COMPUTO

ISSN 2824-7795

HoloML in Stan

Low-photon Image Reconstruction using Bayesian Tools

Brian Ward [1] Center for Computational Mathematics, Flatiron Institute
Bob Carpenter Center for Computational Mathematics, Flatiron Institute
David Barmherzig Center for Computational Mathematics, Flatiron Institute

Date published: 2024-02-27 Last modified: 2024-02-27

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur posuere vestibulum facilisis. Aenean pretium orci augue, quis lobortis libero accumsan eu. Nam mollis lorem sit amet pellentesque ullamcorper. Curabitur lobortis libero eget malesuada vestibulum. Nam nec nibh massa. Pellentesque porttitor cursus tellus. Mauris urna erat, rhoncus sed faucibus sit amet, venenatis eu ipsum.

Keywords: stan, coherent-diffraction-imaging, statistics

Contents

[1]Corresponding author: bward@flatironinstitute.org

1 Introduction

The HoloML technique is an approach to solving a specific kind of inverse problem inherent to imaging nanoscale specimens using X-ray diffraction.

To solve this problem in Stan, we first write down the forward scientific model given by Barmherzig and Sun, including the Poisson photon distribution and censored data inherent to the physical problem, and then find a solution via penalized maximum likelihood.

1.1 Experimental setup

In coherent diffraction imaging (CDI), a radiation source, typically an X-ray, is directed at a biomolecule or other specimen of interest, which causes diffraction. The resulting photon flux is measured by a far-field detector. The expected photon flux is approximately the squared magnitude of the Fourier transform of the electric field causing the diffraction. Inverting this to recover an image of the specimen is a problem usually known as *phase retrieval*. The phase retrieval problem is highly challenging and often lacks a unique solution (Barnett et al. 2020).

Holographic coherent diffraction imaging (HCDI) is a variant in which the specimen is placed some distance away from a known reference object, and the data observed is the pattern of diffraction around both the specimen and the reference. The addition of a reference object provides additional constraints on this problem, and transforms it into a linear deconvolution problem which has a unique, closed-form solution in the idealized setting (David A. Barmherzig et al. 2019).

The idealized version of HCDI is formulated as

- Given a reference R, data $Y = |\mathscr{F}(X + R)|^2$
- Recover the source image X

Where $\mathscr{F}$ is an oversampled Fourier transform operator.

However, the real-world set up of these experiments introduces two additional difficulties. Data is measured from a limited number of photons, where the number of photons received by each detector is modeled as Poisson distributed with expectation given by Y_{ij} (referred to in the paper as *Poisson-shot noise*). The expected number of photons each detector receives is denoted N_p. We typically have $N_p < 10$ due to the damage that radiation causes the biomolecule under observation. Secondly, to prevent damage to the detectors, the lowest frequencies are removed by a *beamstop*, which censors low-frequency observations.

The maximum likelihood estimation of the model presented here is able to recover reasonable images even under a regime featuring low photon counts and a beamstop.

2 Simulating Data

We simulate data from the generative model directly. This corresponds to the approach taken by Barmherzig and Sun, and is based on MATLAB code provided by Barmherzig.

2.1 Imports and helper code

Generating the data requires a few standard Python numerical libraries such as scipy and numpy. Matplotlib is also used to simplify loading in the source image and displaying results.

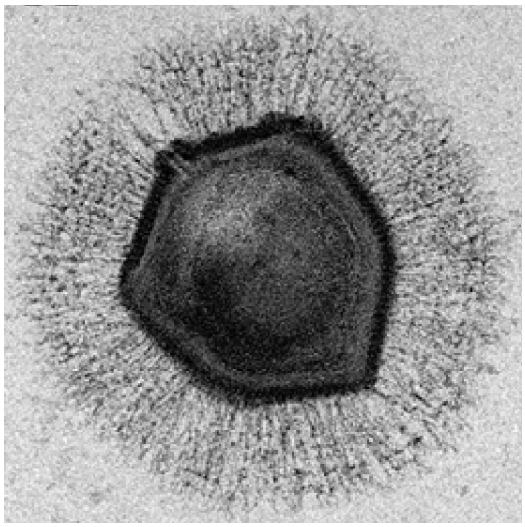
2.2 Simulation parameters

To match the figures in the paper (in particular, Figure 9), we use an image of size 256x256, $N_p = 1$ (meaning each detector is expected to receive one photon), and a beamstop of size 25x25 (corresponding to a radius of 13), and a separation `d` equal to the size of the image.

