The Poisson-Influenced K-Means Algorithm

A Maximum-Likelihood Procedure for Clusters with a Known Probability Distribution

Brian P. M. Morris
Zachary H. Levine

We present an implementation of the Poisson-Influenced K-Means Algorithm (PIKA), first developed to characterize the output of a superconducting transition edge sensor (TES) in the few-photon-counting regime. The algorithm seeks to group data into several clusters that minimize their distances from their means, as in classical K-means clustering, but with the added knowledge that the cluster sizes should follow a Poisson distribution.

1. Run PIKA Here

The algorithm proper is run when it is submitted using the button near the lower-right corner of the form. You also have the option to use a separate input file to manually override the form, which may be more useful for automated runs on multiple datasets. This function launches the program; evaluating it generates the form and then executes PIKA. The function `pika[]` is defined and documented in Section 7.9. After taking input through the form, `pika` calls `runPIKA`, the *de facto* main function for the program, defined in Section 3. You can also specify a separate options file in the form that overrides the variable assignments that the form makes. The first command ensures that the paclet for forms is current.

```
PacletUpdate["Forms"];
pika[]
```

If you get a popup that asks, Do you want to automatically evaluate all the initialization cells…?, answer yes. When you get the form, it is necessary to replace [directory] with the pathname of the location of the data; you might also have to change the backslash to a forward slash.

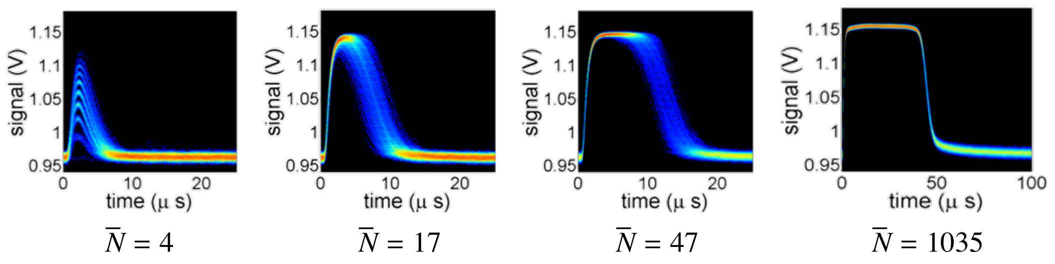
2. General Formulation (No Code)

The Poisson-Influenced K-means Algorithm (PIKA) was first described in [1] as a way of calibrating a transition edge sensor (TES), a superconducting few-photon detector. A TES can discern the number of photons in a very weak pulse of light, but it must be calibrated in order to do so. Our implementation deals with photon counting, but many of its features are applicable to more general probability-assisted K-means clustering situations.

2.1. Background

A TES is a superconductor kept in its transition from its superconducting phase to the normal regime, where it loses its superconducting properties. Photons incident on the sensor heat it, causing its resistance to rise sharply and then slowly fall to superconducting levels as the heat dissipates. A current is run through the TES, and the change in resistance is captured by the voltage signal of a superconducting quantum interference device (SQUID) inductively coupled to the TES circuit.

Several groups of TES signal waveforms are shown in Figure 1 (each graph shows the set of signals elicited by an ensemble of laser pulses with an average number of photons per pulse given by $\overline{N}$). For $\overline{N} = 4$, one can clearly distinguish the different photon numbers and their relative frequencies; for higher numbers this is harder. Higher photon numbers create higher signal amplitudes, but at a certain point the TES saturates in the normal regime and additional photons change the signal maximum very little.



▲ **Figure 1.** Several collections of TES waveforms resulting from pulses with particular mean photon numbers given by $\overline{N}$, from [2].

The goal of PIKA is to characterize individual TES waveforms by the integer photon numbers of the pulses that cause them. The photon numbers of individual pulses cannot be determined directly; we can only estimate the average photon number of all of the pulses, based on the nominal laser and attenuator parameters of the light source.

K-means clustering, upon which PIKA is based, is a fundamental part of unsupervised machine learning. PIKA extends the K-means algorithm to scenarios in which the ideal distribution that the clusters should follow is known, and though some of the implementation is specific to the context of TES calibration (e.g. the use of the Poisson distribution, the idea of ordering observations by photon number), much of it can be generalized without much difficulty to other situations with known probability distributions.

◻ 2.2. *K*-means Clustering and the Poisson Distribution

Traditional *K*-means clustering consists of taking some amount of data and organizing it into clusters that minimize their members' distance from the cluster mean. Essentially this is a minimization of an objective function, the sum over each piece of data of its deviation from its cluster mean (where deviation is measured by some relevant definition of distance). We can use a similar approach by considering each signal as a high-dimensional vector and its deviation from some mean as squared Euclidean distance. Then the *K*-means component of the objective function becomes

$$O_K = \sum_{n=n_0}^{n_0+K-1} \sum_{i\in C_n} \frac{1}{N_t} \sum_t \left[V_i(t) - \overline{V}_n(t)\right]^2, \tag{1}$$

where $\mathbf{V}_i$ is the signal vector for observation i ($V_i(t)$ is the t^{th} element of the vector $\mathbf{V}_i$), $\overline{\mathbf{V}}_n$ is the mean of the cluster C_n, and N_t is the number of time points; n_0 and K give the first cluster's photon number and number of clusters, respectively, and are determined by which photon numbers we expect to be associated with at least one pulse based on the Poisson distribution. More physically, $\mathbf{V}_i$ is an individual waveform and $\overline{\mathbf{V}}_n$ is the average of the waveforms with an assigned photon number n.

To account for the Poisson-distributed cluster sizes, we introduce another term, $O_{PC} = -\ln \mathcal{L}$, where $\mathcal{L}$ is the likelihood, according to the Poisson distribution, of a group of clusters associated with a group of photon numbers having the particular sizes that a given clustering asserts that they do, given the mean photon number of the ensemble of pulses. The likelihood of a particular sequence of photon numbers occurring in an ensemble with mean μ is

$$\mathcal{L}_P = \prod_{n=n_0}^{n_0+K-1} \left(\frac{e^{-\mu}\,\mu^n}{n!}\right)^{m_n}, \tag{2}$$

where m_n is the number of waveforms in cluster n. We also need a combinatorial component, since different photon-number sequences can yield the same eventual cluster sizes:

$$\mathcal{L}_C = \frac{M!}{\prod_{n=n_0}^{n_0+K-1} m_n!}, \tag{3}$$

where $M = \sum m_n$. Then $\mathcal{L} = \mathcal{L}_P\,\mathcal{L}_C$, and the PIKA objective function is

$$O_{KPC} = \frac{1}{2\,\sigma^2}\,O_K + O_{PC}. \tag{4}$$

The constant σ relating the two terms can be estimated from the data, since the objective function is itself the negative log-likelihood of a normal distribution. (This also means that minimizing O_{KPC} is equivalent to maximizing the product of two likelihoods.) PIKA minimizes the objective function by moving waveforms to neighboring clusters.

Once the clusters are optimized, each waveform is assigned an effective photon number by a linear interpolation between the two closest cluster means. First, we find the value α_i

that minimizes the root mean square deviation of $\mathbf{V}_i$ from $(1-\alpha_i)\,\overline{\mathbf{V}}_n + \alpha_i\,\overline{\mathbf{V}}_{n'}$, where $\overline{\mathbf{V}}_n$ and $\overline{\mathbf{V}}_{n'}$ are the closest and second-closest mean waveforms to $\mathbf{V}_i$. In practice, $i \in C_n$ and $n' = n \pm 1$. One can easily show that

$$\alpha_i = \frac{\left(\overline{\mathbf{V}}_n - \mathbf{V}_i\right)\cdot\left(\overline{\mathbf{V}}_n - \overline{\mathbf{V}}_{n'}\right)}{\|\,\overline{\mathbf{V}}_n - \overline{\mathbf{V}}_{n'}\,\|^2} \tag{5}$$

for each $\mathbf{V}_i$. The effective photon number is then given by

$$n_i^{(\mathrm{eff})} = (1-\alpha_i)\,n + \alpha_i\,n'. \tag{6}$$

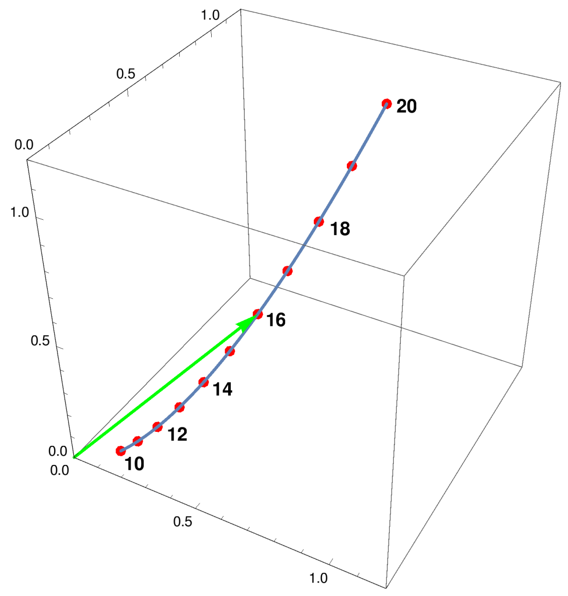
2.3. Initial Clustering

PIKA needs an initial clustering upon which to improve. Random cluster assignment is an option, but a better alternative is to give the observations a rough order by photon number, so that our initial guess is actually a meaningful estimate. This is done via the dot product method: we assign each observation an initial effective photon number

$$n_i^{(\mathrm{eff})} = \overline{N}\,\frac{\overline{\mathbf{V}}\cdot\mathbf{V}_i}{\|\,\overline{\mathbf{V}}\,\|^2}, \tag{7}$$

where $\overline{\mathbf{V}}$ is the entire ensemble's mean, not a cluster mean $\overline{\mathbf{V}}_n$. The initial clusters are sized to fit each observation and conform to the Poisson distribution, and the observations are placed in the clusters by order of effective photon number.

