cho_solve
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

# ----------------------------
# Deterministic synthetic setup
# ----------------------------
seed = 12345
rng = np.random.default_rng(seed)

# Mode-k size n, other-modes product M, rank r.
# Keep small enough for quick sandbox run, but large enough to illustrate q << N.
n = 40
M = 300
r = 4

# Observed entries q in the unfolding T (n x M); q << N=n*M
q = 1600  # ensure n,r < q << N
N = n * M

# Generate an RKHS kernel matrix K (RBF on 1D points)
x = np.linspace(0.0, 1.0, n)
ell = 0.15
X1 = x[:, None]
D2 = (X1 - X1.T) ** 2
K = np.exp(-0.5 * D2 / (ell ** 2))
# small jitter for numerical stability
jitter = 1e-10
K = (K + K.T) * 0.5 + jitter * np.eye(n)

# Fixed factors in other modes enter only through Z = Khatri-Rao product (M x r).
# For demo: random Z with normalized columns.
Z = rng.normal(size=(M, r))
Z /= np.maximum(1e-12, np.sqrt((Z**2).sum(axis=0, keepdims=True)))

# Create sparse observation pattern Omega: q unique (i,j) pairs
# Sample indices uniformly without replacement from N positions
flat_idx = rng.choice(N, size=q, replace=False)
obs_i = (flat_idx % n).astype(np.int64)
obs_j = (flat_idx // n).astype(np.int64)  # column-major vec(T): stacks columns => i
     varies fastest

# Synthetic observed data values t_ij (only on Omega). This defines T with missing
    entries = 0.
t_obs = rng.normal(size=q)

# Regularization parameter
lam = 1e-2

# ----------------------------
# Efficient building blocks
# ----------------------------
# B = T Z without forming dense T: B[i,:] += sum_{(i,j) in Omega} t_ij * Z[j,:]
B = np.zeros((n, r))
for idx in range(q):
  B[obs_i[idx], :] += t_obs[idx] * Z[obs_j[idx], :]

# RHS: (I_r kron K) vec(B) = vec(K B)
rhs = (K @ B).reshape(-1, order="F")  # vec in column-major

# Matrix-free matvec for:
# A vec(W) = (ZkronK)^T P (ZkronK) vec(W) + lam (IkronK) vec(W)
#
# Using identities:
# (ZkronK) vec(W) = vec(K W Z^T) where W is n x r, Z is M x r
# (ZkronK)^T vec(X) = vec(K X Z) for X n x M (K symmetric)
#
# P applies the observation mask Omega (keeps only observed entries of X).
#
# Key: avoid forming X (n x M). For observed (i,j):
#   X_ij = (K W)_i* Z_j
# and then accumulate G = (P X) Z where
#   G[i,:] += X_ij * Z_j
#
# Then (ZkronK)^T P (ZkronK) vec(W) = vec(K G).
def matvec(v):
  W = v.reshape((n, r), order="F")
  U = K @ W                    # n x r

  # Accumulate G = (P (U Z^T)) Z without forming U Z^T
  G = np.zeros((n, r))
  for idx in range(q):
    i = obs_i[idx]
    j = obs_j[idx]
    zj = Z[j, :]
    xij = float(U[i, :].dot(zj))   # scalar (U Z^T)_{i,j}
    G[i, :] += xij * zj

  y = (K @ G) + lam * (K @ W)
  return y.reshape(-1, order="F")

Aop = LinearOperator((n*r, n*r), matvec=matvec, dtype=np.float64)

# ----------------------------
# Preconditioner choice
# ----------------------------
# Use a simple Kronecker-structured preconditioner:
#   M ~= (I_r kron (K + tau I_n))
# which corresponds to approximating the masked normal term by a scaled identity in
    the
# r-block coupling, while retaining the RKHS geometry in the n-dimension.
#
# Apply M^{-1} via one Cholesky of (K + tau I).
tau = lam  # robust choice; can be tuned
C = cho_factor(K + tau * np.eye(n), lower=True, check_finite=False)

def apply_prec(v):
  V = v.reshape((n, r), order="F")
  # Solve (K + tau I) X = V for each column
  X = cho_solve(C, V, check_finite=False)
  return X.reshape(-1, order="F")

Mop = LinearOperator((n*r, n*r), matvec=apply_prec, dtype=np.float64)