We can then load the source image used for these simulations. In this model, the pixels of X grayscale values represented on the interval $[0, 1]$. A conversion is done here from the standard RGBA encoding using the above `rgb2gray` function.

The following is a picture of a giant virus known as a mimivirus.

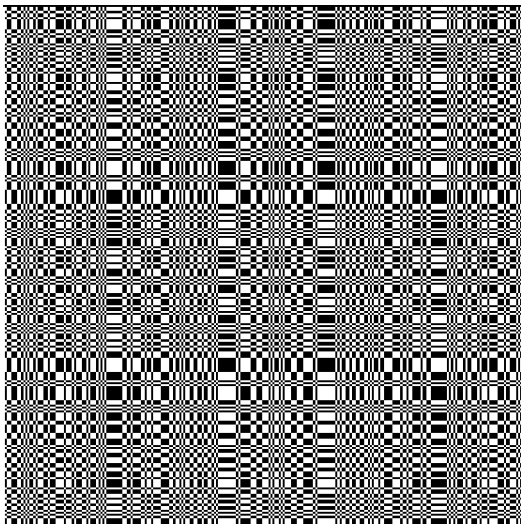
Image credit: Ghigo E, Kartenbeck J, Lien P, Pelkmans L, Capo C, Mege JL, Raoult D., CC BY 2.5, via Wikimedia Commons



Additionally, we load in the pattern of the reference object.

The pattern used here is known as a *uniformly redundant array* (URA) (Fenimore and Cannon 1978). It has been shown to be an optimal reference image for this kind of work, but other references (including none at all) could be used with the same Stan model.

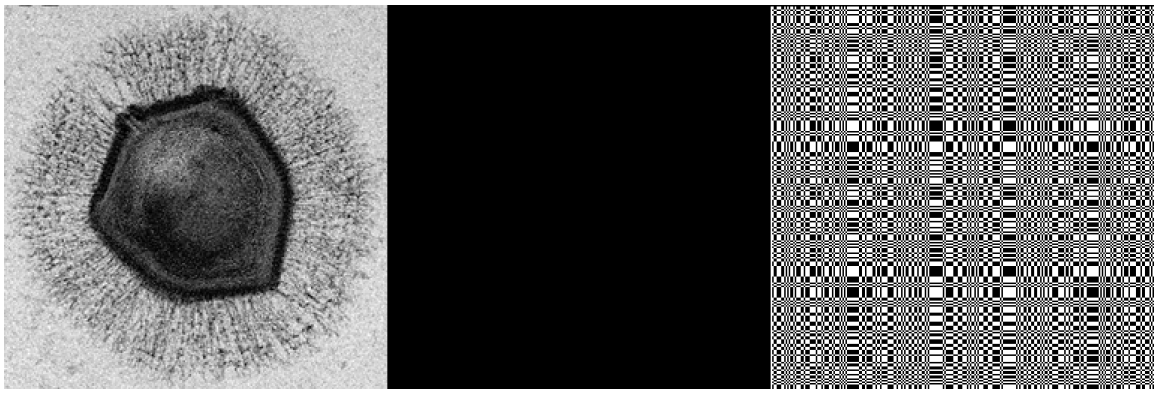
The code used to generate this grid is omitted from this case study. Various options such as cappy exist to generate these patterns in Python.



We create the specimen-reference hybrid image by concatenating the X image, a matrix of zeros, and the reference R. In the true experiment, this is done by placing the specimen some distance `d` away from the reference, with opaque material between.

