Compression of Averaging Kernels and Covariance Matrices in TROPESS L2 Ozone Archival Data Products

Matthew Thill

October 21, 2024

1 Introduction

In this document, we describe the algorithms and software that we have produced to compress the archival TROPESS data products. The bulk of the data stored in the uncompressed files is attributed to one of the following datasets:

1. Averaging Kernels
2. Measurement Error Covariances
3. Prior Covariances
4. Total Error Covariances
5. Observation Error Covariances

Each of these datasets consists of a set of matrices corresponding to each sounding in the file, and as such, can be viewed as a 3D data object. Typical dimensions of each such 3D object will be $d \times d \times N$, where d corresponds to the number of atmospheric pressure levels (typically $d = 67$) and N is the number of soundings within the file, roughly on the order of 30,000-35,000. These are typically stored with single-point precision, so each accounts for approximately 500-600 MB. When a given sounding has no data for some of the d pressure levels (as is the case for levels beyond the surface pressure), the corresponding data entries are populated with a constant “fill” value, v_{fill}, which by default is -999.

Our compression algorithm consists of two overall steps. The first is a transformation step, where we apply a transformation to all matrices in a single 3D data object. The second is a compression step, where we apply Rice-Golomb coding to all entries of the transformed matrices. Our algorithm is designed to minimize the data needed to characterize the transformation, while exploiting commonalities over multiple soundings to increase our compression factor in such a way that we can efficiently reconstruct the matrices for individual user-specified soundings. We find, in practice, that we can compress the 3D datasets by a factor of at least 18 without degrading the quality of data assimilations.

We organize this report as follows: In the first section, we describe our preprocessing methods (Section 2.1) and how we transform the data (Section 2.2). In Section 2.3, we discuss the compression portion of our algorithm. In Section 3, we detail the format of our final compressed data product. In Section 4, we provide a quick user's guide for how to use our software in Python. **Users may wish to skip directly to Section 4 for immediate use.**

2 Data Transformation and Compression

Previous studies [1, 2] have attempted to find suitable bases in which to transform a data assimilation problem to reduce the dimensionality of the data which must be stored. In the case of atmospheric retrievals, these methods include a combination of selecting a basis in which the transformed retrieval error has unit covariance (eliminating the need to store this covariance information) and in which the bulk of the information content in the retrieved state vector is combined to a small number of basis elements (allowing less significant basis components to be discarded). In practice, neglecting less significant basis components can create systematic errors in the data assimilation, and the extra data needed to store the basis transformation accumulates significantly due to variation between soundings.

To address these issues, we sought to design a simple transformation which is fast to compute and common to all soundings, and such that the resulting variance in the transformed parameters is low. We combined this with lossy compression, imposing the additional requirement that negligible systematic error would be introduced to the reconstructed data products.

Our method is straightforward. We will describe each $d \times d \times N$ three-dimensional object as an ordered set of $d \times d$ matrices, $\{\mathbf{A}_k\}_{k=1}^{N}$. This is either the set of averaging kernels or one of the sets of covariance matrices. We perform the following steps:

1. **Preprocessing:** We first perform pre-processing to remove corrupt data and replace fill-entries in each $\mathbf{A}_k$ corresponding to ozone levels which were not retrieved. (See Section 2.1)

2. **Transformation:** We use singular value decompositions (SVDs) to form bases for the column and row space of the $\{\mathbf{A}_k\}_{k=1}^{N}$. We use these bases to perform a common unitary transformation to each $\mathbf{A}_k$. (Sec. 2.2)

3. **Compression:** We use a variation of Rice-Golomb coding to compress each entry of a transformed $\mathbf{A}_k$ based on the statistics of the corresponding entries of all the transformed matrices. (Sec. 2.3)

2.1 Preprocessing

Since each sounding contains data for only a subset of the d pressure levels, it is necessary to reformat them before applying a transformation and compressing. The alternative would be to define different transformation and compression schemes for each subset of pressure levels, which is more complicated and requires extra storage overhead for the transformation and compression parameters.