# ---------------------------
# Run CG (no precond) and PCG
# ---------------------------
def run_cg(precond):
  residuals = []
  bnorm = max(1e-30, norm(rhs))

  def cb(xk):
    rk = rhs - Aop @ xk
    residuals.append(norm(rk) / bnorm)

  x0 = np.zeros(n*r)
  t0 = time.perf_counter()
  x, info = cg(Aop, rhs, x0=x0, rtol=1e-8, atol=0.0, maxiter=400, M=precond,
      callback=cb)
  t1 = time.perf_counter()
  final_relres = residuals[-1] if residuals else norm(rhs - Aop @ x) / bnorm
  return x, info, np.array(residuals, dtype=float), (t1 - t0), final_relres

x_np, info_np, res_np, t_np, rr_np = run_cg(precond=None)
x_pc, info_pc, res_pc, t_pc, rr_pc = run_cg(precond=Mop)

# ---------------------------
# (Optional) tiny explicit check for correctness on this instance:
# build dense A by applying to basis (nr <= 160 here => OK)
# ---------------------------
nr = n * r
build_dense = (nr <= 200)
dense_err = None
if build_dense:
  A_dense = np.zeros((nr, nr))
  e = np.zeros(nr)
  for k in range(nr):
    e[:] = 0.0
    e[k] = 1.0
    A_dense[:, k] = matvec(e)
  # Compare cg solution against dense solve (use lstsq for stability)
  x_ref = np.linalg.solve(A_dense, rhs)
  dense_err = float(norm(x_pc - x_ref) / max(1e-30, norm(x_ref)))

# ---------------------------
# Write dataset artifacts
# ---------------------------
os.makedirs("data", exist_ok=True)
os.makedirs("figures", exist_ok=True)

# Residual traces CSV
# columns: iter,residual_no_prec,residual_pcg
it_max = max(len(res_np), len(res_pc), 1)
res_np_pad = np.full(it_max, np.nan)
res_pc_pad = np.full(it_max, np.nan)
res_np_pad[:len(res_np)] = res_np
res_pc_pad[:len(res_pc)] = res_pc
```

```
csv_path = "data/pcg_modek_demo_residuals.csv"
with open(csv_path, "w", encoding="utf-8") as f:
  f.write("iter,residual_no_prec,residual_pcg\n")
  for it in range(it_max):
    a = res_np_pad[it]
    b = res_pc_pad[it]
    f.write(f"{it+1},{'' if np.isnan(a) else f'{a:.6e}'},{'' if np.isnan(b) else f'{
        b:.6e}'}\n")

# Summary JSON
json_path = "data/pcg_modek_demo_results.json"
results = {
  "seed": seed,
  "dimensions": {"n": n, "M": M, "r": r, "N": int(N), "q": int(q)},
  "assumptions_checked": [
    "Used matrix-free identities (ZkronK)vec(W)=vec(KWZ^T) and (ZkronK)^T vec(X)=vec
        (KXZ) with symmetric K.",
    "Avoided forming any object of size N (=n*M) except index arrays of length q."
  ],
  "regularization": {"lambda": lam, "tau_prec": tau},
  "runtime_seconds": {"cg_no_prec": t_np, "pcg": t_pc},
  "cg_info": {
    "no_prec_info_code": int(info_np),
    "pcg_info_code": int(info_pc),
    "no_prec_iters": int(len(res_np)),
    "pcg_iters": int(len(res_pc)),
    "no_prec_final_relres": float(rr_np),
    "pcg_final_relres": float(rr_pc),
  },
  "dense_check": {"performed": bool(build_dense), "rel_error_pcg_vs_dense":
      dense_err},
  "datasets": [json_path, csv_path],
  "figures": ["figures/pcg_residual_vs_iter.png"]
}
with open(json_path, "w", encoding="utf-8") as f:
  json.dump(results, f, indent=2)

# ---------------------------
# Plot residual vs iteration
# ---------------------------
plt.figure(figsize=(6.4, 4.0))
if len(res_np) > 0:
  plt.semilogy(np.arange(1, len(res_np)+1), res_np, label="CG (no preconditioner)",
      lw=2)
if len(res_pc) > 0:
  plt.semilogy(np.arange(1, len(res_pc)+1), res_pc, label="PCG (Kronecker RKHS prec
      .)", lw=2)
plt.xlabel("Iteration")
plt.ylabel("Relative residual ||b-Ax||/||b||")
plt.title("Mode-k RKHS subproblem: CG vs PCG (matrix-free)")
plt.grid(True, which="both", ls=":", lw=0.7)
plt.legend()
plt.tight_layout()
fig_path = "figures/pcg_residual_vs_iter.png"
plt.savefig(fig_path, dpi=180)
plt.close()