The geometric interpretation of PIKA and the dot product method is a curve and a line, respectively, evolving through hyperspace (shown in Figure 2). The dot product method projects each observation onto the mean waveform vector (a line) and then assumes that photon number scales linearly with distance along the mean vector (which is not actually true, but suffices for a first guess); that converts distance relative to the mean to photon number relative to the mean. PIKA, in contrast, finds a piecewise linear approximation of a curve that passes through the cluster means and projects each observation onto that. Both essentially measure photon number by progress along a one-dimensional path through high-dimensional space.



▲ **Figure 2.** An illustration of the geometric differences between the dot product method (green arrow) and PIKA (blue curve). The 3D space here stands in for a high-dimensional space.

2.4. Choice of Mean Photon Number

PIKA requires knowledge of the ensemble's mean photon number in addition to an initial clustering. There are two ways of supplying that knowledge. The first is simply to give the exact mean photon number of the incoming pulses, if it is known; then PIKA clusters the data accordingly.

The second is to test several mean photon numbers on the data if the true mean is not known exactly. The test means should be close together in some range around a rough estimate of the true mean; PIKA clusters the data once for each value, returning a new (usually better) estimate of the mean based on each optimized clustering, as well as the value of the objective function associated with each new estimate. (In addition, since the test means are close together so that adjacent distributions should be very similar, the effective photon numbers for the waveforms from one round are used as the initial ordering for the next, instead of the dot product method.)

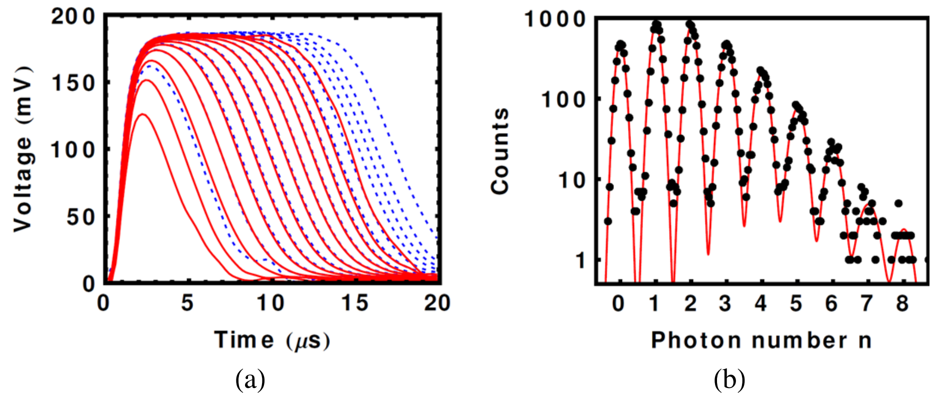
The optimized means are usually closer to the true mean than their initial seeds, but, depending on the structure of the probability distribution underlying the objective function, some optimized means may be moved farther away due to attraction to local minima. In the Poisson case, we have observed the objective function to have secondary minima at integer differences in mean photon number from the primary minimum and convex regions of width 1 centered around each minimum. Thus, test means that land in the same region as the true mean should be optimized toward the true mean, but those outside are diverted by secondary minima. Reference [3] estimates the range of test means to within half of a photon number from the nominal laser power and attenuation.

▫ 2.5. PIKA's Results

The following results and images (in Section 2.5) except Figure 6 come from [1].

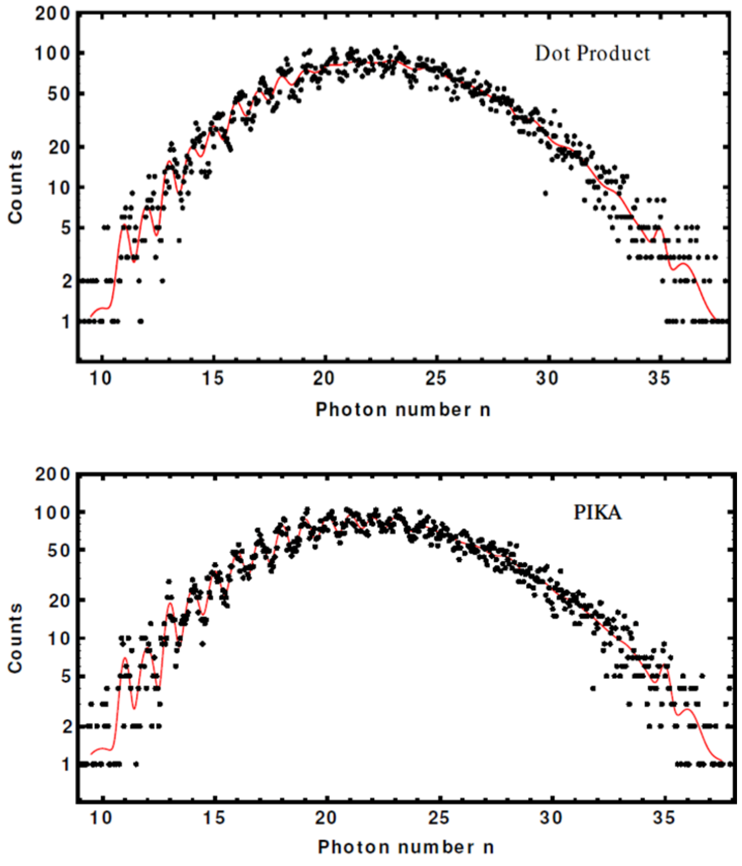
Figure 3(a) shows the optimized cluster means from two PIKA runs ($\overline{N} = 22.6$, solid red, and $\overline{N} = 31.6$, dotted blue). The shapes of the mean waveforms appear independent of $\overline{N}$—reassuring, since the average photon number of an ensemble should not affect the shape of individual waveforms. This independence indicates that PIKA is properly identifying actual photon numbers in the data, and that the average photon number with which it is supplied is not unduly affecting the results.

Figure 3(b) is a histogram of the optimized effective photon numbers from $\overline{N} = 2.00$ (bin widths of 0.05), which follow a Poisson distribution but with some Gaussian spread around the integers (the red curves are Gaussians centered on the integers and fitted to the data), resulting in a comb-like shape.

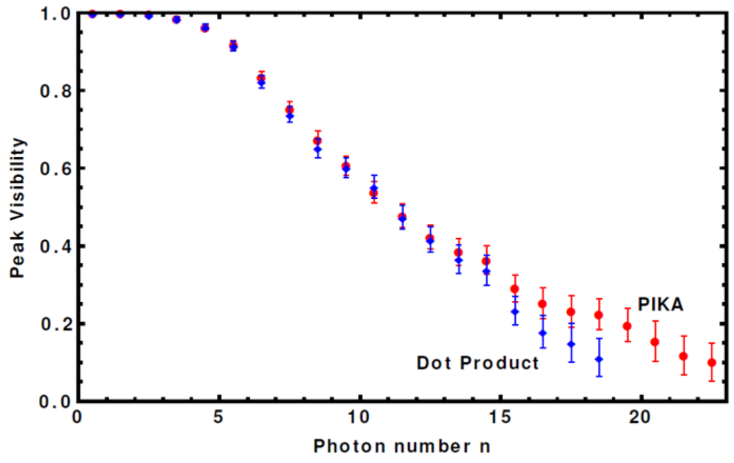


▲ **Figure 3.** (a) Optimized cluster mean waveforms for $\overline{N} = 22.6$ (solid red) and $\overline{N} = 31.6$ (dotted blue). (b) Optimized effective photon numbers for $\overline{N} = 2.00$ (black) and integer-centered Gaussians fitted to the data (red).

Figure 4 shows a similar comb structure for $\overline{N} = 22.6$ for both the dot product method and PIKA. As the photon number n increases, the teeth of the comb grow less defined—that is, the peak visibility (max − min) / (max + min) falls, and with it the photon-resolving capability. Figure 5 shows this drop in visibility. The power of PIKA is that it retains nonzero visibility (i.e. the uncertainty does not include 0) through $n = 23$, whereas the dot product method alone loses it after $n = 19$. PIKA has been used to explore the regime just beyond the loss of peak visibility [4].

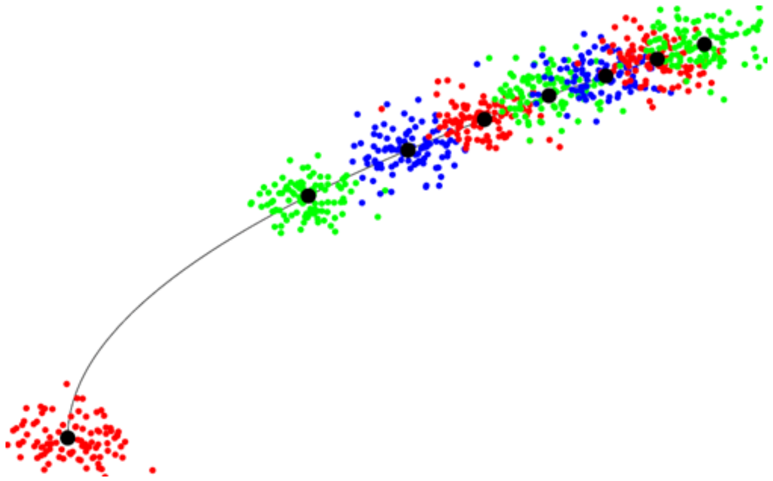


▲ **Figure 4.** Effective photon numbers from the dot product method and from PIKA for $\overline{N} = 22.6$.



▲ **Figure 5.** Peak visibility for the dot product method (blue) and PIKA (red) through $n = 23$ (both lose visibility altogether after that point). Uncertainties are given at two standard deviations and are purely statistical.

Figure 6 illustrates the loss of visibility with high photon numbers. As n increases, cluster means become closer to each other (saturation in the normal regime) and individual waveforms overlap with each other more and more, making it more difficult to differentiate between adjacent photon numbers.



▲ **Figure 6.** Illustration of the loss of visibility as n increases. Cluster means become closer together, so individual waveforms overlap more and more.

