Supplementary Information

Diversity-enhanced Neutron Microscopy

Andreas Menzel, Klaus Wakonig, Yuhe Zhang, Elisabeth Müller, Michael Schulz, Pavel Trtik
Correspondence: andreas.menzel@psi.ch

Reading the data file

The deposited stack is self-contained: it bundles the diversity scan with the raw *and* composite bright-field and dark-field frames the pipeline uses. Its datasets are the following.

Path	Shape (dtype)	Contents
/sample/frames	$740 \times 2048 \times 2048$ (uint16)	diversity scan: 74 Fermat positions $\times$ 10 exposures (position-major)
/sample/position, /sample/position_target	740×2 (float64)	commanded and nominal (x, y) per frame, in the same length units as the pixel size
/brightfield/frames	$114 \times 2048 \times 2048$ (uint16)	raw focus-scan open-beam acquisitions
/brightfield/composite	2048×2048 (float64)	their pixel-wise median — the flat actually applied
/dark/frames	$583 \times 2048 \times 2048$ (uint16)	raw beam-blocked dark acquisitions
/dark/composite	2048×2048 (float64)	their mean — the dark master subtracted by the pipeline
/static/frames	$210 \times 2048 \times 2048$ (uint16)	single-position series (the conventional single-position reference of the main text)
/static/composite	2048×2048 (float64)	its mean
/pixel_size_um	scalar	detector pixel size (µm)

All frames share the 30 s exposure and are stored as `uint16`; the provenance (source acquisition, dark index range) is recorded in the file's root attributes.

Processing pipeline and code listings

The listings below reproduce the load-bearing steps of the pipeline, in the order applied; each caption names its source script and the marginal numbers are source line numbers.

Pipeline step	Listing	Script
Temporal gamma-spot rejection	S1	remove_outliers.py
Sub-pixel registration	S2, S3	fine_align.py
Bad-pixel map	S4	badpixel_map.py
Drizzle / shift-and-add reconstruction	S5	drizzle_recon.py
Open-beam-free flat	S6, S7	blindflat_recon.py
Per-pixel uncertainty	S8	mcmc_uncertainty.py
FRC resolution metric	S9	drizzle_recon.py
Sampling-requirement resampling	S10, S11	sampling_sweep_stats.py

Listing S1. Resolution-preserving temporal gamma-spot rejection across a position's repeats (remove_outliers.py).

```
43 def burst_robust_mean(burst, ksigma=3.0):
44     """Resolution-preserving TEMPORAL outlier rejection across a burst.
45
46     burst: (T, H, W) -- T nominally-identical exposures at one scan position.
47     For each pixel, reject exposures deviating > ksigma robust-sigma from the
48     temporal median and average the rest. The replacement for a zapped pixel
49     comes from the SAME pixel at other times -- no spatial neighbourhood. Catches
50     transient outliers (zappers); does NOT catch hot/dead pixels that are
51     consistent across the burst (those look like signal temporally).
52     Returns (clean_mean, n_rejected).
53     """
54     burst = np.asarray(burst, float)
55     med = np.median(burst, axis=0)
56     mad = np.median(np.abs(burst - med), axis=0)
57     sigma = 1.4826 * mad + 1e-6
58     inlier = np.abs(burst - med) <= ksigma * sigma
59     clean = (burst * inlier).sum(0) / np.maximum(inlier.sum(0), 1)
60     return clean, int((~inlier).sum())
```

Listing S2. The nominal motor shifts and the difference-of-Gaussians band-pass (`bp`) that precede registration, and the band-passed reference FFT (`fine_align.py`).

```
    # motor shift (dy, dx) used by the reconstruction
    motor = np.stack([-T[:, 1] / pix, -T[:, 0] / pix], 1)
    # star-centre detector position per frame = C - motor
    sc = C - motor

    def bp(x):
        x = np.nan_to_num(x)
        return gaussian_filter(x, 1) - gaussian_filter(x, 20)

    # reference: centre window of the best reconstruction
    refimg = np.asarray(h5py.File(args.ref, "r")["image"])
    ry, rx = np.array(refimg.shape) // 2
    refF = np.fft.fft2(bp(refimg[ry - A // 2:ry + A // 2, rx - A // 2:rx + A // 2]))
```