For each index (i, j), with $1 \leq i, j \leq d$, we first identify the sounding indices k corresponding to non-fill entries $\mathbf{A}_k[i, j]$:

$$\mathcal{I}_{i,j} := \{k \ : \ \mathbf{A}_k[i, j] \neq v_{\text{fill}}\}. \tag{1}$$

We compute the mean of all the corresponding non-fill entries:

$$\overline{m}_{ij} := \frac{1}{|\mathcal{I}_{i,j}|} \sum_{k \in \mathcal{I}_{i,j}} \mathbf{A}_k[i, j]. \tag{2}$$

We then define the modified matrix $\tilde{\mathbf{A}}_k$ to have entries

$$\tilde{\mathbf{A}}_k[i, j] = \begin{cases} \mathbf{A}_k[i, j], & k \notin \mathcal{I}_{i,j} \\ \overline{m}_{ij}, & k \in \mathcal{I}_{i,j} \end{cases}. \tag{3}$$

In addition, we also track the indices of the d levels for which each sounding has data with a support vector $\mathbf{s}_k \in \{0, 1\}^d$, where $\mathbf{s}_k[\ell] = 1$ if and only if data exists for level ℓ in sounding k.

2.2 Transformation

Next, we construct a common transformation to be applied to all the preprocessed matrices $\tilde{\mathbf{A}}_k$. We begin by computing the SVDs

$$\begin{bmatrix} \tilde{\mathbf{A}}_1 & \dots & \tilde{\mathbf{A}}_N \end{bmatrix} = \mathbf{U}_1 \mathbf{\Sigma}_1 \mathbf{V}_1^H, \tag{4}$$

$$\begin{bmatrix} \tilde{\mathbf{A}}_1 \\ \vdots \\ \tilde{\mathbf{A}}_N \end{bmatrix} = \mathbf{U}_2 \mathbf{\Sigma}_2 \mathbf{V}_2^H. \tag{5}$$

The columns of $\mathbf{U}_1$ form a good representation of the column space of the $\tilde{\mathbf{A}}_k$ in the sense that most of the columns of each $\tilde{\mathbf{A}}_k$ can be approximated by a linear combination of a small number of columns of $\mathbf{U}_1$. Likewise, the rows of $\mathbf{V}_2^H$ are a good representation of the rows of the $\tilde{\mathbf{A}}_k$. When we apply the transformation

$$\tilde{\mathbf{A}}_k \mapsto \mathbf{B}_k := \mathbf{U}_1^H \tilde{\mathbf{A}}_k \mathbf{V}_2, \tag{6}$$

we find that each $\mathbf{B}_k$ is close to being a diagonal matrix, and that for most of the indices (i, j) there is little variation among the entries $\{\mathbf{B}_k[i, j]\}_{k=1}^N$.

Furthermore, the transformation in (6) has the desirable property that it is unitary: for two square matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ in $\mathbb{R}^{d\times d}$

$$\begin{aligned}\operatorname{vec}(\mathbf{U}_1^H\mathbf{M}_1\mathbf{V}_2)^H\cdot\operatorname{vec}(\mathbf{U}_1^H\mathbf{M}_2\mathbf{V}_2) &= \operatorname{Tr}\left(\mathbf{V}_2^H\mathbf{M}_1^H\mathbf{U}_1\mathbf{U}_1^H\mathbf{M}_2\mathbf{V}_2\right)\\ &= \operatorname{Tr}\left(\mathbf{M}_1^H\mathbf{M}_2\right)\\ &= \operatorname{vec}(\mathbf{M}_1)^H\cdot\operatorname{vec}(\mathbf{M}_2).\end{aligned}$$

These properties allow us to achieve a higher level of compression without introducing systematic error, as we will see in Section 2.3. In order to specify our transformation, we need only store the two matrices $\mathbf{U}_1$ and $\mathbf{V}_2$ in our final data product. The $\mathbf{B}_k$ matrices are now ready to be compressed.