3. Terminology and Main Loop (`runPika`)

The code and documentation are written in the context of few-photon-counting with a TES, so variables and functions are often named for physical quantities and concepts specific to the original purpose of the algorithm instead of more abstract ones (e.g. `readTES` instead of `readData`). In addition, the terms “trace,” “observation,” and “waveform” are used interchangeably and synonymously to mean a single detector response in the set of signals—that is, a regular sampling of voltage over time that records the response to a single optical pulse fired at the TES. In the program a trace is a list of voltage values.

Variable and function names also follow certain conventions:

- A variable beginning with `i` or `j` is an index (or list of indices) on the remainder of its name.
  - `iObs` is single index or list of indices on a set of observations, for example.
- A variable beginning with `n` is a number or size of the remainder of its name (or a list of numbers or sizes).
- A variable beginning with `m` is a maximum number of something.
  - `mSample` gives the maximum allowed number of waveforms to sample for graphical output, but fewer are taken if the population of waveforms is smaller than `mSample`.

- A variable of the form `xOfY` lists `x` for each `Y`.
  - `iObsOfClust` gives a list of indices `"iObs"` corresponding to `"Of"` each cluster `"Clust"`.
  - `nClust` gives the number of clusters, but `nOfClust` lists the size of each cluster.
- A function beginning with `get` returns something described by the rest of its name.
  - `getIObsOfClust` returns `iObsOfClust`.
- A variable such as `p1` is a plot.

This section describes the overarching form of the algorithm. Here, the process is broken into a series of more isolated procedures, whose implementation details are described in the next section. Note that the functions referenced in this section and defined in the next take no arguments and return no output—they merely break the whole algorithm into smaller pieces of code that operate on global variables.

Below is the skeleton of the process. We begin by defining constants and options (which can be overridden as necessary by the user), and then reading in the data with a noise filter. One can optionally filter out observations contaminated by background radiation; by default the algorithm does not do this. We then proceed to iterate (Loop 1) over the elements of `nPhotonAvgList`, a list of hypothetical mean photon numbers for the set of pulses incident on the sensor, running PIKA once on the dataset for each element. This lets us compare the optimized objective functions for each hypothesized mean and estimate the actual mean from the runs with the smallest objective functions. The dot-product method supplies the initial effective photon numbers (which create the initial clustering) for the first mean photon number; after that we use the optimized effective photon numbers from the previous mean on the list as our initial guess for the next mean (adjacent elements of `nPhotonAvgList` should be close together, so one round's optimized ordering of the observations should be a reasonable seed for the next).

In Loop 1 (over `nPhotonAvgList`), we organize the traces by their clusters and then find the initial value of the objective function to be minimized. We then enter the actual optimization loop (Loop 2), which repeats some preset number of times, trying to lower the objective function by moving observations between clusters. For each iteration of Loop 2, the nested Loop 3 examines each observation and considers moving it to a neighboring cluster (one with a photon number one greater or less). The greedy algorithm [5] (the default option) accepts any move that reduces the objective function and rejects any that does not, while the simulated annealing algorithm [6] may accept disadvantageous moves so that it can find global minima instead of just local ones. If a move is accepted, the relevant pieces of the objective function are updated. When Loop 2 finishes, the observations have been organized into optimized clusters, and we create various graphical and textual outputs.

```
runPIKA[] := (
  initialSetup[];
  readAndFilterData[];
  optionallyRejectBackgroundTraces[];
  graphGlobalMeanTrace[];

  findDotProductEffectivePhotonNumbers[];
  Do[      (* Loop 1 *)
   createInitialClusters[];
   graphSampleMiddleClusters[];
   graphClusterMeans[];
   findInitialObjectiveFunction[];
   graphClusterDeviationsFromMean[];
   prepareOptimizationLoop[];
   Do[      (* Loop 2 *)
    prepareSubLoop[];
    Do[     (* Loop 3 *)
     getClusterAndNeighbors[];
     updateRelevantDeviations[];
     findChangeInObjectiveFunction[];
     decideWhetherToMove[];
     updateIfMoving[];
     , {jObs, nObs}
    ]; (* End Loop 3 *)
    updateAnnealingTemperature[];
    , {iCool, nCool}
   ]; (* End Loop 2 *)
   prepareAnalysisAndOutput[];
   graphNewSampleMiddleClusters[];
   graphNewClusterMeans[];
   graphProbabilityDistribution[];
   graphNewClusterToMeanDeviations[];
   graphNewMeanToMeanDeviations[];
   graphNewEffectivePhotonNumber[];
   , {iNPhotonAvgList, Length[nPhotonAvgList]}
  ]; (* End Loop 1 *)
  showOutput[]
 )
```

4. Implementation

Here we define the functions referenced in the previous section. The structure of the following subsections mirrors that of the algorithm skeleton above: Subsection 1 defines the procedures called before Loop 1 begins, Subsections 2 through 4 define those used in Loop 1 (with Subsection 3 containing the functions for Loops 2 and 3), and Subsection 5 contains the (very minimal) end of the algorithm after exiting Loop 1. The functions are defined in the order in which they are called, and each is called exactly once. The functions called inside the bodies of these skeleton functions (i.e. those that take arguments, unlike the functions defined in this section) are defined in Sections 5 and 7.

4.1. Set Options and Read Data

The main execution function, `pika`, assigns values to the following variables based on the user's response to the input form:

- `iNPhoton`—the index of the dataset to read (assuming that multiple datasets are stored in the same location)
- `iDataSet`—a list of indices of parts of the dataset to read (assuming that datasets are split over multiple files)
- `nTime`—the number of time points to keep after filtration
- `mSample`—the maximum number of traces to randomly sample when outputting samples
- `nDatUse`—the maximum number of traces to read in (if this is equal to 0, we use all of the traces)
- `backgroundReject`—a Boolean, true if PIKA should reject response waveforms with background radiation
- `peakValCut`—the voltage cutoff for background rejection
- `peakPosCut`—the peak location cutoff for background rejection
- `peakNumCut`—the cutoff for number of peaks for background rejection
- `nCool`—the number of optimizing rounds to perform
- `nGreedy`—the number of rounds that should be run with the greedy algorithm
- `coolConst`—the cooling constant for simulated annealing
- `tAnneal`—the starting annealing temperature
- `probDistName`—the name of the probability distribution
- `nSigma`—the number of standard deviations from the mean of the probability distribution to generate
- `binFract`—the histogram bin size (fraction of a photon number)
- `outputImageSize`—the default graphics size for output

- `nPhotonAvgList`—a list of (close together) mean photon numbers with which to run PIKA on the dataset
- `partialFilePath`—the directory and the beginning of the name of the data files, without the `iNPhoton` or `iDataSet` markers
- `fileExt`—the file extension of the data files (without the `iDataSet` marker)
- `nSamplePerTrace`—the number of samples per trace
- `nTracePerFile`—the number of traces per file
- `useInputFile`—a Boolean, true if there is another options file to read in and override settings from the form
- `pikaInput`—the path to the options override file

`runPIKA` (defined in Section 3 with implementation in this section) begins with some initial setup, processing some of the input given in `pika` and setting up tables to store graphical output.

```
initialSetup[] := (

  SetOptions[$Output, PageWidth → 110];
  SetOptions[ListPlot, Joined → True, Frame → True,
   PlotRange → All];
  SetOptions[ListLogPlot, Joined → True, Frame → True,
   PlotRange → All];
  tStart = TimeUsed[];

  If[useInputFile, Get[pikaInput]];
  (* user defined override *)
  tAnnealOrig = tAnneal;

  Print["Options read; now running"];

  fileInfo = {{partialFilePath, fileExt}, nSamplePerTrace,
    nTracePerFile};

  outputCreateTabs[Length[nPhotonAvgList]];
  print[0, "dataset number (iNPhoton): ", iNPhoton];
  print[0, "file range (iDataSet): ", iDataSet];

 )
```

Now we read in the data and apply a noise filter. `readTESandFilter`, defined in Section 7.7, takes the data from the set of files specified in the options above and applies a Hanning filter to it. We can then take a random sample and reduce the dataset to a smaller size, if desired, but by default we keep all of the observations. `dat` is a list of lists—each sublist corresponds to a waveform in the data and lists its values at regular time points.

```
readAndFilterData[] := (

  print[0, "PIKA: about to read data. iNPhoton ", iNPhoton];
  dat = readTESandFilter[iNPhoton, iDataSet, nTime,
    fileInfo];
  print[0, "PIKA: {nObs,nTime} ", Dimensions[dat]];

  If[nDatUse > 0,
   dat = randomSample[dat, nDatUse];
   print[0, "PIKA: nDatUse … reduced to {nObs, nTime} ",
    Dimensions[dat]];
  ];

 )
```

In addition to the noise filter, we can remove observations from the dataset that have been contaminated by background radiation. `getIObsKeep` returns the indices of the observations in `dat` that should not be thrown out, based on the parameters `peakPosCut`, `peakValCut`, and `peakNumCut` that characterize waveforms unduly influenced by background radiation. The graphical output is a random sample of rejected observations.

```
optionallyRejectBackgroundTraces[] := (

  If[backgroundReject,
   iObsKeep = getIObsKeep[dat, peakPosCut, peakValCut,
     peakNumCut];

   iObsDrop = Complement[Range[Length[dat]], iObsKeep];
   p1 = ListPlot[dat⟦ randomSample[iObsDrop, mSample] ⟧,
     PlotStyle → Blue,
     FrameLabel → {"t (shorter input units)",
       "V (input units)"},
     PlotLabel → ToString[mSample] <> " rejected traces",
     ImageSize → outputImageSize
    ];
   outputAdd[p1, 0, ""];

   dat = dat ⟦ iObsKeep ⟧;
   iObsDrop =.;
   (* after dat is overwritten these index sets lose
    their meaning *)
   iObsKeep =.
  ];

  nObs = Length[dat];
  print[0, "PIKA: nObs ", nObs, " traces selected"];

 )
```

meanTrace is the average of all of the waveforms in dat. The output here is a graph of meanTrace with a sample of accepted traces underlaid.

```
graphGlobalMeanTrace[] := (
  meanTrace = getMeanOfEachTime[dat];
  p2a = ListPlot[randomSample[dat, mSample],
    PlotStyle → Black];
  p2b = ListPlot[meanTrace, PlotStyle → Red];
  p2 = Show[p2a, p2b, DisplayFunction → $DisplayFunction,
    FrameLabel → {"Time (input units)",
      "Voltage (input units)"},
    PlotLabel → ToString[mSample] <>
      " accepted traces (black) and mean trace (red)",
    ImageSize → outputImageSize];
  Export["p2.pdf", p2];
  outputAdd[p2, 0, ""];
 )
```