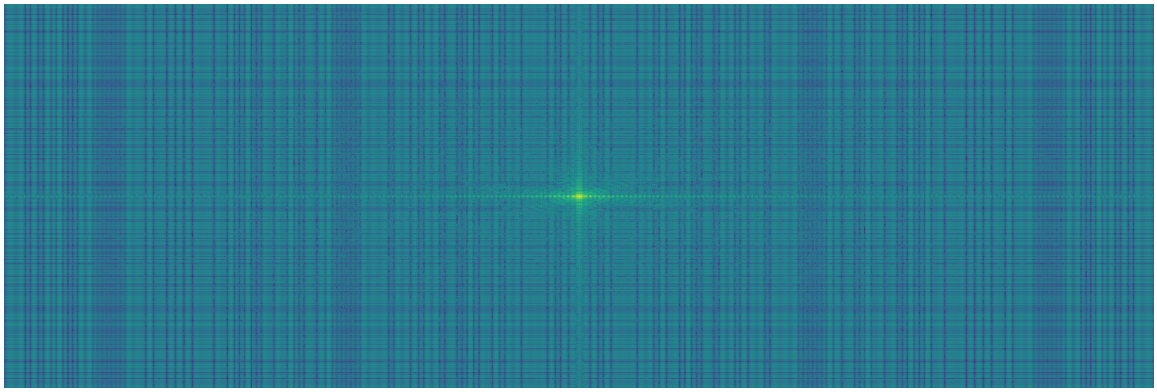
This distance is typically the same as the size of the specimen, `N`. One contribution of the HoloML model is allowing recovery with the reference placed closer to the specimen, and the Stan model allows for this as well.

For this simulation we use the separation of `d = N`.



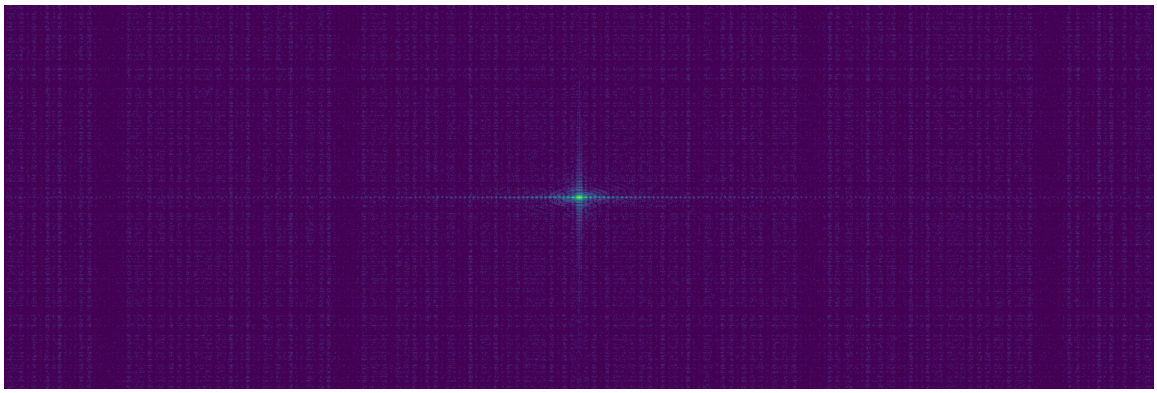
We can simulate the diffraction pattern of photons from the X-ray by taking the absolute value squared of the 2-dimensional oversampled FFT of this hybrid object.

The oversampled FFT (denoted $\mathscr{F}$ in the paper) corresponds to padding the image in both dimensions with zeros until it is a desired size. For our case, we define the size of the padded image, `M1` by `M2`, to be two times the size of our hybrid image, so the resulting FFT is twice oversampled. This is the oversampling ratio traditionally used for this problem, however Barmherzig and Sun also showed that this model can operate with less oversampling as well.



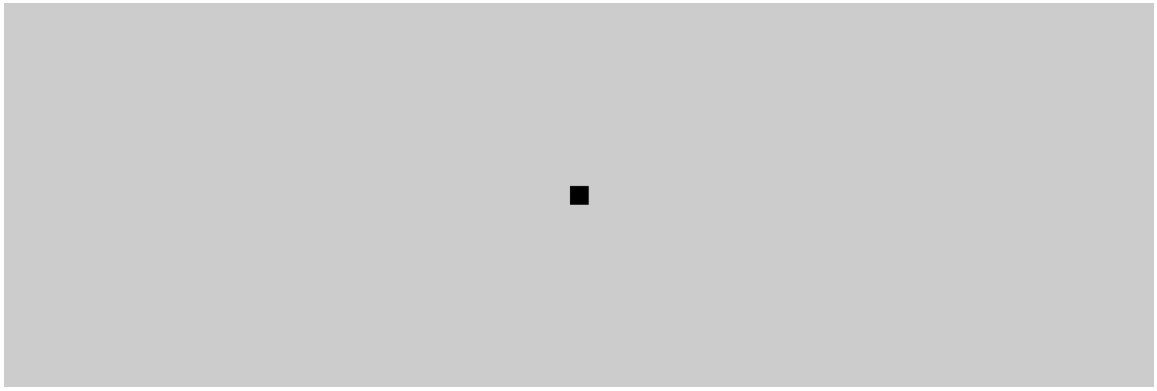
We simulate the photon fluxes with a Poisson pseudorandom number generator.