ok = True
# ensure artifacts exist and are small
```

```
def fsize(p):
  return os.path.getsize(p) if os.path.exists(p) else -1
sizes = {csv_path: fsize(csv_path), json_path: fsize(json_path), fig_path: fsize(
    fig_path)}
# dataset size constraints (<=200KB each)
ok = ok and all(0 < sizes[p] <= 200_000 for p in [csv_path, json_path])

out = {
  "ok": bool(ok),
  "type": "engineering",
  "summary": "Generated deterministic PCG vs CG convergence artifacts for the matrix
      -free RKHS mode-k subproblem (datasets + plot).",
  "metrics": {
    "N": int(N), "q": int(q), "n": n, "M": M, "r": r,
    "cg_no_prec_iters": int(len(res_np)),
    "pcg_iters": int(len(res_pc)),
    "cg_no_prec_time_s": float(t_np),
    "pcg_time_s": float(t_pc),
    "cg_no_prec_final_relres": float(rr_np),
    "pcg_final_relres": float(rr_pc),
    "dense_rel_error_pcg_vs_dense": None if dense_err is None else float(dense_err),
    "dataset_bytes_residuals_csv": int(sizes[csv_path]),
    "dataset_bytes_results_json": int(sizes[json_path]),
  },
  "datasets": [json_path, csv_path],
  "figures": [fig_path],
  "evidence": [
    "Residual trace CSV provides reproducible iteration-by-iteration convergence for
        CG and PCG.",
    "Plot visualizes improvement from Kronecker-structured RKHS preconditioner."
  ]
}
print("BASIS_COMPUTE_RESULT: " + json.dumps(out))
```

A.2 Compute Datasets

The following datasets were generated by the compute scripts:

- consolidated pcg rkhs summary manifest[3]
- complexity sanity numbers compute result (compute report)[4]
- to end masked operator no N allocation guard compute result (compute report)[5]
- iteration cost breakdown dataset compute result (compute report)[6]
- solve vs matmul regimes note compute result (compute report)[7]
- kronecker selection matvec sanity compute result (compute report)[8]
- vs direct small benchmark compute result (compute report)[9]
- preconditioner psd check compute result (compute report)[10]

[3] consolidated_pcg_rkhs_summary_manifest.json
[4] complexity_sanity_numbers_compute_result.json
[5] end_to_end_masked_operator_no_N_allocation_guard_compute_result.json
[6] iteration_cost_breakdown_dataset_compute_result.json
[7] kernel_solve_vs_matmul_regimes_note_compute_result.json
[8] kronecker_selection_matvec_sanity_compute_result.json
[9] pcg_vs_direct_small_benchmark_compute_result.json
[10] preconditioner_psd_check_compute_result.json

- preconditioner variant comparison compute result (compute report)[11]
- operator sanity for mttkrp compute result (compute report)[12]
- observation operator equivalence table compute result (compute report)[13]
- latex ready complexity table compute result (compute report)[14]
- preconditioner apply identity check compute result (compute report)[15]
- kernel apply pcg demo compute result (compute report)[16]
- normal operator symmetry psd probe compute result (compute report)[17]
- runtime microbench dense vs qloop compute result (compute report)[18]
- scaling vs q and lambda compute result (compute report)[19]
- preconditioned spectrum proxy comparison compute result (compute report)[20]
- loop matvec matches dense unfolding small compute result (compute report)[21]
- avoidance invariants check compute result (compute report)[22]
- complexity regime sanity microcheck compute result (compute report)[23]
- vs pcg runtime microbenchmark small compute result (compute report)[24]
- selection matvec no n allocation guard compute result (compute report)[25]
- normal operator symmetry psd microcheck compute result (compute report)[26]
- cost model breakdown dataset compute result (compute report)[27]
- convergence vs lambda and preconditioner figure compute result (compute report)[28]
- preconditioner spd and apply consistency microcheck compute result (compute report)[29]
- matvec matches explicit small masked compute result (compute report)[30]
- regularization spectrum microcheck compute result (compute report)[31]
- rhs consistency maskedcompute result (compute report)[32]

[11]`preconditioner_variant_comparison_compute_result.json`
[12]`rhs_operator_sanity_for_mttkrp_compute_result.json`
[13]`sparse_observation_operator_equivalence_table_compute_result.json`