For the first mean photon number in nPhotonAvgList, the effective photon number nPhotonEff⟦i⟧ for observation i is found via the dot-product method. For subsequent means, the optimized effective photon number from the previous item on nPhoton⋱AvgList serves as the initial estimate for the next mean, since the elements of nPhotonAvgList are expected to be in order and close together. In the beginning of Loop 1, we create the initial clustering based on these effective photon numbers.

```
findDotProductEffectivePhotonNumbers[] := (
  nPhotonAvg = First[nPhotonAvgList];
  nPhotonEff = nPhotonAvg * getDotToIdeal[dat, meanTrace];
 )
```

◻ 4.2. Preprocessing (Loop 1)

Now we enter Loop 1, whose index, iNPhotonAvgList, is the position in nPhoton⋱AvgList of the current mean photon number. Each iteration of Loop 1 runs PIKA on the given dataset with the assumption that the waveforms in the data come from pulses incident on the sensor with an average of nPhotonAvgList⟦iNPhotonAvgList⟧ photons per pulse. The runs are independent of each other apart from the fact that one round's optimized effective photon numbers are used as initial effective photon numbers for the next.

The initial clusters are formed from the effective photon numbers of the individual observations. The effective photon numbers order the observations, but the relative sizes of the clusters and the photon number to which each one corresponds are determined by the mean photon number, nPhotonAvg, and the Poisson distribution. groupProbability, defined and described in more detail in Section 7.4, returns:

1. prob—a list of ordered pairs, the first a photon number and the second the Poisson probability mass function at that photon number.

2. `nPhotonOfClust`—a list of photon numbers for the clusters created.
3. `iObsOfClust`—a list of lists. Each sublist corresponds to a cluster and contains the indices in `dat` of the waveforms in that cluster. The cluster `iObsOfClust⟦i⟧` has photon number `nPhotonOfClust⟦i⟧`.
4. `clustMeanTrace`—a list of waveforms. `clustMeanTrace⟦i⟧` is the average waveform of the cluster `iObsOfClust⟦i⟧`.

`nClust` is then the number of clusters, which `groupProbability` decides based on which photon numbers we expect to see in at least one observation.

```
createInitialClusters[] := (
  nPhotonAvg = nPhotonAvgList⟦iNPhotonAvgList⟧;
  {prob, nPhotonOfClust, iObsOfClust, clustMeanTrace} =
   groupProbability[dat, nPhotonAvg, nPhotonEff, nSigma];
  nClust = Length[clustMeanTrace];
  print[iNPhotonAvgList, "PIKA: nClust ", nClust];
  printSp[iNPhotonAvgList, "",
   "Initial mean photon number: ", nPhotonAvg];
 )
```

The following code graphs samples of waveforms from the middle three clusters.

```
graphSampleMiddleClusters[] := (
  p4a =
   ListPlot[randomSample[
     dat⟦ iObsOfClust⟦ Round[nClust / 2] - 1 ⟧ ⟧, mSample],
    PlotStyle → Red, DisplayFunction → Identity];
  p4b =
   ListPlot[randomSample[
     dat⟦ iObsOfClust⟦ Round[nClust / 2] ⟧ ⟧, mSample],
    PlotStyle → Green, DisplayFunction → Identity];
  p4c =
   ListPlot[randomSample[
     dat⟦ iObsOfClust⟦ Round[nClust / 2] + 1 ⟧ ⟧, mSample],
    PlotStyle → Blue, DisplayFunction → Identity];
  p4 = Show[p4c, p4b, p4a,
    FrameLabel → {"Time (input units)",
      "Voltage (input units)"},
    PlotLabel → ToString[mSample] <>
      " traces for three clusters n=" <>
      ToString[nPhotonOfClust⟦Round[nClust / 2] - 1⟧] <>
      " (red) to n=" <>
      ToString[nPhotonOfClust⟦Round[nClust / 2] + 1⟧] <>
      " (blue)",
    DisplayFunction → $DisplayFunction,
    ImageSize → outputImageSize];
  printSp[iNPhotonAvgList, "Sample Traces",
   "Before optimization:"];
  outputAdd[p4, iNPhotonAvgList, "Sample Traces"];
 )
```

Here we graph the average waveform of each cluster.

```
graphClusterMeans[] := (
  p6 = ListPlot[clustMeanTrace,
    PlotStyle → Green,
    PlotLabel → "Cluster mean traces",
    FrameLabel → {"Time (input units)",
      "Voltage (input units)"},
    ImageSize → outputImageSize];
  printSp[iNPhotonAvgList, "Cluster Means",
   "Before optimization:"];
  outputAdd[p6, iNPhotonAvgList, "Cluster Means"];
 )
```

The initial, unoptimized value of the objective function, from equation (4) above, is $O_{KPC} = \frac{1}{2\sigma^2} O_K + O_{PC}$. Here, `sqDevOfClust` is a table containing the square deviation of each cluster from its own mean (each element is $\sum_{i\in C_n} \frac{1}{N_t} \sum \left[V_i(t) - \overline{V}_n(t)\right]^2$ for cluster C_n), so the table's sum is O_K, the K-means term of the objective function. `nOfClust` is a list of cluster sizes, and `logLikeProb` is O_{PC}, the Poisson-combinatorial term. `sigmaObjFtn` is σ, the regularization parameter for the objective function based on the deviations of the initial clusters (unlike O_K, σ does not change as the objective function is minimized).

```
findInitialObjectiveFunction[] := (

  sqDevOfClust = getSqDevOfClust[dat, iObsOfClust,
    clustMeanTrace];
  nOfClust = Map[Length, iObsOfClust];
  logLikeProb = probComboLogLikelihood[nPhotonOfClust,
    nOfClust, nPhotonAvg];
  sigmaObjFtn = Sqrt[Total[sqDevOfClust] / nObs] // N;
  print[iNPhotonAvgList, "PIKA: sigmaObjFtn ",
   sigmaObjFtn];
  sigmaObjFtn = Table[sigmaObjFtn, {nClust}];

  objFtn = getObjFtn[sqDevOfClust, sigmaObjFtn,
    logLikeProb];
  printSp[iNPhotonAvgList, "",
   "Initial objective function: ", objFtn];
 )
```

This graphs the root mean square (RMS) deviations of each cluster to its mean.

```
graphClusterDeviationsFromMean[] := (

  p7 = ListPlot[pair[nPhotonOfClust, Sqrt[sqDevOfClust/nOfClust]],
    AxesOrigin → {
      Floor[Min[nPhotonOfClust], 10],
      Floor[Min[Sqrt[sqDevOfClust/nOfClust]], 10]
    },
    Joined → False,
    PlotStyle → PointSize[0.015],
    FrameLabel → {"Number of photons n",
      "RMS deviation (Voltage input units)"},
    PlotLabel →
     "RMS deviations of clusters to their means",
    ImageSize → outputImageSize
  ];
  printSp[iNPhotonAvgList, "Cluster Deviations",
   "Before optimization:"];
  outputAdd[p7, iNPhotonAvgList, "Cluster Deviations"];
)
```

The optimization loop (Loop 2) is now ready to begin, apart from a few organizational tasks. First, since the optimization moves waveforms between clusters many times, we need a more convenient way to keep track of the clusters. `getIClustOfObs` converts `iObsOfClust`, a list of lists, into `iClustOfObs`, a one-dimensional list, each element of which corresponds to an observation in `dat` and gives the index of the cluster to which that observation belongs.

Next, for each cluster, `neighborOfClust` lists the acceptable clusters to transfer to during the optimization loop. By default, the neighbors for cluster n are the clusters $n+1$ and $n-1$, since the initial clustering (from the dot-product method or the previous round of PIKA) is expected to be at least a somewhat accurate arrangement, and so long-distance transfers should not be necessary.

Finally, a system for keeping track of transfer history lets us reduce unnecessary computation. `kMove` counts the number of transfers that have been made in the loop. `birthOf⋱ Clust` is a table that records the move number when each cluster was last changed, and `birthOfObsClust` is a two-dimensional table that records, for each observation-cluster pair, the last time that the deviation from the observation to the cluster mean was calculated. `sqDevOfObsClust` is a table with the same dimensions that records the actual deviations so that they can be used later in the loop if they are still up to date.

```
prepareOptimizationLoop[] := (

   If[Sort[Flatten[iObsOfClust]] != Range[Length[dat]],
    print[iNPhotonAvgList,
     "PIKA: WARNING: iObsOfClust does not map each
       observation to a cluster"]
   ];
   iClustOfObs = getIClustOfObs[iObsOfClust];
   If[Min[iClustOfObs] <= 0,
    print[iNPhotonAvgList,
     "PIKA: WARNING: iClustOfObs does not map each
       observation to a cluster"]
   ];

   neighborOfClust = pair[Range[nClust] - 1,
     Range[nClust] + 1];
   neighborOfClust[[1]] = Drop[neighborOfClust[[1]], 1];
   neighborOfClust[[-1]] = Drop[neighborOfClust[[-1]], -1];

   kMove = 0; (* count of moves made *)
   birthOfClust = Table[0, {nClust}];
   birthOfObsClust = Table[-1, {nObs}, {nClust}];
   (* older than birthOfClust *)
   sqDevOfObsClust = birthOfObsClust
   (* dummy array of correct shape *)

  )
```

4.3. Optimization Loop (Loop 2)

Loop 2, nested inside Loop 1, begins here. Its index, `iCool`, specifies how many rounds of optimization have already passed. The optimization loop runs for a preset number of iterations (given by `nCool` in the initial setup), the first `nGreedy` of which employs the greedy algorithm. The remaining iterations are run with the simulated annealing algorithm. By default, `nGreedy` is equal to `nCool`, so all of the iterations use the greedy algorithm (i.e. simulated annealing is never used).