This code specifies a fixed seed to ensure the same fake data is generated each time.

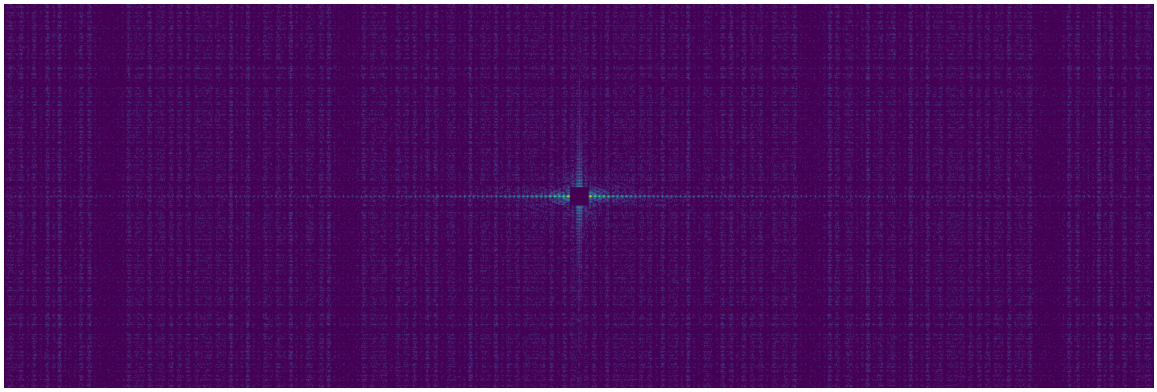


Finally, we need to remove the low frequency content of the data. This is caused in the physical experiment by the inclusion of a beamstop, which protects the instrument used by preventing the strongest parts of the beam from directly shining on the detectors.

The beamstop is represented by $\mathscr{B}$, a matrix of 0s and 1s. Zeros indicate that the data is occluded, while ones represent transparent portions.



We use this matrix $\mathscr{B}$ to mask the low frequencies of the simulated data. After removing these elements from the simulated data, we have the final input which is used in our model



3 Stan Model

The Stan model code is a direct translation of the log density of the forward model described in the paper (David A. Barmherzig and Sun 2022) and above. The full model can be seen in the appendix.

3.1 Functions

We define two helper functions to implement this model in Stan. The first is a function responsible for generating the $\mathscr{B}$ matrix. Because Stan currently does not have FFT shifting functions, this is done by manually assigning to the corners of the matrix

```
functions {
  matrix beamstop_gen(int M1, int M2, int r) {
    matrix[M1, M2] B_cal = rep_matrix(1, M1, M2);

    // upper left
    B_cal[1 : r, 1 : r] = rep_matrix(0, r, r);
    // upper right
    B_cal[1 : r, M2 - r + 2 : M2] = rep_matrix(0, r, r - 1);
    // lower left
    B_cal[M1 - r + 2 : M1, 1 : r] = rep_matrix(0, r - 1, r);
    // lower right
    B_cal[M1 - r + 2 : M1, M2 - r + 2 : M2] = rep_matrix(0, r - 1, r - 1);
    return B_cal;
  }
```

The FFT described in the paper is an oversampled FFT. This corresponds to embedding the image in a larger array of zeros and results in a sort of interpolation between frequencies in the result.

We write an overload of the `fft2` function which implements this behavior, similar to the signatures found in Matlab or Python libraries.

```
  complex_matrix fft2(complex_matrix Z, int N, int M) {
    int r = rows(Z);
    int c = cols(Z);
    complex_matrix[N, M] pad = rep_matrix(0, N, M);
    pad[1 : r, 1 : c] = Z;

    return fft2(pad);
```

```
  }
} // end functions block
```

Note that while the first input of this function is a `complex_matrix`, it will also accept real matrices due to the built-in type promotion in Stan.