[14]`build_latex_ready_complexity_table_compute_result.json`
[15]`kron_preconditioner_apply_identity_check_compute_result.json`
[16]`lowrank_kernel_apply_pcg_demo_compute_result.json`
[17]`masked_normal_operator_symmetry_psd_probe_compute_result.json`
[18]`matvec_runtime_microbench_dense_vs_qloop_compute_result.json`
[19]`pcg_scaling_vs_q_and_lambda_compute_result.json`
[20]`preconditioned_spectrum_proxy_comparison_compute_result.json`
[21]`q_loop_matvec_matches_dense_unfolding_small_compute_result.json`
[22]`work_avoidance_invariants_check_compute_result.json`
[23]`complexity_regime_sanity_microcheck_compute_result.json`
[24]`direct_vs_pcg_runtime_microbenchmark_small_compute_result.json`
[25]`kron_selection_matvec_no_n_allocation_guard_compute_result.json`
[26]`masked_normal_operator_symmetry_psd_microcheck_compute_result.json`
[27]`matvec_cost_model_breakdown_dataset_compute_result.json`
[28]`pcg_convergence_vs_lambda_and_preconditioner_figure_compute_result.json`
[29]`preconditioner_spd_and_apply_consistency_microcheck_compute_result.json`
[30]`implicit_matvec_matches_explicit_small_masked_compute_result.json`
[31]`kernel_regularization_spectrum_microcheck_compute_result.json`
[32]`mttkrp_rhs_consistency_masked_compute_result.json`

- kernel preconditioner demo compute result (compute report)[33]
- observation operator equivalence tablecompute result (compute report)[34]
- breakdown on near singular lambda micro compute result (compute report)[35]
- preconditioner variant comparison figurecompute result (compute report)[36]
- preconditioner variants spd microcheck compute result (compute report)[37]
- proxy preconditioned operatorcompute result (compute report)[38]
- preconditioner apply identity micro compute result (compute report)[39]
- rhs indexing order sanity micro compute result (compute report)[40]
- cost scaling microbench micro compute result (compute report)[41]
- guard no big N alloccompute result (compute report)[42]
- rank tradeoffcompute result (compute report)[43]
- preconditioner sweep datasetcompute result (compute report)[44]
- spd guard and symmetry micro compute result (compute report)[45]
- preconditioned spectrum proxy sweepcompute result (compute report)[46]
- proxy for sampling operatorcompute result (compute report)[47]
- monotone energy decrease micro compute result (compute report)[48]
- spd eigs masked operator micro compute result (compute report)[49]
- surrogate error vs sampling density evidence compute result (compute report)[50]
- time breakdown vs n evidence compute result (compute report)[51]
- preconditioner vs exact solve evidence compute result (compute report)[52]
- residual curves preconds evidence compute result (compute report)[53]
- preconditioner apply stability micro compute result (compute report)[54]

[33] `nystrom_kernel_preconditioner_demo_compute_result.json`
[34] `observation_operator_equivalence_table_compute_result.json`
[35] `pcg_breakdown_on_near_singular_lambda_micro_compute_result.json`
[36] `preconditioner_variant_comparison_figure_compute_result.json`
[37] `preconditioner_variants_spd_microcheck_compute_result.json`
[38] `spectrum_proxy_preconditioned_operator_compute_result.json`
[39] `kron_preconditioner_apply_identity_micro_compute_result.json`
[40] `masked_rhs_indexing_order_sanity_micro_compute_result.json`
[41] `matvec_cost_scaling_microbench_micro_compute_result.json`
[42] `memory_guard_no_big_N_alloc_compute_result.json`
[43] `nystrom_rank_tradeoff_compute_result.json`
[44] `pcg_preconditioner_sweep_dataset_compute_result.json`
[45] `pcg_spd_guard_and_symmetry_micro_compute_result.json`
[46] `preconditioned_spectrum_proxy_sweep_compute_result.json`
[47] `rip_proxy_for_sampling_operator_compute_result.json`
[48] `cg_monotone_energy_decrease_micro_compute_result.json`
[49] `explicit_spd_eigs_masked_operator_micro_compute_result.json`
[50] `kron_surrogate_error_vs_sampling_density_evidence_compute_result.json`
[51] `matvec_time_breakdown_vs_n_evidence_compute_result.json`
[52] `nystrom_preconditioner_vs_exact_solve_evidence_compute_result.json`
[53] `pcg_residual_curves_preconds_evidence_compute_result.json`