The bulk of Loop 2 is contained in the nested Loop 3, the waveform transfer loop that passes over all of the observations in `dat` and decides whether to move them to neighboring clusters. Before starting Loop 3, we randomly order the indices in `dat` of the waveforms upon which Loop 3 is to operate, so that the original ordering of the data does not bias the algorithm in any systematic way.

```
prepareSubLoop[] := (
  iObsAll = RandomSample[Range[nObs]];
  printSp[iNPhotonAvgList, "Detailed Log", "Mean ",
   iNPhotonAvgList, ", round ", iCool,
   ": kMove ", kMove, " logLikeProb ", logLikeProb,
   " objFtn ", objFtn, " TimeUsed ", TimeUsed[] - tStart];
 )
```

■ *4.3.1. Waveform Transfer Loop (Loop 3)*

The index of Loop 3, `jObs`, is the current position of the loop in `iObsAll`, the random permutation of the observation indices. `iObs` is then set to the index of the observation itself, rather than the index of the index. `iClust` is the index of the associated cluster. If the cluster has only one observation then we skip to the next iteration of Loop 3, since no cluster is allowed to become empty. We then randomly pick a cluster `jClust` from the list of neighbors, and we consider moving the waveform from its current cluster to the new one.

```
getClusterAndNeighbors[] := (
  iObs = iObsAll⟦jObs⟧;
  iClust = iClustOfObs ⟦ iObs ⟧;
  If[nOfClust⟦iClust⟧ <= 1, Continue[]];
  neighbor = neighborOfClust⟦ iClust ⟧;
  jClust = neighbor⟦ RandomInteger[{1, Length[neighbor]}] ⟧
 )
```

To reduce the computation time, `birthOfObsClust` and `birthOfClust` record the "ages" of observation-to-mean deviation calculations and changes to cluster contents, respectively, so this step checks whether `iClust` and `jClust` have been modified since we last calculated the deviation of `iObs` from their means. If they are unchanged, then `sqDevOfObsClust` is up to date; this condition becomes increasingly common as the calculation approaches convergence. If they are changed, we need to recalculate one or both deviations and store them in `sqDevOfObsClust`.

```
updateRelevantDeviations[] := (
  If[birthOfObsClust⟦iObs, iClust⟧ < birthOfClust⟦iClust⟧,
   sqDevOfObsClust⟦iObs, iClust⟧ =
    meanSquare[dat⟦iObs⟧ - clustMeanTrace⟦iClust⟧];
   birthOfObsClust⟦iObs, iClust⟧ = kMove
  ];
  If[birthOfObsClust⟦iObs, jClust⟧ < birthOfClust⟦jClust⟧,
   sqDevOfObsClust⟦iObs, jClust⟧ =
    meanSquare[dat⟦iObs⟧ - clustMeanTrace⟦jClust⟧];
   birthOfObsClust⟦iObs, jClust⟧ = kMove
  ]
 )
```

We need to know how a proposed move would change the objective function before deciding whether to accept it. Section 5 gives more detail on the functions called here that efficiently compute the change in the objective function.

```
findChangeInObjectiveFunction[] := (
  deltaI = deltaKMeansDel[nOfClust[[iClust]],
    sqDevOfObsClust[[iObs, iClust]]];
  deltaJ = deltaKMeansAdd[nOfClust[[jClust]],
    sqDevOfObsClust[[iObs, jClust]]];
  deltaKMeansScaled = deltaJ / (2 * sigmaObjFtn[[jClust]]^2) -
    deltaI / (2 * sigmaObjFtn[[iClust]]^2);
  deltaProb = If[jClust < iClust,
    deltaPoissonDn[nPhotonOfClust[[iClust]], nOfClust[[jClust]],
     nOfClust[[iClust]], nPhotonAvg],
    deltaPoissonUp[nPhotonOfClust[[iClust]], nOfClust[[jClust]],
     nOfClust[[iClust]], nPhotonAvg]
   ];
  deltaObjFtn = deltaKMeansScaled - deltaProb;
 )
```

If the transfer would decrease the objective function, we automatically accept it, and if we use the greedy algorithm, we reject any transfer that does not decrease the objective function. If we use simulated annealing, however, we may accept a transfer that increases the objective function for the purpose of exploring a greater number of assignments and possibly avoiding local minima that would trap the greedy algorithm. In simulated annealing, it grows harder for a disadvantageous transfer to occur as time goes on and the “temperature” drops.

```
decideWhetherToMove[] := (
  acceptMove = If[deltaObjFtn < 0,
    True,
    If[iCool > nGreedy,
     RandomReal[] < Exp[-deltaObjFtn / tAnneal],
     False
    ]
   ]
 )
```

If the move is accepted, then we need to move the observation from one cluster to the other and update the clusters' deviations and mean traces. Both clusters have been changed in this move, so we update `birthOfClust` accordingly.

```
updateIfMoving[] := (
  If[acceptMove, (* transfer a trace from cluster i
    to cluster j *)
   printSp[iNPhotonAvgList, "Detailed Log", "   Round ",
    iCool, ", observation ", iObs, ": deltaObjFtn ",
    deltaObjFtn, " from ", iClust, " to ", jClust];
   kMove++;
   birthOfClust⟦iClust⟧ = kMove;
   clustMeanTrace⟦iClust⟧ =
    newMeanTraceDel[nOfClust⟦iClust⟧,
     clustMeanTrace⟦iClust⟧, dat⟦iObs⟧];
   nOfClust⟦iClust⟧--;
   birthOfClust⟦jClust⟧ = kMove;
   clustMeanTrace⟦jClust⟧ =
    newMeanTraceAdd[nOfClust⟦jClust⟧,
     clustMeanTrace⟦jClust⟧, dat⟦iObs⟧];
   nOfClust⟦jClust⟧++;
   iClustOfObs⟦iObs⟧ = jClust;
   sqDevOfClust⟦iClust⟧ -= deltaI;
   sqDevOfClust⟦jClust⟧ += deltaJ;
   objFtn += deltaObjFtn;
  ]
 )
```

After making a transfer or deciding not to, Loop 3 moves to the next waveform in the randomly ordered list or finishes if there is none.

■ *4.3.2. Annealing Update and End of Loop 2*

Loop 3 is now over, after examining all of the observations and moving some to neighboring clusters. If we use the simulated annealing algorithm, the annealing temperature decreases after Loop 3, making it more difficult for a move that increases the objective function to be accepted.

```
updateAnnealingTemperature[] := (
  tAnneal *= coolConst
 )
```

Loop 2 repeats a preset number of times, randomly ordering the waveforms, running Loop 3 on each one, and then decreasing the annealing temperature in each iteration.

▫ 4.4. Data Analysis and Output (Loop 1)

When Loop 2 is finished, PIKA has run for many rounds on the data and the observations should be arranged into optimal clusters. This subsection concerns itself primarily with creating graphical and numerical outputs to understand and visualize the clustering. Now that the optimization is finished, `iObsOfClust` (the list of lists) is a more useful format than `iClustOfObs` (the simple list). In order to compare the new clustering to the old, we generate `iObsOfClustNew` from `iClustOfObs`, which reflects the optimized clusters, and simply copy `iObsOfClustOld` from `iObsOfClust`, which has not been changed since before Loop 2. `freqNew` and `freqOld` are lists of the new and old cluster sizes, respectively, and `nPhotonAvgNew` is an estimate of the actual mean photon number based on the results of PIKA.

```
prepareAnalysisAndOutput[] := (
  print[iNPhotonAvgList, "final ", nCool, " kMove ",
   kMove, " logLikeProb ", logLikeProb, " objFtn ",
   objFtn, " TimeUsed ", TimeUsed[] - tStart];
  iObsOfClustNew = getIObsOfClust[iClustOfObs];
  iObsOfClustOld = iObsOfClust;
  (* not changed since before the optimization loop *)
  freqNew = Map[Length, iObsOfClustNew];
  freqOld = Map[Length, iObsOfClustOld];
                  nPhotonOfClust.freqNew
  nPhotonAvgNew = ---------------------- // N;
                     Total[freqNew]
  print[iNPhotonAvgList, "nPhotonAvg ", nPhotonAvg,
   " nPhotonAvgNew ", nPhotonAvgNew];
  printSp[iNPhotonAvgList, "",
   "Optimized mean photon number: ", nPhotonAvgNew];
  printSp[iNPhotonAvgList, "",
   "Optimized objective function: ", objFtn];
 )
```

As before, we graph some sample waveforms from middle clusters.

```
graphNewSampleMiddleClusters[] := (
  p8a = ListPlot[
    randomSample[dat[[ iObsOfClustNew[[ Round[nClust / 2] - 1 ]] ]],
     mSample],
    PlotStyle → Red,
    DisplayFunction → Identity];
  p8b = ListPlot[
    randomSample[dat[[ iObsOfClustNew[[ Round[nClust / 2] ]] ]],
     mSample],
    PlotStyle → Green,
    DisplayFunction → Identity];
```

```
   p8c = ListPlot[
     randomSample[dat⟦ iObsOfClustNew⟦ Round[nClust / 2] + 1 ⟧ ⟧,
      mSample],
     PlotStyle → Blue,
     DisplayFunction → Identity];
   p8 = Show[p8c, p8b, p8a,
     PlotLabel → ToString[mSample] <>
       " traces for three clusters n=" <>
       ToString[nPhotonOfClust⟦Round[nClust / 2] - 1⟧] <>
       " (red) to n=" <>
       ToString[nPhotonOfClust⟦Round[nClust / 2] + 1⟧] <>
       " (blue)",
     FrameLabel → {"Time (input units)",
       "Voltage (input units)"},
     DisplayFunction → $DisplayFunction,
     ImageSize → outputImageSize];
   printSp[iNPhotonAvgList, "Sample Traces",
    "After optimization:"];
   outputAdd[p8, iNPhotonAvgList, "Sample Traces"];
  )
```