2.2.1 Symmetric Matrices

In the case of the $\mathbf{A}_k$ being symmetric matrices, as is true for any of the aforementioned covariance data objects, we see that we may take $\mathbf{U}_1$ and $\mathbf{V}_2$ to be equal in Equations (4), (5), and subsequently (6). In this case, our transformation takes the form

$$\tilde{\mathbf{A}}_k \mapsto \mathbf{B}_k := \mathbf{U}_1^H\tilde{\mathbf{A}}_k\mathbf{U}_1, \tag{7}$$

and we need only store $\mathbf{U}_1$ in our final data product.

2.3 Compression

For each index (i, j), $1 \le i, j \le d$, we use a variation of Rice-Golomb compression applied to the elements

$$\mathcal{B}_{i,j} := \{\mathbf{B}_k[i,j]\}_{k=1}^N. \tag{8}$$

For each set $\mathcal{B}_{i,j}$, we compute the mean and the variance

$$\mu_{ij} = \frac{1}{N}\sum_{k=1}^N \mathbf{B}_k[i,j], \tag{9}$$

$$\sigma_{ij}^2 = \frac{1}{N}\sum_{k=1}^N (\mathbf{B}_k[i,j]-\mu_{ij})^2. \tag{10}$$

Then for each sounding k, we construct a concise bit-encoding of the differences

$$\delta_{ij}^{(k)} := \mathbf{B}_k[i,j]-\mu_{ij},\ 1\le i,j\le d. \tag{11}$$

We will be using lossy compression so that we may use our bit-encoding to construct an estimate $\hat{\delta}_{ij}^{(k)}$ for $\delta_{ij}^{(k)}$. We specify a maximum error parameter, $\Delta > 0$, defined such that $|\hat{\delta}_{ij}^{(k)} - \delta_{ij}^{(k)}| \le \Delta$. For each index (i, j), we select a quantization parameter $Q_{ij} > 0$, to be defined (see Sec. 2.3.2). Then for each $\delta_{ij}^{(k)}$, we compute

$$s_{ij}^{(k)} := \operatorname{sign}(\delta_{ij}^{(k)}), \tag{12}$$

$$q_{ij}^{(k)} := \left\lfloor \frac{|\delta_{ij}^{(k)}|}{Q_{ij}} \right\rfloor, \tag{13}$$

$$r_{ij}^{(k)} := \left\lfloor \frac{|\delta_{ij}^{(k)}| - q_{ij}^{(k)} Q_{ij}}{2\Delta} \right\rfloor. \tag{14}$$

Loosely speaking, we have divided the real line into intervals of length Q_{ij} and divided each such interval into subintervals of length 2Δ. The values of $s_{ij}^{(k)}$ and $q_{ij}^{(k)}$ specify which length-Q_{ij} interval contains $\delta_{ij}^{(k)}$, and the value of $r_{ij}^{(k)}$ designates in which of the $\frac{Q_{ij}}{2\Delta}$ subintervals it lies. In compressed form, we store a single bit representing the sign $s_{ij}^{(k)}$, followed by $q_{ij}^{(k)} + 1$ bits representing $q_{ij}^{(k)}$, consisting of $q_{ij}^{(k)}$ zeros followed by a terminating '1'. Finally, we append $\max\left(0, \left\lceil \log_2\left(\frac{Q_{ij}}{2\Delta}\right)\right\rceil\right)$ bits to encode the binary representation of $r_{ij}^{(k)}$. We concatenate all of these bits into a binary vector $\mathbf{b}_{ij}^{(k)} \in \{0,1\}^{q_{ij}^{(k)}+2+\max(0,\lceil \log_2(Q_{ij}/2\Delta)\rceil)}$.

Given $\mathbf{b}_{ij}^{(k)}$ along with Q_{ij} and Δ, we can exactly decode $s_{ij}^{(k)}$, $q_{ij}^{(k)}$ and $r_{ij}^{(k)}$, from which we construct our estimate as

$$\hat{\delta}_{ij}^{(k)} = s_{ij}^{(k)} \cdot \left(q_{ij}^{(k)} Q_{ij} + 2 r_{ij}^{(k)} \Delta + \Delta \right). \tag{15}$$

Thus, $\hat{\delta}_{ij}^{(k)}$ is chosen to be the center of the length-2Δ subinterval containing $\delta_{ij}^{(k)}$, so our quantization error is at most Δ by construction.