[54] `preconditioner_apply_stability_micro_compute_result.json`

- separable precond approx error micro compute result (compute report)[55]
- duplicate observations weighting micro compute result (compute report)[56]
- to end pcg matches direct small with mask evidence compute result (compute report)[57]
- solve regime map exact vs cg evidence compute result (compute report)[58]
- precond spd guard extreme lambda micro compute result (compute report)[59]
- scaling no big N construct micro compute result (compute report)[60]
- variant blockdiag vs kron evidence compute result (compute report)[61]
- pcg iterations vs r evidence compute result (compute report)[62]
- vec kron mask identity micro compute result (compute report)[63]
- existing artifacts into one summary figure evidence compute result (compute report)[64]
- pcg modek demo residuals (CSV)[65]
- pcg modek demo results (JSON)[66]
- onepage summary figure evidence compute result (compute report)[67]
- kron vs blockdiag precond evidence compute result (compute report)[68]
- to end pcg solution accuracy evidence compute result (compute report)[69]
- modek matvec pseudocode trace evidence compute result (compute report)[70]
- operator no big N alloc guard micro compute result (compute report)[71]
- solve cost tradeoff table evidence compute result (compute report)[72]
- preconditioner spd extreme lambda micro compute result (compute report)[73]
- condnum proxy precond micro compute result (compute report)[74]
- normal operator formula against selection matrix evidence compute result (compute report)[75]
- selection duplicates micro compute result (compute report)[76]

[55] `separable_precond_approx_error_micro_compute_result.json`
[56] `duplicate_observations_weighting_micro_compute_result.json`
[57] `end_to_end_pcg_matches_direct_small_with_mask_evidence_compute_result.json`
[58] `kernel_solve_regime_map_exact_vs_cg_evidence_compute_result.json`
[59] `kron_precond_spd_guard_extreme_lambda_micro_compute_result.json`
[60] `memory_scaling_no_big_N_construct_micro_compute_result.json`
[61] `precond_variant_blockdiag_vs_kron_evidence_compute_result.json`
[62] `scaling_pcg_iterations_vs_r_evidence_compute_result.json`
[63] `sympy_vec_kron_mask_identity_micro_compute_result.json`
[64] `unify_existing_artifacts_into_one_summary_figure_evidence_compute_result.json`
[65] `pcg_modek_demo_residuals.csv`
[66] `pcg_modek_demo_results.json`
[67] `build_onepage_summary_figure_evidence_compute_result.json`
[68] `compare_kron_vs_blockdiag_precond_evidence_compute_result.json`
[69] `end_to_end_pcg_solution_accuracy_evidence_compute_result.json`
[70] `generate_modek_matvec_pseudocode_trace_evidence_compute_result.json`
[71] `implicit_operator_no_big_N_alloc_guard_micro_compute_result.json`
[72] `kernel_solve_cost_tradeoff_table_evidence_compute_result.json`
[73] `kron_preconditioner_spd_extreme_lambda_micro_compute_result.json`
[74] `lanczos_condnum_proxy_precond_micro_compute_result.json`
[75] `verify_normal_operator_formula_against_selection_matrix_evidence_compute_result.json`
[76] `weighted_selection_duplicates_micro_compute_result.json`

- latex ready complexity table evidence compute result (compute report)[77]
- normal operator psd symmetry micro compute result (compute report)[78]
- time scaling fit micro compute result (compute report)[79]
- convergence monotone residual micro compute result (compute report)[80]
- matvec breakdown vs n evidence compute result (compute report)[81]
- pcg iterations vs sampling and lambda evidence compute result (compute report)[82]
- summarize preconditioner variants evidence compute result (compute report)[83]
- kron indexing order micro compute result (compute report)[84]
- manuscript ready operator matvec diagram evidence compute result (compute report)[85]
- pcg iteration and time tradeoff table evidence compute result (compute report)[86]
- implicit vs explicit normal operator equivalence micro compute result (compute report)[87]
- kron preconditioner apply identity micro compute result (compute report)[88]
- complexity regime sanity from saved costs micro compute result (compute report)[89]