Here we numerically output and then graph the optimized cluster mean waveforms, with a graph of both the optimized and initial means as well.

```
graphNewClusterMeans[] := (

   clustFreq = pair[nPhotonOfClust, freqNew];
   printSp[iNPhotonAvgList, "Numerical Cluster Means",
    "After optimization,
      {nPhotonAvgNew,clustFreq,clustMeanTrace}="];
   outputAdd[{nPhotonAvgNew, clustFreq, clustMeanTrace},
    iNPhotonAvgList, "Numerical Cluster Means"];

   p9a = ListPlot[clustMeanTrace, DisplayFunction → Identity,
     PlotStyle → Blue,
     PlotLabel → "Optimized cluster mean traces",
     ImageSize → outputImageSize];
   p9 = Show[p6, p9a, DisplayFunction → $DisplayFunction,
     PlotLabel →
      "Initial (green) and optimized (blue) cluster
        mean traces"];
   printSp[iNPhotonAvgList, "Cluster Means",
    "After optimization:"];
   outputAdd[p9a, iNPhotonAvgList, "Cluster Means"];
   outputAdd[p9, iNPhotonAvgList, "Cluster Means"];

  )
```

This graph compares the initial and final cluster counts to a Poisson distribution.

```
graphProbabilityDistribution[] := (
  p10 = ListPlot[
    {pair[nPhotonOfClust, freqOld],
     pair[nPhotonOfClust, freqNew],
     pair[prob⟦All, 1⟧, prob⟦All, 2⟧ * nObs]},
    PlotStyle → {Green, Red, Black},
    Joined → {False, False, True},
    PlotLabel →
     "Initial (green) and final (red) cluster
        frequencies; " <> probDistName <>
      " distribution (black)",
    FrameLabel → {"Number of photons n", "Frequencies"},
    ImageSize → outputImageSize
   ];
  outputAdd[p10, iNPhotonAvgList, probDistName];
 )
```

`sqDevClustClust⟦i, j⟧` is the average mean square deviation of the observations in cluster `i` to the mean waveform in cluster `j`. This graph is of the RMS deviations of the traces in clusters to adjacent means.

```
graphNewClusterToMeanDeviations[] := (
  sqDevClustClust = getSqDevClustClust[dat, iObsOfClustNew,
    clustMeanTrace];
  sqrtsdccDiag = Table[
    {nPhotonOfClust⟦iClust⟧,
     √sqDevClustClust⟦iClust, iClust⟧},
    {iClust, nClust}];
  sqrtsdccUDiag = Table[
    {nPhotonOfClust⟦iClust⟧,
     √sqDevClustClust⟦iClust, iClust + 1⟧},
    {iClust, nClust - 1}];
  sqrtsdccLDiag = Table[
    {nPhotonOfClust⟦iClust⟧,
     √sqDevClustClust⟦iClust, iClust - 1⟧},
    {iClust, 2, nClust}];
  p13 = ListPlot[{sqrtsdccDiag, sqrtsdccLDiag,
      sqrtsdccUDiag}, PlotStyle → {Red, Green, Blue},
     Joined → True, Frame → True,
     PlotLabel →
      "RMS deviation of cluster traces to means of
        clusters: R: same; G: +1; B: -1",
     FrameLabel →
      {"Cluster", "RMS deviation (input voltage units)"},
     ImageSize → outputImageSize
    ];
  printSp[iNPhotonAvgList, "Cluster Deviations",
   "After optimization:"];
  outputAdd[p13, iNPhotonAvgList, "Cluster Deviations"];
 )
```

`sqDevMeanMean⟦i, j⟧` is the mean square deviation of the mean of cluster `i` to that of cluster `j`. This graph is of the RMS deviations of cluster means to means one, two, and three clusters away.

```
graphNewMeanToMeanDeviations[] := (
  sqDevMeanMean = getSqDevClustObs[clustMeanTrace,
    clustMeanTrace];
  sqrtsdmmDiag1 = Table[
    {nPhotonOfClust⟦iClust⟧,
     √(sqDevMeanMean⟦iClust, iClust + 1⟧)},
    {iClust, nClust - 1}];
  sqrtsdmmDiag2 = Table[
    {nPhotonOfClust⟦iClust⟧,
     √(sqDevMeanMean⟦iClust, iClust + 2⟧)},
    {iClust, nClust - 2}];
  sqrtsdmmDiag3 = Table[
    {nPhotonOfClust⟦iClust⟧,
     √(sqDevMeanMean⟦iClust, iClust + 3⟧)},
    {iClust, nClust - 3}];
```

```
   p15 = ListPlot[{sqrtsdmmDiag1, sqrtsdmmDiag2,
       sqrtsdmmDiag3},
     PlotStyle → {Red, Green, Blue},
     Joined → True,
     Frame → True,
     PlotLabel →
      "RMS deviation of cluster means to each other,
        R: +1 G: +2 B: +3",
     FrameLabel →
      {"Cluster", "RMS deviation (voltage input units)"},
     ImageSize → outputImageSize
 ];
   outputAdd[p15, iNPhotonAvgList, "Cluster Deviations"];
  )
```

nPhotonEff lists the effective photon number of each observation (getNPhotonEff is implemented in Section 7.2). The rest of this code creates a histogram of nPhotonEff with bin widths of binFract: binEdge specifies where the edges of the bins should be, binCnt counts the number of waveforms in each bin, and binCtr lists the centers of the bins.

```
graphNewEffectivePhotonNumber[] := (
   nPhotonEff = getNPhotonEff[dat, nPhotonOfClust,
     clustMeanTrace];
   binEdge = Table[nEff,
     {nEff,
      (Floor[Min[nPhotonEff]/binFract] - 1/2) * binFract,
      (Ceiling[Max[nPhotonEff]/binFract] + 1/2) * binFract,
      binFract}
    ];
   binCnt = BinCounts[nPhotonEff,
     {(Floor[Min[nPhotonEff]/binFract] - 1/2) * binFract,
      (Ceiling[Max[nPhotonEff]/binFract] + 1/2) * binFract,
      binFract}
    ];
   binCtr = (Most[binEdge] + Rest[binEdge])/2;
   pairCnt = pair[binCtr, binCnt];
   pairCntPlot =
    pair[binCtr, Table[If[binCnt⟦i⟧ < 0.5, 0.5, binCnt⟦i⟧],
      {i, Length[binCnt]}]];
p17 = ListLogPlot[pairCntPlot,
   PlotLabel →
    "Effective photon number histogram (bin width " <>
     TextString[binFract] <> ")",
   FrameLabel → {"Photon number", "counts"},
   ImageSize → outputImageSize
  ];
outputAdd[p17, iNPhotonAvgList,
  "Effective Photon Numbers"];

printSp[iNPhotonAvgList, "Numerical $n_{eff}$ Bin Counts",
  "First element is bin center; second is bin count."];
outputAdd[pairCnt, iNPhotonAvgList,
  "Numerical $n_{eff}$ Bin Counts"];
printSp[iNPhotonAvgList, "Numerical $n_{eff}$",
  "Effective photon number for each observation:"];
outputAdd[nPhotonEff, iNPhotonAvgList, "Numerical $n_{eff}$"];
)
```

Loop 1 runs PIKA on the data once with each mean photon number in `nPhotonAvgList` and then exits.

4.5. Show Output

After Loop 1 finishes, the algorithm outputs the computation time used and displays the output from PIKA.

```
showOutput[] := (
  tEnd = TimeUsed[];
  print[0, "PIKA: time used: ", tEnd - tStart, " s"];
  outputShowTabView[]
 )
```

5. Updates to the Objective Function

A naive implementation of the transfer of waveforms between clusters would recalculate the various components of the objective function from scratch with each change, resulting in a computationally expensive process whose running time would depend on the size of the clusters in question. We can avoid this excessive calculation by noticing that, given some basic summary information about the clusters and waveform to which a given transfer pertains, the changes to the objective function and the cluster means are very easy to compute without any knowledge of the other waveforms in the clusters. The running times of the updates in this section are independent of cluster size.

5.1. Updating the *K*-Means Objective Function

From equation (1), we can decompose the *K*-means term of the objective function as

$$O_K = \sum_{n=n_0}^{n_0+K-1} J_n, \tag{8}$$

where

$$J_n = \sum_{i \in C_n} \frac{1}{N_t} \sum_t \left[V_i(t) - \overline{V}_n(t) \right]^2. \tag{9}$$

If we transfer a waveform j into the cluster $n = a$, the new cluster members form the set $C_a^+ = C_a \bigcup \{j\}$. It can be shown that

$$J_a^+ = J_a + \left(\frac{m_a}{m_a + 1} \right) \frac{1}{N_t} \sum_t \left[V_j(t) - \overline{V}_a(t) \right]^2. \tag{10}$$

If we transfer a waveform j out of the cluster $n = b$, the new cluster members form the set $C_b^- = C_b - \{j\}$, with

$$J_b^- = J_b - \left(\frac{m_b}{m_b - 1} \right) \frac{1}{N_t} \sum_t \left[V_j(t) - \overline{V}_b(t) \right]^2. \tag{11}$$

In equations (10) and (11), $\overline{V}_a(t)$ and $\overline{V}_b(t)$ refer to the original clusters C_a and C_b, before the transfer of j. See below for a proof that these equations indeed give the proper changes in cluster deviation. Equation (11) corrects equation (A4) in [1]. After the transfer, the cluster means for $n = a$ and $n = b$ become

$$\overline{V}_n'(t) = \frac{m_n \overline{V}_n(t) \pm V_j(t)}{m_n \pm 1}, \tag{12}$$

with plus signs for $n = a$ and minus signs for $n = b$. No cluster is ever allowed to become empty (in such a case we would have not K but $K - 1$ means), so the denominator never becomes zero. Therefore, we can determine the appropriate change to the means and square deviations of the clusters between which the transfer takes place.

```
deltaKMeansAdd[
  freq_, (* size of the cluster to be added to *)
  meanSquareDiff_ (* mean square of the difference
   between the waveform to be transferred and the
   cluster mean waveform *)
 ] :=
 (freq/(freq + 1)) * meanSquareDiff
```

```
deltaKMeansDel[
  freq_, (* size of the cluster to be removed from *)
  meanSquareDiff_ (* mean square of the difference
   between the waveform to be transferred and the
   cluster mean waveform *)
 ] :=
 (freq / (freq - 1)) * meanSquareDiff

newMeanTraceAdd[freq_, oldMeanTrace_, transferTrace_] :=
 (freq * oldMeanTrace + transferTrace) / (freq + 1)

newMeanTraceDel[freq_, oldMeanTrace_, transferTrace_] :=
 (freq * oldMeanTrace - transferTrace) / (freq - 1)
```