3.2 Model inputs

The Stan model needs the same information the generative model did, except it is supplied with $\tilde{Y}$ instead of the source image X, plus a scale parameter for the prior, σ. Smaller values of σ (approaching 0) lead to increasing amounts of blur in the resulting image.

```
data {
  int<lower=0> N;                    // image dimension
  matrix<lower=0, upper=1>[N, N] R;  // reference image
  int<lower=0, upper=N> d;           // separation between sample and reference image
  int<lower=N> M1;                   // rows of padded matrices
  int<lower=2 * N + d> M2;           // cols of padded matrices
  int<lower=0, upper=M1> r;          // beamstop radius. replaces omega1, omega2 in paper

  real<lower=0> N_p;                  // avg number of photons per pixel
  array[M1, M2] int<lower=0> Y_tilde; // observed number of photons

  real<lower=0> sigma;                // standard deviation of pixel prior.
}
```

The constraints listed above, such as `lower=0`, perform input validation. For example, the size of the padded FFT is, at a minimum, the size of the hybrid $X0R$ specimen, and we are able to encode this in the model with the lower bounds on `M1` and `M2`.

3.3 Additional fixed information

Stan provides the ability to compute transformed data, values which depend on the inputs but only need to be evaluated once per model. This allows us to construct and store $\mathscr{B}$ once, without recomputing it each iteration or requiring it as input.

```
transformed data {
  matrix[M1, M2] B_cal = beamstop_gen(M1, M2, r);
  matrix[d, N] separation = rep_matrix(0, d, N);
}
```

3.4 Parameters

This model has only one parameter, the image X. It is constrained to grayscale values between 0 and 1.

```
parameters {
  matrix<lower=0, upper=1>[N, N] X;
}
```

3.5 Model code

3.5.1 Priors

We add a prior on X to impose an L2 penalty on adjacent pixels. This induces a Gaussian blur on the result, and it is not strictly necessary for running the model.

This prior is coded in our Stan program by looping over the rows and columns and using a vectorized call to the `normal` distribution. This results in each pixel being adjacent to 4 others. One could also formulate a prior which includes diagonally adjacent pixels

```
model {
  for (i in 1 : rows(X) - 1) {
    X[i] ~ normal(X[i + 1], sigma);
  }
  for (j in 1 : cols(X) - 1) {
    X[ : , j] ~ normal(X[ : , j + 1], sigma);
  }
```

3.5.2 Likelihood

The model likelihood encodes the forward model. We construct the hybrid specimen, compute $|\mathscr{F}(X0R)|^2$, and then compute the rate λ by scaling by the average number of photons N_p.

We then loop over this result. If the current indices are not occluded by the beamstop $\mathscr{B}$, we say that the data $\tilde{Y}$ is distributed by a Poisson distribution with λ as the rate parameter.

```
  // object representing specimen and reference together
  matrix[N, 2 * N + d] X0R = append_col(X, append_col(separation, R));
  // signal - squared magnitude of the (oversampled) FFT
  matrix[M1, M2] Y = abs(fft2(X0R, M1, M2)) .^ 2;

  real N_p_over_Y_bar = N_p / mean(Y);
  matrix[M1, M2] lambda = N_p_over_Y_bar * Y;

  for (m1 in 1 : M1) {
    for (m2 in 1 : M2) {
      if (B_cal[m1, m2] != 0) {
        Y_tilde[m1, m2] ~ poisson(lambda[m1, m2]);
      }
    }
  }
} // end model block
```

4 Optimization

Now that we have our simulated data and our generative model, we solve the inverse problem.

4.1 Data preparation

We prepare a dictionary of data corresponding to the models `data` block. This is mostly reusing constants defined earlier for the data simulation.

To run the model from Python, we instantiate it as a CmdStanModel object from cmdstanpy.

Here we use optimization via the limited-memory quasi-Newton L-BFGS algorithm. This method has a bit more curvature information than what is available to the conjugate gradient approach, but less than the second order trust-region method used in the paper. This should take a few (1-3) minutes, depending on the machine you are running on.