Listing S3. The sub-pixel alignment iteration: band-pass, upsampled cross-correlation, accumulate the correction, stop below 0.05 px (`fine_align.py`).

```
    corr = np.zeros((N, 2))
    for it in range(args.iters):
        res = np.zeros((N, 2))
        for k in range(N):
            r = register(refF, np.fft.fft2(bp(frame_crop(k, corr[k]))), args.upsample)
            res[k] = r
            corr[k] = corr[k] + r            # accumulate (frac subtracts it -> converges)
        shifts = motor + corr
        mag = np.hypot(res[:, 0], res[:, 1])
        print(f"iter {it}: residual |r| median {np.median(mag):.3f} px, "
              f"90th {np.percentile(mag,90):.3f} px, max {mag.max():.3f}")
        if np.median(mag) < 0.05:
            break
```

Listing S4. Static hot/dead pixel flags from the averaged master dark and open beam (`badpixel_map.py`).

```
    dark_master = np.median(np.stack([read_fits(byidx[i]) for i in dark_ids]), 0)
    ob_master = np.mean(np.stack([read_fits(byidx[i]) for i in ob_ids]), 0)
    response = ob_master - dark_master

    # Absolute, physically-motivated thresholds (normal pixels ~104-125 ADU dark,
    # response ratio ~1). Hot = clear dark spike above local; dead = response well
    # below local (flat-field can't recover it); plus saturated/nonpositive.
    dd = dark_master - median_filter(dark_master, size=5)
    rratio = response / np.maximum(median_filter(response, size=5), 1.0)
    hot = dd > args.hot_adu
    dead = (rratio < args.dead_frac) | (response <= 0)
    saturated = dark_master >= 65535
    bad = hot | dead | saturated
    good = ~bad
```

Listing S5. Area-weighted deposition of frames onto the common (optionally upsampled) canvas; data (`num`) and weight (`den`) accumulate identically (`drizzle_recon.py`).

```
def drizzle(frames: np.ndarray, shifts: np.ndarray, masks: Optional[np.ndarray],
            geom: Geometry, subset: Optional[np.ndarray] = None
            ) -> Tuple[np.ndarray, np.ndarray]:
    """Deposit (a subset of) frames onto the common canvas.

    frames : (N, H, W) float32     flat-fielded transmission frames
    shifts : (N, 2)    float        sub-pixel (sy, sx) in input-pixel units
    masks  : (N, H, W) bool or None good-pixel masks (True = good)
    Returns (num, den): accumulated flux and weight; image = num / max(den, eps).
    """
    up = geom.up
    H, W = geom.in_h, geom.in_w
    hw = 0.5 * geom.pixfrac * up
    num = np.zeros((geom.out_h, geom.out_w), np.float64)
    den = np.zeros((geom.out_h, geom.out_w), np.float64)

```

```
    idx = range(frames.shape[0]) if subset is None else subset
    for k in idx:
        m = (masks[k] if masks is not None else np.ones((H, W), bool)).astype(np.float64)
        fm = frames[k].astype(np.float64) * m

        beta_y = -up * shifts[k, 0] + geom.Oy
        beta_x = -up * shifts[k, 1] + geom.Ox
        nby, fby = int(np.floor(beta_y)), beta_y - np.floor(beta_y)
        nbx, fbx = int(np.floor(beta_x)), beta_x - np.floor(beta_x)
        ky = _area_kernel(fby, hw)
        kx = _area_kernel(fbx, hw)

        for jy, wy in ky:
            ry = nby + jy
            if ry < 0:
                continue
            for jx, wx in kx:
                rx = nbx + jx
                if rx < 0:
                    continue
                w = wy * wx
                num[ry::up, rx::up][:H, :W] += w * fm
                den[ry::up, rx::up][:H, :W] += w * m
    return num, den
```

Listing S6. Detector-frame estimate of the flat $\hat{B}$: each frame contributes its dark-subtracted intensity divided by the back-shifted sample $\hat{T}$, skipping near-opaque pixels ($\hat{T} > \tau$) (`blindflat_recon.py`).

```
    def update_B(That):
        Bacc = np.zeros((a, a)); Bw = np.zeros((a, a))
        for k in range(N):
            Tdet = ndshift(That, [-shifts[k][0], -shifts[k][1]], order=1,
                           mode="constant", cval=0.0)
            w = gd * (Tdet > args.tau)
            Td = np.where(Tdet > args.tau, Tdet, 1.0)
            Bacc += w * (Im[k] - dk) / Td; Bw += w
        return Bacc / np.maximum(Bw, 1e-6)
```