2.3.1 Avoiding Systematic Error

An important observation is that this compression method avoids introducing systematic error in the reconstructed matrices after decompressing and inverting the transformation described in Section 2.2. There are two reasons for this: 1) The quantization error for each entry, $\hat{\delta}_{ij}^{(k)} - \delta_{ij}^{(k)}$, is approximately uniformly distributed over the interval $[-\Delta, \Delta]$ and independent for all i and j, and 2) the transformation described in Section 2.2 is unitary. The quantization error for sounding k can be represented by a random matrix $\mathbf{E}_k$ whose $(i,j)^{th}$ entry is equal to $\hat{\delta}_{ij}^{(k)} - \delta_{ij}^{(k)}$. After inverting the transformation, this will map to an error pattern given by $\mathbf{U}_1 \mathbf{E}_k \mathbf{V}_2^H$. Since the transformation is unitary, we can describe the inverse transformation with a unitary matrix $\mathbf{W} \in \mathbb{R}^{d^2 \times d^2}$ applied to the vectorization of $\mathbf{E}_k$. Explicitly, we can express this as $\mathbf{W} = \mathbf{V}_2 \otimes \mathbf{U}_1$.

Since the entries of $\text{vec}(\mathbf{E}_k)$ are iid uniform on $[-\Delta, \Delta]$, we have

$$\begin{aligned} E(\mathbf{W} \cdot \text{vec}(\mathbf{E}_k)) &= \mathbf{0}, && (16) \\ \text{Cov}(\mathbf{W} \cdot \text{vec}(\mathbf{E}_k)) &= E(\mathbf{W} \cdot \text{vec}(\mathbf{E}_k)\text{vec}(\mathbf{E}_k)^H \mathbf{W}^H) && (17) \\ &= \mathbf{W} \cdot E(\text{vec}(\mathbf{E}_k)\text{vec}(\mathbf{E}_k)^H) \cdot \mathbf{W}^H && (18) \\ &= \mathbf{W} \cdot (\sigma^2 \mathbf{I}_{d^2}) \cdot \mathbf{W}^H && (19) \\ &= \sigma^2 \mathbf{I}_{d^2}, && (20) \end{aligned}$$

where $\mathbf{I}_{d^2}$ is the $d^2 \times d^2$ identity matrix, and σ^2 is the variance of each entry of $\mathbf{E}_k$, which for the uniform distribution is $\sigma^2 = \frac{\Delta^2}{3}$. Thus, the entries of the reconstructed matrix have uncorrelated, identically distributed, zero-mean error.

2.3.2 Selecting the Quantization Parameters

We return now to the question of how to choose the quantization parameters Q_{ij}. In general, the Q_{ij} can be chosen on a case-by-case basis to optimize our compression ratio. Our final product stores the Q_{ij}, so our decompression algorithm allows for its specification by a user. But in practice, we have designed a method to select Q_{ij} in an automated fashion.

We approximate the $\delta_{ij}^{(k)}$ as being normally distributed,

$$\delta_{ij}^{(k)} \sim \mathcal{N}(0, \sigma_{ij}^2). \tag{21}$$

Let $c_{ij}(x)$ denote the cumulative distribution function of $|\delta_{ij}^{(k)}|$, the absolute value of the normally-distributed variable. Its derivative is then the associated probability density function

$$c'_{ij}(x) = \frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot e^{-\frac{x^2}{2\sigma_{ij}^2}}. \tag{22}$$