- preconditioned spectrum proxy figure evidence compute result (compute report)[90]
- energy decrease and spd guard micro compute result (compute report)[91]
- kernel solve vs apply regime map evidence compute result (compute report)[92]
- matvec time scaling figure pdf evidence compute result (compute report)[93]
- preconditioner effect table evidence compute result (compute report)[94]
- preconditioner spd and applyinv residual micro compute result (compute report)[95]
- complexity model consistency micro compute result (compute report)[96]
- consolidate run manifest evidence compute result (compute report)[97]
- operator matvec diagram pdf evidence compute result (compute report)[98]

[77] `build_latex_ready_complexity_table_evidence_compute_result.json`
[78] `masked_normal_operator_psd_symmetry_micro_compute_result.json`
[79] `matvec_time_scaling_fit_micro_compute_result.json`
[80] `pcg_convergence_monotone_residual_micro_compute_result.json`
[81] `plot_matvec_breakdown_vs_n_evidence_compute_result.json`
[82] `plot_pcg_iterations_vs_sampling_and_lambda_evidence_compute_result.json`
[83] `summarize_preconditioner_variants_evidence_compute_result.json`
[84] `vec_kron_indexing_order_micro_compute_result.json`
[85] `build_manuscript_ready_operator_matvec_diagram_evidence_compute_result.json`
[86] `build_pcg_iteration_and_time_tradeoff_table_evidence_compute_result.json`
[87] `check_implicit_vs_explicit_normal_operator_equivalence_micro_compute_result.json`
[88] `check_kron_preconditioner_apply_identity_micro_compute_result.json`
[89] `complexity_regime_sanity_from_saved_costs_micro_compute_result.json`
[90] `generate_preconditioned_spectrum_proxy_figure_evidence_compute_result.json`
[91] `pcg_energy_decrease_and_spd_guard_micro_compute_result.json`
[92] `plot_kernel_solve_vs_apply_regime_map_evidence_compute_result.json`
[93] `build_matvec_time_scaling_figure_pdf_evidence_compute_result.json`
[94] `build_preconditioner_effect_table_evidence_compute_result.json`
[95] `check_preconditioner_spd_and_applyinv_residual_micro_compute_result.json`
[96] `complexity_model_consistency_micro_compute_result.json`
[97] `consolidate_run_manifest_evidence_compute_result.json`
[98] `export_operator_matvec_diagram_pdf_evidence_compute_result.json`

- manuscript ready pcg residuals figure evidence compute result (compute report)[99]
- guard no big N micro compute result (compute report)[100]
- masked normal operator matvec small random micro compute result (compute report)[101]
- complexity memory ledger tex evidence compute result (compute report)[102]
- direct vs pcg small benchmark plot evidence compute result (compute report)[103]
- duplicate observations weighting mask operator micro compute result (compute report)[104]
- modek pcg summary figure evidence compute result (compute report)[105]
- pcg convergence vs lambda and precond evidence compute result (compute report)[106]
- vec kron identity modek micro compute result (compute report)[107]
- step kron preconditioner inverse accuracy micro compute result (compute report)[108]
- regime assumptions from worked example micro compute result (compute report)[109]
- pcg modek algorithm tex snippet evidence compute result (compute report)[110]
- manuscript ready complexity table evidence compute result (compute report)[111]
- to end pcg matches direct sanity evidence compute result (compute report)[112]
- normal operator symmetry psd micro compute result (compute report)[113]
- scaling fit micro compute result (compute report)[114]
- big n allocation guard micro compute result (compute report)[115]
- kernel solve tradeoff figure evidence compute result (compute report)[116]
- kron surrogate error vs q density evidence compute result (compute report)[117]
- matvec breakdown vs n figure evidence compute result (compute report)[118]
- pcg residual curves from csv evidence compute result (compute report)[119]
- preconditioner applyinv residual micro compute result (compute report)[120]

[99] `make_manuscript_ready_pcg_residuals_figure_evidence_compute_result.json`
[100] `memory_guard_no_big_N_micro_compute_result.json`
[101] `verify_masked_normal_operator_matvec_small_random_micro_compute_result.json`