■ 5.1.1. Example: The *K*-Means Update

Suppose that cluster *a* has nine waveforms and cluster *b* has 14 (and each has some mean waveform in addition), and we want to move a waveform from *b* to *a*. When we make this transfer, O_K changes by the following amount.

```
freqAddA = 9;
freqDelA = 14;
meanTraceAdd = {1.1, 2.9, 8.9, 9.0, 6.1, 4.3};
meanTraceDel = {0.9, 1.7, 4.0, 8.8, 5.2, 3.3};
transferTrace = {1.2, 3.1, 8.9, 9.5, 8.1, 5.2};
deltaKMeansAdd[freqAddA,
  meanSquare[meanTraceAdd - transferTrace]] -
 deltaKMeansDel[freqDelA,
  meanSquare[meanTraceDel - transferTrace]]

-6.15632
```

The mean waveforms of clusters *a* and *b* become the following.

```
newMeanTraceAdd[freqAddA, meanTraceAdd, transferTrace]

{1.11, 2.92, 8.9, 9.05, 6.3, 4.39}

newMeanTraceDel[freqDelA, meanTraceDel, transferTrace]

{0.876923, 1.59231, 3.62308, 8.74615, 4.97692, 3.15385}
```

Of course, the cluster sizes change as well: *a* now has 10 waveforms, while *b* has 13.

5.1.2. Derivation of the Update Formulas

The derivation for equations (10) and (11) is slightly involved and requires some preliminaries. Voltage signals are considered as vectors for the derivation, denoted V_i, where the discrete time index t is not given explicitly in this section.

Lemma:

Let $\mathbf{V}_i$ *be the signal vector of the* i^{th} *observation, and* $\overline{\mathbf{V}}_n$ *and* J_n *be the cluster mean vector and sum of deviations, respectively, for cluster* C_n*. Then,*

$$J_n = \sum_{i \in C_n} \frac{\| \mathbf{V}_i - \overline{\mathbf{V}}_n \|^2}{N_t} = \sum_{i,j \in C_n} \frac{\| \mathbf{V}_i - \mathbf{V}_j \|^2}{2\, m_n\, N_t}.$$

Proof:

The first expression for J_n is merely the definition; we now show that the second expression is equivalent.

$$\sum_{i,j \in C_n} \frac{\| \mathbf{V}_i - \mathbf{V}_j \|^2}{2\, m_n\, N_t}$$

$$= \frac{1}{2\, m_n\, N_t} \left[m_n \sum_{i \in C_n} \| \mathbf{V}_i \|^2 - \sum_{i,j \in C_n} 2\, \mathbf{V}_i \cdot \mathbf{V}_j + m_n \sum_{j \in C_n} \| \mathbf{V}_j \|^2 \right]$$

$$= \frac{1}{N_t} \sum_{i \in C_n} \left(\| \mathbf{V}_i \|^2 - \mathbf{V}_i \cdot \overline{\mathbf{V}}_n \right)$$

$$= \frac{1}{N_t} \sum_{i \in C_n} \mathbf{V}_i \cdot \left(\mathbf{V}_i - \overline{\mathbf{V}}_n \right)$$

$$= \frac{1}{N_t} \sum_{i \in C_n} \left(\mathbf{V}_i - \overline{\mathbf{V}}_n \right) \cdot \left(\mathbf{V}_i - \overline{\mathbf{V}}_n \right),$$

which is equivalent to the first expression. (The final step is valid because $\sum_{i \in C_n} \left(\mathbf{V}_i - \overline{\mathbf{V}}_n \right) = 0$.) □

Lemma:

Let $\mathbf{V}_i$*,* $\overline{\mathbf{V}}_n$*,* J_n*, and* C_n *be as before. Then,*

$$\| \mathbf{V}_k - \overline{\mathbf{V}}_n \|^2 + \frac{N_t\, J_n}{m_n} = \| \mathbf{V}_k \|^2 - 2\, \mathbf{V}_k \cdot \overline{\mathbf{V}}_n + \frac{1}{m_n} \sum_{i \in C_n} \| \mathbf{V}_i \|^2 .$$

Proof:

$$\| \mathbf{V}_k - \overline{\mathbf{V}}_n \|^2 = \| \mathbf{V}_k \|^2 - 2\, \mathbf{V}_k \cdot \overline{\mathbf{V}}_n + \frac{1}{{m_n}^2} \sum_{i,j \in C_n} \mathbf{V}_i \cdot \mathbf{V}_j.$$

From the proof of the previous lemma, we have

$$J_n = \frac{1}{N_t}\left(\sum_{i\in C_n} \| \mathbf{V}_i \|^2 - \frac{1}{m_n}\sum_{i,j\in C_n} \mathbf{V}_i \cdot \mathbf{V}_j\right),$$

and thus

$$\frac{1}{m_n}\sum_{i,j\in C_n} \mathbf{V}_i \cdot \mathbf{V}_j = \sum_{i\in C_n} \| \mathbf{V}_i \|^2 - N_t J_n.$$

Finally,

$$\| \mathbf{V}_k - \overline{\mathbf{V}}_n \|^2 = \| \mathbf{V}_k \|^2 - 2\,\mathbf{V}_k \cdot \overline{\mathbf{V}}_n + \frac{1}{m_n}\left(\sum_{i\in C_n} \| \mathbf{V}_i \|^2 - N_t J_n\right),$$

and the lemma follows. □

Now we can prove that equations (10) and (11) hold true.

Theorem:

Suppose we add or remove a waveform $\mathbf{V}_p$ *to or from a cluster* C_n*, forming a cluster* $C_n{}'$*. Then,*

$$J_n{}' = J_n \pm \frac{m_n}{m_n'}\,\frac{\| \mathbf{V}_p - \overline{\mathbf{V}}_n \|^2}{N_t},$$

with $m_n{}' = m_n \pm 1$*. (Throughout this theorem, the top symbol of a plus-or-minus sign indicates the addition of* $\mathbf{V}_p$ *to* C_n*, and the bottom symbol indicates removal.)*

Proof:

By the first lemma,

$$J_n{}' = \sum_{i,j\in C_n{}'} \frac{\| \mathbf{V}_i - \mathbf{V}_j \|^2}{2\,m_n{}'\,N_t}.$$

(Note the summation over $C_n{}'$, not C_n.) Since $\mathbf{V}_p$ is the observation to add to or remove from C_n, we can write

$$\begin{aligned} J_n{}' = {} & \\ & \frac{1}{2\,m_n{}'\,N_t}\left(\sum_{i,j\in C_n} \| \mathbf{V}_i - \mathbf{V}_j \|^2 \pm \sum_{j\in C_n} \| \mathbf{V}_p - \mathbf{V}_j \|^2 \pm \sum_{i\in C_n} \| \mathbf{V}_i - \mathbf{V}_p \|^2 \pm \| \mathbf{V}_p - \mathbf{V}_p \|^2\right) \\ = {} & \frac{m_n}{m_n{}'} J_n \pm \frac{1}{m_n{}'\,N_t}\sum_{j\in C_n} \| V_p - \mathbf{V}_j \|^2 \\ = {} & \frac{m_n}{m_n{}'} J_n \pm \frac{m_n}{m_n{}'\,N_t}\left(\| \mathbf{V}_p \|^2 - 2\,\mathbf{V}_p \cdot \overline{\mathbf{V}}_n + \frac{1}{m_n}\sum_{j\in C_n} \| \mathbf{V}_j \|^2\right). \end{aligned}$$

By the second lemma, we have

$$
\begin{aligned}
J_n' &= \frac{m_n}{m_n'} J_n \pm \frac{m_n}{m_n' N_t}\left(\| \mathbf{V}_p - \bar{\mathbf{V}}_n \|^2 + \frac{N_t J_n}{m_n} \right) \\
&= \left(\frac{m_n}{m_n'} \pm \frac{1}{m_n'} \right) J_n \pm \frac{m_n}{m_n'} \frac{\| \mathbf{V}_p - \bar{\mathbf{V}}_n \|^2}{N_t},
\end{aligned}
$$

and the theorem follows. □

5.2. Updating the Poisson Log-Likelihood

From equation (2), the Poisson log-likelihood is

$$
\ln \mathcal{L}_P(m_0, \ldots, m_a, \ldots, m_b, \ldots, m_{n_0+K-1}; \mu) = -\mu M + \sum_{n=n_0}^{n_0+K-1} m_n [n \ln \mu - \ln(n!)], \tag{13}
$$

where $M = \sum m_n$. If some waveform is transferred from the $n = b$ to the $n = a$ cluster, then the change in the objective function consists of a term from the Poisson factor and a term from the combinatorial factor from equation (3). The term due to the Poisson factor is

$$
\begin{aligned}
\Delta \ln \mathcal{L}_P = {} & \ln \mathcal{L}_P(m_0, \ldots, m_a + 1, \ldots, m_b - 1, \ldots, m_{n_0+K-1}; \mu) - \\
& \ln \mathcal{L}_P(m_0, \ldots, m_a, \ldots, m_b, \ldots, m_{n_0+K-1}; \mu) = \\
& (a - b) \ln \mu - \ln(a!) + \ln(b!).
\end{aligned} \tag{14}
$$

(The notation suggests $b > a$, but the same formula applies for the $b < a$ case.)