Listing S7. Open-beam-free flat by alternation over `args.iters`=3 passes: `recon` reconstructs the sample from the current flat, `update_B` (Listing S6) re-estimates the flat, which is rescaled to unit median (the gauge) (`blindflat_recon.py`).

```
    # blind, open-beam-free
    print(f"\nblind flat (open-beam-free, {args.iters} iters)")
    B = np.mean(Im - dk, axis=0)                          # B^(0): detector-frame mean
    Tblind = None
    for it in range(args.iters):
        TA, TB, Tf, cov = recon(B)
        Tblind = Tf
        B = update_B(Tf)
        B = B / np.median(B[gd > 0])                      # gauge: unit-median flat
        print(f"  iter {it} done")
    Tbl, mbl = assess(B, "blind flat (final)")
```

Listing S8. Red-black Gibbs sampler for the per-pixel posterior: a Gaussian likelihood (precision from the bootstrap) plus a GMRF smoothness prior of strength `lam`; only covered pixels are updated, and running mean/variance over post-burn-in sweeps give the posterior map and its uncertainty (`mcmc_uncertainty.py`).

```
def gibbs(y, sig, lam, rng, iters, burn, cov_mask):
    """Red-black Gibbs for N(x;y,sig^2) likelihood + GMRF prior, strength lam.

    Only covered pixels are sampled; uncovered ones are frozen at 0 and excluded
    from neighbour sums, so there is no unanchored (improper) GMRF region to
    diverge. lam is an ABSOLUTE precision per neighbour edge.
    """
    covf = cov_mask.astype(np.float64)
    prec_like = np.where(cov_mask, 1.0 / np.maximum(sig, 1e-12) ** 2, 0.0)
    H, W = y.shape
    ii, jj = np.meshgrid(np.arange(H), np.arange(W), indexing="ij")
    red = ((ii + jj) % 2 == 0)
```

```
    cnt = np.zeros((H, W))                          # covered-neighbour count (static)
    for s in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        cnt += np.roll(covf, s, axis=(0, 1))
    x = np.where(cov_mask, y, 0.0).astype(np.float64)
    sm = np.zeros_like(x)
    sm2 = np.zeros_like(x)
    n = 0
    for it in range(iters):
        for color in (red, ~red):
            xc = x * covf                           # uncovered contribute nothing
            nb = (np.roll(xc, (1, 0), (0, 1)) + np.roll(xc, (-1, 0), (0, 1))
                  + np.roll(xc, (0, 1), (0, 1)) + np.roll(xc, (0, -1), (0, 1)))
            prec = prec_like + lam * cnt
            mu = np.where(prec > 0, (prec_like * y + lam * nb) / np.maximum(prec, 1e-12), 0.0)
            draw = mu + rng.standard_normal(x.shape) / np.sqrt(np.maximum(prec, 1e-12))
            upd = color & cov_mask                  # freeze uncovered pixels
            x = np.where(upd, draw, x)
        if it >= burn:
            n += 1
            d = x - sm
            sm += d / n
            sm2 += d * (x - sm)
    return sm, np.sqrt(sm2 / max(n - 1, 1))
```