The expected number of bits needed to encode $\delta_{ij}^{(k)}$, for given Δ and Q_{ij}, is then

$$E(\text{length}(\mathbf{b}_{ij}^{(k)})) = 2 + f_1(Q_{ij}) + f_2(Q_{ij}), \tag{23}$$

where

$$f_1(Q_{ij}) := \sum_{q=1}^{\infty} q \cdot [c_{ij}(qQ_{ij}) - c_{ij}((q-1)Q_{ij})] \tag{24}$$

is the expected length of the bit-encoding of $q_{ij}^{(k)}$, and

$$f_2(Q_{ij}) := \max\left(0, \left\lceil \log_2\left(\frac{Q_{ij}}{2\Delta}\right)\right\rceil\right) \tag{25}$$

is the length of the bit-encoding of $r_{ij}^{(k)}$. We wish to minimize Equation (23) with respect to Q_{ij}. Toward this end, we define the relaxation of $f_2(Q_{ij})$,

$$f_3(Q_{ij}) := \log_2\left(\frac{Q_{ij}}{2\Delta}\right) \tag{26}$$

$$= \frac{1}{\ln 2}\left(\ln(Q_{ij}) - \ln(2\Delta)\right), \tag{27}$$

and we consider the relaxed optimization

$$\text{minimize}_{Q_{ij}}\ f_1(Q_{ij}) + f_3(Q_{ij}). \tag{28}$$

Examining the derivative of $f_1(Q_{ij})$,

$$\frac{df_1}{dQ_{ij}} = \sum_{q=1}^{\infty} q \cdot \left[q \cdot c'_{ij}(qQ_{ij}) - (q-1)\cdot c'_{ij}((q-1)Q_{ij})\right] \tag{29}$$

$$= \frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \sum_{q=1}^{\infty}\left[q^2 \cdot e^{-\frac{q^2 Q_{ij}^2}{2\sigma_{ij}^2}} - q(q-1)\cdot e^{-\frac{(q-1)^2 Q_{ij}^2}{2\sigma_{ij}^2}}\right] \tag{30}$$

$$= \frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \sum_{q=1}^{\infty}\left[q^2 \cdot w(Q_{ij})^{q^2} - q(q-1)\cdot w(Q_{ij})^{(q-1)^2}\right] \tag{31}$$

$$= \frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \left[\sum_{q=1}^{\infty} q^2 \cdot w(Q_{ij})^{q^2} - \sum_{\tilde{q}=1}^{\infty}(\tilde{q}+1)\tilde{q}\cdot w(Q_{ij})^{\tilde{q}^2}\right] \tag{32}$$

$$= \frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \left[\sum_{q=1}^{\infty}(q^2 - (q+1)q)\cdot w(Q_{ij})^{q^2}\right] \tag{33}$$

$$= -\frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \sum_{q=0}^{\infty} q \cdot w(Q_{ij})^{q^2} \tag{34}$$

$$= -\frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \sum_{q=0}^{\infty}\frac{d}{dq}\left(\frac{w(Q_{ij})^{q^2}}{2\ln w(Q_{ij})}\right) \tag{35}$$

where in Equation (31) we have substituted $w(Q_{ij}) := \exp\left(-\frac{Q_{ij}^2}{2\sigma_{ij}}\right)$, and in Equation (32) we have defined $\tilde{q} := q-1$ and dropped the leading trivial term of the second summation. Notice from Equation (34) that df_1/dQ_{ij} is strictly negative, so $f_1(Q_{ij})$ is decreasing in Q_{ij}. We can approximate the final sum as the integral

$$\frac{df_1}{dQ_{ij}} \approx -\frac{2}{\sigma_{ij}\sqrt{2\pi}} \cdot \int_{q=0}^{\infty}\frac{d}{dq}\left(\frac{w(Q_{ij})^{q^2}}{2\ln w(Q_{ij})}\right) dq \tag{36}$$

$$= \frac{2}{\sigma_{ij}\sqrt{2\pi}\ln w(Q_{ij})} \tag{37}$$

$$= -\frac{2\sigma_{ij}}{\sqrt{2\pi}Q_{ij}^2}. \tag{38}$$

The derivative of $f_3(Q_{ij})$ with respect to Q_{ij} is simple to compute:

$$\frac{df_3}{dQ_{ij}} = \frac{1}{Q_{ij}\ln 2}. \tag{39}$$

Setting $\frac{df_1}{dQ_{ij}} + \frac{df_3}{dQ_{ij}} = 0$, and solving for Q_{ij}, we find that

$$\hat{Q}_{ij} := \arg\min_{Q_{ij}} \;(f_1(Q_{ij}) + f_3(Q_{ij})) = \frac{2\sigma_{ij}\ln 2}{\sqrt{2\pi}}. \tag{40}$$