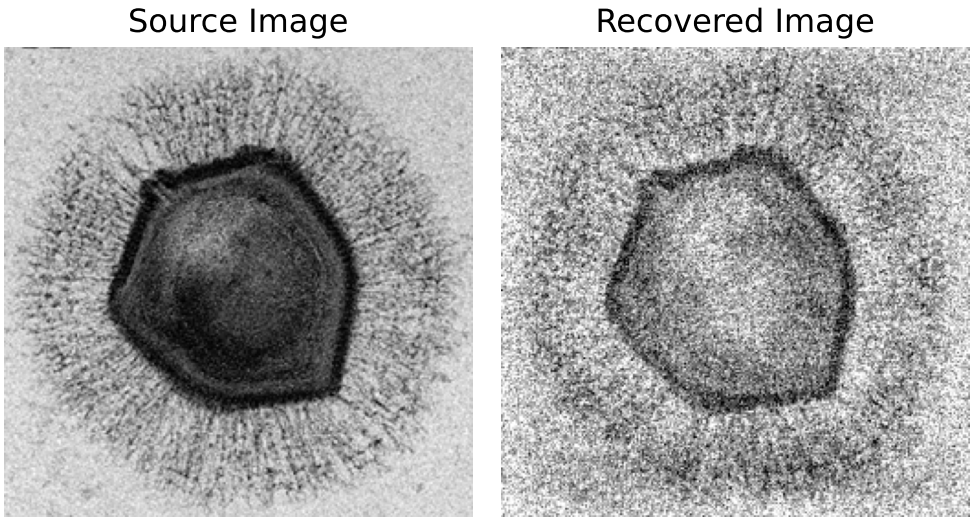
It is also possible to sample the model using the No-U-Turn Sampler (NUTS), but evaluations of this are out of the scope of this case study.

```
20:13:24 - cmdstanpy - INFO - Chain [1] start processing
20:14:40 - cmdstanpy - INFO - Chain [1] done processing
```

```
CPU times: user 628 ms, sys: 20.7 ms, total: 648 ms
Wall time: 1min 16s
```

We use the function `stan_variable` to extract the maximum likelihood estimate (MLE) from the fit object returned by optimization.

We can use this to plot the recovered image alongside the original.



5 Varying N_p

The above selection of $N_p = 1$ is a reasonable choice for real experiment, but both smaller and larger numbers of expected photons may be used. The following are results for two other levels, $N_p = 0.1$ and $N_p = 10$

This requires repeating the final few steps of the data generation and then re-fitting the model accordingly.

```
20:14:41 - cmdstanpy - INFO - Chain [1] start processing
20:16:06 - cmdstanpy - INFO - Chain [1] done processing
```

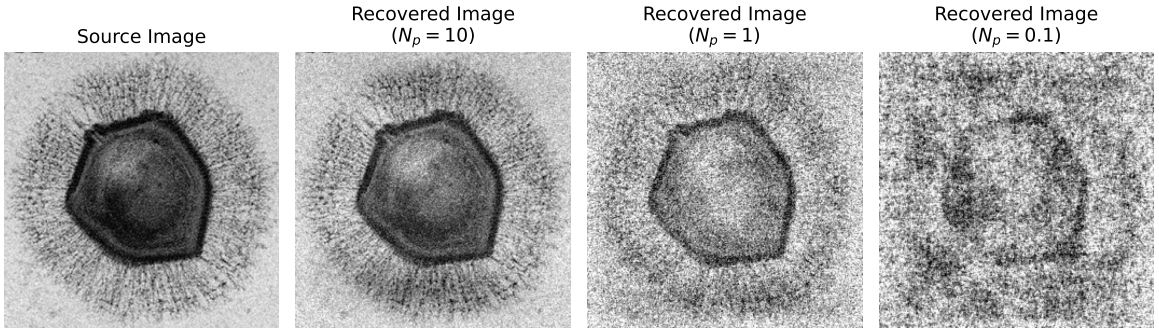
```
CPU times: user 636 ms, sys: 12.6 ms, total: 648 ms
Wall time: 1min 25s
```

```
20:16:07 - cmdstanpy - INFO - Chain [1] start processing
20:16:46 - cmdstanpy - INFO - Chain [1] done processing
```

```
CPU times: user 605 ms, sys: 31.4 ms, total: 636 ms
Wall time: 40.4 s
```

It is worth noting that these two optimizations take very different amounts of time compared to the original, as the differing amounts of data yield posteriors which are more or less normal.

In addition to the difference in runtime, the resulting images are very different.



5.1 Prior tuning

The above choice of $\sigma = 1$ has a very slight effect on the output image.