Listing S9. Position-disjoint Fourier ring correlation (Hann-windowed; frequency axis in fractions of Nyquist) and the threshold crossing that defines the resolution; $\texttt{thr} \approx 1/7$ (`drizzle_recon.py`).

```
def compute_frc(a: np.ndarray, b: np.ndarray, window: bool = True):
    """Fourier ring correlation between two equal-shape images.

    A Hann window suppresses spectral leakage off the (finite, coverage-limited)
    patch edges. The DC ring is set to NaN -- it is undefined after mean
    subtraction and must be excluded from any threshold crossing.
    """
    a = a.astype(np.float64)
    b = b.astype(np.float64)
    if window:
        wy = np.hanning(a.shape[0])[:, None]
        wx = np.hanning(a.shape[1])[None, :]
        a = (a - a.mean()) * wy * wx
        b = (b - b.mean()) * wy * wx
    else:
        a = a - a.mean()
        b = b - b.mean()
    A = np.fft.fftshift(np.fft.fft2(a))
    B = np.fft.fftshift(np.fft.fft2(b))
    ny, nx = a.shape
    yy, xx = np.indices((ny, nx))
    rr = np.hypot(yy - ny / 2.0, xx - nx / 2.0)
    rmax = int(rr.max())
    binid = rr.astype(int).ravel()
    num = np.bincount(binid, (A * np.conj(B)).real.ravel(), rmax + 1)
    pa = np.bincount(binid, (np.abs(A) ** 2).ravel(), rmax + 1)
    pb = np.bincount(binid, (np.abs(B) ** 2).ravel(), rmax + 1)
    frc = num / np.sqrt(np.maximum(pa * pb, 1e-30))
    frc[0] = np.nan                                 # DC: undefined
    # Frequency axis in units of the detector NYQUIST (1.0 = Nyquist). The radial
    # bins run out to the FFT corner at r = (n/2)*sqrt(2) = sqrt(2)*Nyquist, so we
    # normalise by the Nyquist radius n/2, NOT by rmax (the corner).
    freq = np.arange(rmax + 1) / (min(ny, nx) / 2.0)
    return freq, frc


def frc_cutoff(freq: np.ndarray, frc: np.ndarray, thr: float = 0.143) -> float:
    """First spatial frequency (fraction of Nyquist) where FRC drops below thr,
    skipping the DC ring. Returns 1.0 if it never crosses (resolved to Nyquist)."""
    below = np.where((freq > 0) & np.isfinite(frc) & (frc < thr))[0]
    return float(freq[below[0]]) if below.size else 1.0
```

Listing S10. One random resample of the full sampling grid: a random position subset, a random position-disjoint split, and a random exposure subset per position; each cell scored with the open beam and open-beam-free (`sampling_sweep_stats.py`).

```python
def one_sweep(seed: int) -> dict:
    """One full grid for a single random draw. Returns {(mode,npos,nacq):(res,noise)}."""
    rng = np.random.default_rng(seed)
    P, POS, ACQ, burst, blind_iters = _G["P"], _G["POS"], _G["ACQ"], _G["burst"], _G["blind_iters"]
    flat_ob = _G["flat_ob"]
    out = {}
    for npos in POS:
        pos_ids = np.sort(rng.choice(P, npos, replace=False))
        perm = rng.permutation(pos_ids)
        A_pos, B_pos = perm[0::2], perm[1::2]
        for nacq in ACQ:
            def frames(ps):
                return [int(p) * burst + int(j) for p in ps
                        for j in rng.choice(burst, nacq, replace=False)]
            fa, fb = frames(A_pos), frames(B_pos)
            fall = fa + fb
            TA, cA = shiftadd(flat_ob, fa); TB, cB = shiftadd(flat_ob, fb)
            out[("ob", npos, nacq)] = assess(TA, TB, cA + cB)
            Bf = blind_flat(fall, blind_iters)
            TA, cA = shiftadd(Bf, fa); TB, cB = shiftadd(Bf, fb)
            out[("blind", npos, nacq)] = assess(TA, TB, cA + cB)
    return out
```

Listing S11. Sub-bin linear interpolation of the FRC threshold crossing, used for the sampling grid (`sampling_sweep_stats.py`).

```python
def frc_cutoff_interp(freq: np.ndarray, frc: np.ndarray, thr: float = 0.143) -> float:
    """First frequency (fraction of Nyquist) where the FRC crosses below thr,
    with sub-bin linear interpolation between the bracketing bins. Skips the DC
    ring; returns 1.0 if it never crosses. Falls back to the bin frequency when
    the preceding bin is DC/non-finite (no bracket to interpolate against)."""
    below = np.where((freq > 0) & np.isfinite(frc) & (frc < thr))[0]
    if not below.size:
        return 1.0
    i = int(below[0])
    f0, f1 = frc[i - 1], frc[i]
    if i - 1 <= 0 or not np.isfinite(f0) or f0 <= f1:
        return float(freq[i])                        # cannot bracket -> bin value
    t = (f0 - thr) / (f0 - f1)                        # f0 >= thr > f1 by construction
    t = min(max(t, 0.0), 1.0)
    return float(freq[i - 1] + t * (freq[i] - freq[i - 1]))
```