If $\hat{Q}_{ij} \geq 2\Delta$, we can see that setting $Q_{ij} = \hat{Q}_{ij}$ will minimize the expected length of $\mathbf{b}_{ij}^{(k)}$. Otherwise, we have $f_2(\hat{Q}_{ij}) = 0$. In this case, since $f_1(Q_{ij})$ is decreasing in Q_{ij} as previously noted, the minimum expected length is achieved by increasing Q_{ij} to 2Δ. Thus, for a given maximum error Δ, we opt to set

$$Q_{ij} = \max\left(\frac{2\sigma_{ij}\ln 2}{\sqrt{2\pi}},\ 2\Delta\right). \tag{41}$$

3 Final Data Product

We organize our final compressed data product in such a way that a user-specified subset of the soundings can be quickly decompressed independently from all other soundings. Our final data product is in the form of a byte array, with initial bytes containing information needed to reconstruct all soundings, followed by subsequent packets of bytes containing data specific to each individual sounding. Thus, given a user-specified subset of soundings to decompress, our decompression algorithm only reads in the initial bytes (common to all soundings), and the byte packets corresponding to the requested soundings.

The initial bytes of common information to all soundings, in order, consist of

1. A single byte encoding the compression mode (‘1’ for asymmetric matrices such as averaging kernels, ‘2’ for symmetric matrices such as covariances),
2. 8 bytes encoding the subinterval length 2Δ, stored with double precision,
3. $8d^2$ bytes for each of the entries of the left transformation matrix $\mathbf{U}_1$, stored with double precision in row-major order,
4. $8d^2$ bytes for each of the entries of the right transformation matrix $\mathbf{V}_2$, stored with double precision in row-major order,
    - These bytes are omitted in the case of symmetric matrices, as commented in Section 2.2.1.
5. $8d^2$ bytes for the entries Q_{ij}, stored with double precision in row-major order when considered as the matrix $[Q_{ij}]$,

6. $8d^2$ bytes for the number of length-2Δ subintervals for each index, specifically $\left\lceil \frac{Q_{ij}}{2\Delta} \right\rceil$, each stored (in row-major order) as an unsigned long long integer,

7. $8N$ bytes for the N locations of the individual sounding-specific data packets, each stored as an unsigned long long integer index representing the starting position of the sounding's data packet in this array of bytes.

Following this preamble, we sequentially add the data packets which are specific to each of the N soundings. Note that within our final compressed object, the starting byte position of each packet is referenced in item 7 above, allowing for our decompressor to quickly retrieve it in the event that a user wants only the information for a single sounding. Each packet consists of a sequence of bits, followed by between 1 and 7 zero-bits such that the total number of bits is a multiple of 8. The bit string is partitioned into bytes, interpreted with big-endian format. The bit string itself, as described before, consists of

1. A single bit to encode the sign $s_{ij}^{(k)}$ (a '0' if $s_{ij}^{(k)} < 0$ and a '1' otherwise),

2. $q_{ij}^{(k)} + 1$ bits to encode the value of $q_{ij}^{(k)}$, in the form of $q_{ij}^{(k)}$ '0's followed by a terminating '1,'

3. $\max\left(0, \left\lceil \log_2\left(\frac{Q_{ij}}{2\Delta}\right) \right\rceil\right)$ bits to encode the binary representation of $r_{ij}^{(k)}$. The most significant bit in the binary representation is first, and the least significant is last. For example, if $\left\lceil \log_2\left(\frac{Q_{ij}}{2\Delta}\right) \right\rceil = 8$, the number 13 would be encoded as '00001101.'