We also show the recovered image for $\sigma = 20$, which provides even less smoothing than the above, and for $\sigma = 0.05$. This smaller value imposes a greater penalty on adjacent pixels which are significantly different than each other, smoothing out the result.

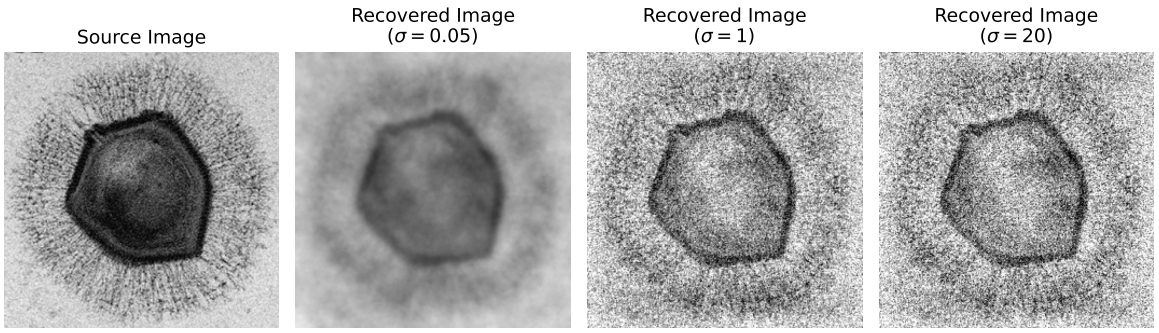
Each of these is done with the original value of $N_p = 1$

```
20:16:48 - cmdstanpy - INFO - Chain [1] start processing
20:18:15 - cmdstanpy - INFO - Chain [1] done processing

CPU times: user 633 ms, sys: 24 ms, total: 657 ms
Wall time: 1min 27s

20:18:15 - cmdstanpy - INFO - Chain [1] start processing
20:19:43 - cmdstanpy - INFO - Chain [1] done processing

CPU times: user 642 ms, sys: 4.57 ms, total: 647 ms
Wall time: 1min 27s
```



References

Barmherzig, David A., and Ju Sun. 2022. "Towards Practical Holographic Coherent Diffraction Imaging via Maximum Likelihood Estimation." *Opt. Express* 30 (5): 6886–906. https://doi.org/10.1364/OE.445015.

Barmherzig, David A, Ju Sun, Po-Nan Li, T J Lane, and Emmanuel J Candès. 2019. "Holographic Phase Retrieval and Reference Design." *Inverse Problems* 35 (9): 094001. https://doi.org/10.1088/1361-6420/ab23d1.

Barnett, Alexander H, Charles L Epstein, Leslie F Greengard, and Jeremy F Magland. 2020. "Geometry of the Phase Retrieval Problem." *Inverse Problems* 36 (9): 094003. https://doi.org/10.1088/1361-6420/aba5ed.

Fenimore, E. E., and T. M. Cannon. 1978. "Coded Aperture Imaging with Uniformly Redundant Arrays." *Appl. Opt.* 17 (3): 337–47. https://doi.org/10.1364/AO.17.000337.