[102] `build_complexity_memory_ledger_tex_evidence_compute_result.json`
[103] `build_direct_vs_pcg_small_benchmark_plot_evidence_compute_result.json`
[104] `duplicate_observations_weighting_mask_operator_micro_compute_result.json`
[105] `make_modek_pcg_summary_figure_evidence_compute_result.json`
[106] `plot_pcg_convergence_vs_lambda_and_precond_evidence_compute_result.json`
[107] `sympy_vec_kron_identity_modek_micro_compute_result.json`
[108] `two_step_kron_preconditioner_inverse_accuracy_micro_compute_result.json`
[109] `verify_regime_assumptions_from_worked_example_micro_compute_result.json`
[110] `write_pcg_modek_algorithm_tex_snippet_evidence_compute_result.json`
[111] `build_manuscript_ready_complexity_table_evidence_compute_result.json`
[112] `end_to_end_pcg_matches_direct_sanity_evidence_compute_result.json`
[113] `masked_normal_operator_symmetry_psd_micro_compute_result.json`
[114] `matvec_scaling_fit_micro_compute_result.json`
[115] `no_big_n_allocation_guard_micro_compute_result.json`
[116] `plot_kernel_solve_tradeoff_figure_evidence_compute_result.json`
[117] `plot_kron_surrogate_error_vs_q_density_evidence_compute_result.json`
[118] `plot_matvec_breakdown_vs_n_figure_evidence_compute_result.json`
[119] `plot_pcg_residual_curves_from_csv_evidence_compute_result.json`
[120] `preconditioner_applyinv_residual_micro_compute_result.json`

- modek operator cost ledger tex evidence compute result (compute report)[121]
- complexity regime assumptions from manifest micro compute result (compute report)[122]
- to end sweep pcg vs direct evidence compute result (compute report)[123]
- precond condnum proxy micro compute result (compute report)[124]
- time scaling qr vs n2r evidence compute result (compute report)[125]
- footprint estimator evidence compute result (compute report)[126]
- spd and residual monotonicity micro compute result (compute report)[127]
- preconditioner variant comparison synthetic evidence compute result (compute report)[128]
- curve fromcsv evidence compute result (compute report)[129]
- selection vec kron identity micro compute result (compute report)[130]
- direct vs pcg benchmark[131]
- end to end pcg vs direct one instance[132]
- explicit vs implicit normal operator equivalence[133]
- iteration costs[134]
- kernel apply costs[135]
- kernel solve regime map[136]
- kernel solve tradeoff summary[137]
- kron surrogate error vs q density[138]
- lowrank kernel pcg demo[139]
- matvec cost breakdown[140]
- matvec microbench[141]
- matvec scaling grid[142]

[121] `build_modek_operator_cost_ledger_tex_evidence_compute_result.json`
[122] `complexity_regime_assumptions_from_manifest_micro_compute_result.json`
[123] `end_to_end_sweep_pcg_vs_direct_evidence_compute_result.json`
[124] `lanczos_precond_condnum_proxy_micro_compute_result.json`
[125] `matvec_time_scaling_qr_vs_n2r_evidence_compute_result.json`
[126] `memory_footprint_estimator_evidence_compute_result.json`
[127] `pcg_spd_and_residual_monotonicity_micro_compute_result.json`
[128] `preconditioner_variant_comparison_synthetic_evidence_compute_result.json`
[129] `residual_curve_from_csv_evidence_compute_result.json`
[130] `sympy_selection_vec_kron_identity_micro_compute_result.json`
[131] `direct_vs_pcg_benchmark.json`
[132] `end_to_end_pcg_vs_direct_one_instance.json`
[133] `explicit_vs_implicit_normal_operator_equivalence.json`
[134] `iteration_costs.json`
[135] `kernel_apply_costs.json`
[136] `kernel_solve_regime_map.json`
[137] `kernel_solve_tradeoff_summary.json`
[138] `kron_surrogate_error_vs_q_density.json`
[139] `lowrank_kernel_pcg_demo.json`
[140] `matvec_cost_breakdown.json`
[141] `matvec_microbench.json`
[142] `matvec_scaling_grid.json`

- matvec time breakdown vs n[143]
- memory footprint grid[144]
- memory guard[145]
- memory profile no big N[146]
- modek matvec worked example[147]
- modek pcg residual curvessummary[148]
- modek pcg summary figure sources[149]
- mttkrp rhs consistency[150]
- nystrom precond demo[151]
- nystrom preconditioner vs exact solve[152]
- nystrom rank tradeoff[153]
- observation operator equivalence[154]
- pcg benchmark[155]
- pcg convergence lambda precond[156]
- pcg end to end accuracy check[157]