Each of these sets of $1 + (q_{ij}^{(k)} + 1) + \max\left(0, \left\lceil \log_2\left(\frac{Q_{ij}}{2\Delta}\right) \right\rceil\right)$ bits is stored in sequence for the (i, j) entries of the k^{th} sounding, in row-major order of (i, j) when interpreted as the indices of a matrix. As mentioned above, the sequence is then padded with 1-7 zero-bits to form the complete packet of bytes for the k^{th} sounding. The byte packets for each sounding are then stored in order for $k = 1, 2, ..., N$.

4 Using the Software

Our software is written in Python, and the main library is `TROPESS_compression_v2.py`. To run the compression and decompression algorithms,

1. Install Python 3 as "python3."

2. Make sure virtualenv is installed. If not, install it using pip with the command `pip install virtualenv`.

3. Set up a Python virtual environment:

```
virtualenv -p python3 /sample/path/environmentname.
```

4. Install the required Python libraries using our provided “requirements.txt” file: `pip install -r requirements.txt`.

To compress a large data object, such as the set of averaging kernels, they must be stored as a numpy 3D array, with dimensions $N \times d \times d$. For example, in a Python terminal,

```
import h5py
import numpy as np
from TROPESS_compression_v2 import Multiple_Sounding_Compression

data_file = h5py.File('path/to/data/AIRS_OMI_ATrain_L2-O3_2016_04_02_F01_01.hdf',
'r')
AK_dataset = data_file['HDFEOS/SWATHS/O3NadirSwath/Data Fields/AveragingKernel']
AK_nparray = np.array(AK_dataset)
data_file.close()

MSC = Multiple_Sounding_Compression(AK_nparray)
compressed_bytes = MSC.compress_3D()
```

By default, the compression will treat the matrices as asymmetric (“compression mode 1”), and uses a subinterval half-width of $\Delta = 0.00005$ which has been found to introduce negligible error to all large data objects in testing. For symmetric matrices, we can use “compression mode 2,” achieving roughly twice as much compression, by altering the final line to

```
compressed_bytes = MSC.compress_3D(compression_mode=2).
```

The value of Δ can be changed using the input parameter `max_error`. For instance, to double the error threshold, we may use the line

```
compressed_bytes = MSC.compress_3D(max_error=0.0001)
```

or

```
compressed_bytes = MSC.compress_3D(compression_mode=2, max_error=0.0001).
```

We emphasize again that the parameter Δ represents the maximum entry-wise error in the *transformed* data space. Since our transformation is unitary, the maximum error on the final reconstructed data will be close to Δ, but not identical.

`compressed_bytes` will be a Python bytearray object which can be stored directly in an hdf5 or netcdf file. The object can be decompressed using the following code:

```
from TROPESS_compression_v2 import Multiple_Sounding_Decompression
MSD = Multiple_Sounding_Decompression(compressed_bytes)
```

```
AK_nparray_decompressed = MSD.decompress_3D()
```

This will by default decompress all N soundings. We warn users that this can quickly accumulate large volumes of memory. A small subset of soundings to be decompressed can be specified with the `sounding_inds` input parameter by creating a list of the desired indices. Note that Python uses zero-indexing. For instance, if soundings 1000 through 1999 are desired, we can specify this with the lines

```
index_list = [i for i in range(1000, 2000)]
AK_nparray_decompressed = MSD.decompress_3D(sounding_inds=index_list)
```

Similarly, soundings 0-99 can be specified by setting `index_list = [i for i in range(100)]`, and so on.

References

[1] S. Migliorini, C. Piccolo, and C. D. Rogers, "Use of the Information Content in Satellite Measurements for an Efficient Interface to Data Assimilation," *Monthly Weather Review*, **136** (7): 2633-2650, 2008.

[2] A. P. Mizzi, A. F. Arellano Jr., D. P. Edwards, J. L. Anderson, and G. G. Pfister, "Assimilating compact phase space retrievals of atmospheric composition with WRF-Chem/DART: a regional chemical transport/ensemble Kalman filter data assimilation system," Geoscientific Model Development, 9, 965-978, 2016.