Appendicies

6 Full Stan Code

```
functions {
  /**
   * Return M1 x M2 matrix of 1 values with blocks in corners set to
   * 0, where the upper left is (r x r), the upper right is (r x r-1),
   * the lower left is (r-1 x r), and the lower right is (r-1 x r-1).
   * This corresponds to zeroing out the lowest-frequency portions of
   * an FFT.
   * @param M1 number of rows
   * @param M2 number of cols
   * @param r block dimension
   * @return matrix of 1 values with 0-padded corners
   */
  matrix beamstop_gen(int M1, int M2, int r) {
    matrix[M1, M2] B_cal = rep_matrix(1, M1, M2);
    if (r == 0) {
      return B_cal;
    }
    // upper left
    B_cal[1 : r, 1 : r] = rep_matrix(0, r, r);
    // upper right
    B_cal[1 : r, M2 - r + 2 : M2] = rep_matrix(0, r, r - 1);
    // lower left
    B_cal[M1 - r + 2 : M1, 1 : r] = rep_matrix(0, r - 1, r);
    // lower right
    B_cal[M1 - r + 2 : M1, M2 - r + 2 : M2] = rep_matrix(0, r - 1, r - 1);
    return B_cal;
  }

  /**
   * Return the matrix corresponding to the fast Fourier
   * transform of Z after it is padded with zeros to size
   * N by M
   * When N by M is larger than the dimensions of Z,
   * this computes an oversampled FFT.
   *
   * @param Z matrix of values
   * @param N number of rows desired (must be >= rows(Z))
```

```
   * @param M number of columns desired (must be >= cols(Z))
   * @return the FFT of Z padded with zeros
   */
  complex_matrix fft2(complex_matrix Z, int N, int M) {
    int r = rows(Z);
    int c = cols(Z);
    if (r > N) {
      reject("N must be at least rows(Z)");
    }
    if (c > M) {
      reject("M must be at least cols(Z)");
    }

    complex_matrix[N, M] pad = rep_matrix(0, N, M);
    pad[1 : r, 1 : c] = Z;

    return fft2(pad);
  }
}
data {
  int<lower=0> N; // image dimension
  matrix<lower=0, upper=1>[N, N] R; // registration image
  int<lower=0, upper=N> d; // separation between sample and registration image
  int<lower=N> M1; // rows of padded matrices
  int<lower=2 * N + d> M2; // cols of padded matrices
  int<lower=0, upper=M1> r; // beamstop radius. replaces omega1, omega2 in paper

  real<lower=0> N_p; // avg number of photons per pixel
  array[M1, M2] int<lower=0> Y_tilde; // observed number of photons

  real<lower=0> sigma; // standard deviation of pixel prior.
}
transformed data {
  matrix[M1, M2] B_cal = beamstop_gen(M1, M2, r);
  matrix[d, N] separation = rep_matrix(0, d, N);
}
parameters {
  matrix<lower=0, upper=1>[N, N] X;
}
model {
  // prior - penalizing L2 on adjacent pixels
  for (i in 1 : rows(X) - 1) {
    X[i] ~ normal(X[i + 1], sigma);
  }
  for (j in 1 : cols(X) - 1) {
    X[ : , j] ~ normal(X[ : , j + 1], sigma);
  }

  // likelihood
  // object representing specimen and reference together
```

```
  matrix[N, 2 * N + d] XOR = append_col(X, append_col(separation, R));
  // signal - squared magnitude of the (oversampled) FFT
  matrix[M1, M2] Y = abs(fft2(XOR, M1, M2)) .^ 2;

  real N_p_over_Y_bar = N_p / mean(Y);
  matrix[M1, M2] lambda = N_p_over_Y_bar * Y;

  for (m1 in 1 : M1) {
    for (m2 in 1 : M2) {
      if (B_cal[m1, m2] != 0) {
        Y_tilde[m1, m2] ~ poisson(lambda[m1, m2]);
      }
    }
  }
}
```

6.1 Digression: Efficiency

The model above is coded for readability and sticks closely to the mathematical formulation of the process. However, this does lead to an inefficient condition inside the tightest loop of the model to handle the beamstop occlusion.

In practice, it is possible to avoid this conditional by changing how the data is stored. Instead of storing the beamstop occlusion as a parallel matrix, we can pre-compute the list of indices which are included once and store it. Then, we can create flat representations of both the data $\tilde{Y}$ and the rate λ, allowing us to use a vectorized version of the Poisson distribution.

```
transformed data {
  array[M1, M2] int B_cal = beamstop_gen(M1, M2, r);
  int total = sum(to_array_1d(B_cal));
  array[total, 2] idxs;
  // pre-compute indices
  int current = 1;
  for (n in 1:M1){
    for (m in 1:M2){
      if (B_cal[n, m]){
        idxs[current, :] = {n,m};
        current += 1;
      }
    }
  }
  // flatten data accordingly
  array[total] int<lower=0> Ys;
  for (n in 1:total) {
    Ys[n] = Y_tilde[idxs[n, 1], idxs[n, 2]];
  }
}
model {
  // ... same code for computing matrix[M1, M2] lambda here
  array[total] real lambdas;
  for (n in 1:total) {
```

```
    lambdas[n] = lambda[idxs[n, i], idxs[n, j]];  // much cheaper than branching
  }

  Ys ~ poisson(lambdas);  // fully vectorized
}
```

This formulation of the model reduces the amount of time per gradient evaluation by 15-20%. A brief evaluation suggests however that the impact on optimization runtime is minimal.