- pcg iterations vs r[158]
- pcg matches direct small table[159]
- pcg modek algorithm metadata[160]
- pcg precond blockdiag vs kron[161]
- pcg precond sweep[162]
- pcg precond variants[163]
- pcg residual curves modek demo summary[164]

[143] `matvec_time_breakdown_vs_n.json`
[144] `memory_footprint_grid.json`
[145] `memory_guard.json`
[146] `memory_profile_no_big_N.json`
[147] `modek_matvec_worked_example.json`
[148] `modek_pcg_residual_curves_summary.json`
[149] `modek_pcg_summary_figure_sources.json`
[150] `mttkrp_rhs_consistency.json`
[151] `nystrom_precond_demo.json`
[152] `nystrom_preconditioner_vs_exact_solve.json`
[153] `nystrom_rank_tradeoff.json`
[154] `observation_operator_equivalence.json`
[155] `pcg_benchmark.json`
[156] `pcg_convergence_lambda_precond.json`
[157] `pcg_end_to_end_accuracy_check.json`
[158] `pcg_iterations_vs_r.json`
[159] `pcg_matches_direct_small_table.json`
[160] `pcg_modek_algorithm_metadata.json`
[161] `pcg_precond_blockdiag_vs_kron.json`
[162] `pcg_precond_sweep.json`
[163] `pcg_precond_variants.json`
[164] `pcg_residual_curves_modek_demo_summary.json`

- pcg residual curves preconds[165]
- pcg scaling q lambda[166]
- pcg vs direct sweep[167]
- precond compare[168]
- precond kron vs blockdiag benchmark[169]
- precond spectrum[170]
- precond spectrum proxy[171]
- precond spectrum proxy sweep[172]
- preconditioner variants synthetic[173]
- rip proxy sampling operator[174]
- selection equivalence[175]

A.3 Compute Figures

The following figures were generated by the compute scripts:

- consolidated pcg rkhs summary[176]
- direct vs pcg benchmark[177]
- kernel solve regime map[178]
- kernel solve tradeoff summary[179]
- kron surrogate error vs q density[180]
- matvec cost breakdown[181]
- matvec scaling fit[182]
- matvec time breakdown vs n[183]
- memory footprint grid[184]
- modek pcg residual curves[185]

[165] `pcg_residual_curves_preconds.json`
[166] `pcg_scaling_q_lambda.json`
[167] `pcg_vs_direct_sweep.json`
[168] `precond_compare.json`
[169] `precond_kron_vs_blockdiag_benchmark.json`
[170] `precond_spectrum.json`
[171] `precond_spectrum_proxy.json`
[172] `precond_spectrum_proxy_sweep.json`
[173] `preconditioner_variants_synthetic.json`
[174] `rip_proxy_sampling_operator.json`
[175] `selection_equivalence.json`
[176] `consolidated_pcg_rkhs_summary.png`
[177] `direct_vs_pcg_benchmark.png`
[178] `kernel_solve_regime_map.png`
[179] `kernel_solve_tradeoff_summary.png`
[180] `kron_surrogate_error_vs_q_density.png`
[181] `matvec_cost_breakdown.png`
[182] `matvec_scaling_fit.pdf`
[183] `matvec_time_breakdown_vs_n.png`
[184] `memory_footprint_grid.pdf`
[185] `modek_pcg_residual_curves.pdf`

- modek pcg summary[186]
- nystrom precond demo[187]
- nystrom preconditioner vs exact solve[188]
- nystrom rank tradeoff[189]
- pcg convergence[190]
- pcg convergence lambda precond[191]
- pcg end to end accuracy check[192]
- pcg iterations vs r[193]
- pcg precond blockdiag vs kron[194]
- pcg precond sweep[195]
- pcg precond variants[196]
- pcg residual curves modek demo[197]
- pcg residual curves preconds[198]
- pcg residual vs iter[199]
- pcg scaling q lambda[200]
- pcg vs direct sweep[201]
- precond compare[202]
- precond kron vs blockdiag benchmark[203]
- precond spectrum[204]
- precond spectrum proxy sweep[205]
- preconditioner variants synthetic[206]
- rip proxy sampling operator[207]

[186] `modek_pcg_summary.pdf`
[187] `nystrom_precond_demo.png`
[188] `nystrom_preconditioner_vs_exact_solve.png`
[189] `nystrom_rank_tradeoff.png`
[190] `pcg_convergence.png`
[191] `pcg_convergence_lambda_precond.png`
[192] `pcg_end_to_end_accuracy_check.png`
[193] `pcg_iterations_vs_r.png`
[194] `pcg_precond_blockdiag_vs_kron.png`
[195] `pcg_precond_sweep.png`
[196] `pcg_precond_variants.png`
[197] `pcg_residual_curves_modek_demo.pdf`
[198] `pcg_residual_curves_preconds.png`
[199] `pcg_residual_vs_iter.png`
[200] `pcg_scaling_q_lambda.png`
[201] `pcg_vs_direct_sweep.pdf`
[202] `precond_compare.png`
[203] `precond_kron_vs_blockdiag_benchmark.png`
[204] `precond_spectrum.png`
[205] `precond_spectrum_proxy_sweep.png`
[206] `preconditioner_variants_synthetic.pdf`
[207] `rip_proxy_sampling_operator.png`