The combinatorial factor's treatment is similar. The logarithm of the factor is

$$
\ln \mathcal{L}_C = \ln(M!) - \sum_{n=n_0}^{n_0+K-1} \ln(m_n!). \tag{15}
$$

M does not change with a transfer, so only the second term contributes to the change:

$$
\Delta \ln \mathcal{L}_C = \ln\left(\frac{m_b}{m_a + 1} \right). \tag{16}
$$

The algorithm only considers moves to adjacent clusters, so we only need to calculate the change due to transfers to immediately preceding and succeeding clusters. These two functions give the change in the probabilistic/combinatorial component of the objective function for moves to higher and lower photon numbers.

```
(* from n to n+1 *)
deltaPoissonUp = Compile[{
    {nDel, _Integer}, (* cluster number from which to
     remove the waveform *)
    {freqAdd, _Integer},
    (* size of the cluster to be added to *)
    {freqDel, _Integer},
     (* size of the cluster to be removed from *)
     {mu, _Real} (* mean photon number *)
    },
    Log[mu * freqDel / ((nDel + 1) * (freqAdd + 1))]
   ];

(* from n to n-1 *)
deltaPoissonDn = Compile[{
     {nDel, _Integer},
     {freqAdd, _Integer},
     {freqDel, _Integer},
     {mu, _Real}
    },
    Log[nDel * freqDel / (mu * (freqAdd + 1))]
   ];
```

Bear in mind that the term that $\mathcal{L}_{PC}$ contributes to the overall objective function is the negation of its logarithm, so a positive $\Delta\ln\mathcal{L}_{PC}$ decreases O_{KPC}.

5.2.1. Example: The Poisson Update

Suppose we have a set of clusters, among them an $n = 2$ cluster with five waveforms and an $n = 3$ cluster with seven waveforms, with a mean of 4.4 photons. We are interested in moving a waveform from the $n = 2$ cluster to the $n = 3$ cluster. Then the change in the Poisson log-likelihood is given by `deltaPoissonUp`.

```
mu = 4.4;
nAdd = 3;
nDel = 2;
freqAdd = 7; (* size of the cluster before the
  transfer *)
freqDel = 5;
deltaPoissonUp[nDel, freqAdd, freqDel, mu]

-0.0870114
```

6. Acknowledgments

We would like to thank Boris L. Glebov and Alan L. Migdall for providing the TES data. Figure 1 courtesy Thomas Gerrits and the Optical Society of America.

7. Appendix: Function Definitions

The functions in this section either are used often, and so are given their own names, or perform very particular procedures (such as data reading) whose implementation details are tangential to the core operation of the algorithm, and so have been separated from the main body of the code.

7.1. General Purpose

`boundList` confines the elements of a list between two inclusive bounds.

```
boundList[list_, min_, max_] :=
 Module[{i}, Table[Median[{list⟦i⟧, min, max}],
   {i, Length[list]}]]
```

`getDotToIdeal` takes a single trace or a set of traces and finds its or their dot product with an ideal trace, normalized by the square of the ideal trace's magnitude. If the function is given a single trace as its first argument, the operation in the numerator is a simple dot product; if it is given a list of traces, the operation is matrix-vector multiplication, which returns a list of dot products.

```
getDotToIdeal[traces_, idealTrace_] :=    traces.idealTrace
                                       ───────────────────────
                                       idealTrace.idealTrace
```

`getMeanOfEachTime` averages all of the traces in `dat` at each time point, returning the overall mean waveform.

```
getMeanOfEachTime[dat_] := Map[Mean, Transpose[dat]]
```

`getSquareDiff` returns a table that lists the squared magnitude of the difference between each observation and each cluster mean.

```
getSquareDiff[dat_, clustMeanTrace_] :=
 Module[{iClust, iObs, x},
  Table[x = dat⟦iObs⟧ - clustMeanTrace⟦iClust⟧; x.x,
   {iObs, Length[dat]}, {iClust, Length[clustMeanTrace]}]
 ]
```

`groupCommon` is a small function that rearranges the result of `GatherBy`. It is best understood from its definition.

```
groupCommon[x_] := {x⟦1, 2⟧, x⟦All, 1⟧}
```

`meanSquare` takes a vector and returns its mean square (magnitude squared divided by length).

```
                      x.x
meanSquare[x_] := ─────────
                   Length[x]
```

`pair` takes two lists (of equal length) and pairs elements at the same position, returning a list of pairs.

```
pair[x_, y_] := Transpose[{x, y}]
```

`randomSample` is a wrapper for `RandomSample` that returns a random sample of some size from a list, reducing the sample size if it is greater than the length of the list.

```
randomSample[list_, m_] :=
 RandomSample[list, Min[m, Length[list]]]
```

`twoMin` takes a list and returns the positions of the smallest and second-smallest elements, in that order.

```
twoMin[z_] := Ordering[z, 2]
```

◻ 7.2. Effective Photon Number

The three-argument version of `getNPhotonEff` takes a list of observations, a list of photon numbers associated with clusters, and a list of cluster means. It returns a list specifying the effective photon number of each observation, found via a linear interpolation between the two cluster means closest to a given observation.

`alphaByClustPair` is a list that gives the value of the interpolation parameter α for each observation, organized by the two cluster means to which each observation is closest. `getNPhotonEff` (the two-argument version) takes that information and finds the effective photon number for each observation.

```
getNPhotonEff[dat_, nPhotonList_, clustMeanTrace_] :=
 Module[{alphaByClustPair},
  alphaByClustPair = getAlphaByClustPair[dat,
    clustMeanTrace];
  getNPhotonEff[alphaByClustPair, nPhotonList]
 ]
```

`getAlphaByClustPair` returns a list of ordered triples that give the cluster means closest to a group of observations and the alpha values for those observations. `msDev⟦i, j⟧` is the mean square deviation of observation `i` to cluster mean `j`, and `twoMinIndex⟦i⟧` is a pair of indices giving the two cluster means with the lowest mean square deviations to observation `i` (i.e. a pair of indices giving the two columns with the smallest elements in row `i` of `msDev`). `iObsTwoMinIndex` pairs the relevant observation index with each pair in `twoMinIndex`. In the next line, the `GatherBy` groups the elements of `iObsTwoMinIndex` by their lowest-deviation index pairs, so that each element of `iObsTwoMinIndexGroup` is a list of elements of `iObsTwoMinIndex` that all have the same pair of nearest cluster means. Mapping `groupCommon` onto each of these lists in `iObsTwoMinIndexGroup` converts them into a more useful form; each element of `twoMinIndexIObsList` is now a list of length two, with the first entry being a pair of nearest-mean indices and the second being a list of observation indices whose nearest two means correspond to the pair in the first entry. The output is a list with an entry for each distinct ordered pair of nearest means. Each entry contains a pair of nearest-mean indices, a list of observations whose nearest means are those two, and a list giving the alpha value for each of those observations.

```
getAlphaByClustPair[dat_, idealTraceList_] :=
 Module[{iObsTwoMinIndex, iObsTwoMinIndexGroup, msDev,
   nClustPair, nObs, twoMinIndex, twoMinIndexIObsList},
  nObs = Length[dat];
  msDev = getSquareDiff[dat, idealTraceList];
  twoMinIndex = Map[twoMin, msDev];
  iObsTwoMinIndex = pair[Range[nObs], twoMinIndex];
  iObsTwoMinIndexGroup =
   GatherBy[iObsTwoMinIndex, Last[#] &];
  twoMinIndexIObsList =
   Sort[Map[groupCommon, iObsTwoMinIndexGroup]];
  nClustPair = Length[twoMinIndexIObsList];
  Table[
   {twoMinIndexIObsList⟦ iClustPair, 1 ⟧,
    twoMinIndexIObsList⟦ iClustPair, 2 ⟧,
    getAlpha[dat⟦ twoMinIndexIObsList⟦ iClustPair, 2 ⟧ ⟧,
     idealTraceList⟦ twoMinIndexIObsList⟦ iClustPair, 1 ⟧ ⟧]
   }
   , {iClustPair, nClustPair}]
 ]
```

`getAlpha` takes a list of observations and a pair of ideal waveforms and finds the value of alpha for each observation. Alpha for a waveform $\mathbf{V}_i$ is the value α_i that minimizes the RMS deviations of $(1-\alpha_i)\,\overline{\mathbf{V}}_n + \alpha_i\,\overline{\mathbf{V}}_{n'}$ from $\mathbf{V}_i$, where $\overline{\mathbf{V}}_n$ is the closest mean waveform (the first entry in the second argument of `getAlpha`) and $\overline{\mathbf{V}}_{n'}$ is the second closest. `getAlpha` finds the minimizing value $\alpha_i = \frac{\left(\overline{\mathbf{V}}_n - \mathbf{V}_i\right)\cdot\left(\overline{\mathbf{V}}_n - \overline{\mathbf{V}}_{n'}\right)}{\|\overline{\mathbf{V}}_n - \overline{\mathbf{V}}_{n'}\|^2}$ from equation (5) for each observation in `traces`. A value of 0 indicates perfect agreement with the first ideal trace, and a value of 1 indicates perfect agreement with the second.

```
getAlpha[traces_, idealTraceAB_] :=
 Module[{alpha, diffIdealTrace, idealTraceA, nObs},
  nObs = Length[traces];
  diffIdealTrace = Apply[Subtract, idealTraceAB];
  idealTraceA = First[idealTraceAB];
  alpha = getDotToIdeal[
    Table[idealTraceA - traces⟦iObs⟧, {iObs, nObs}],
    diffIdealTrace]
 ]
```

The two-argument `getNPhotonEff` takes the formatted list of triples `alphaByClustPair` and a list of cluster photon numbers and determines the effective photon number of each observation. `alpha` is a list of alpha values for the observations, `iObs` is a list of observation indices corresponding to those alpha values, and `clustPair` is a list of nearest-mean-index pairs for those observations. `alpha` and `clustPair` are in the same order as `iObs`, but we want them in the same order as the waveforms from the original dataset, so we reorder them with `iiObs`. `nPhotonPair` takes the cluster indices in `clustPair` and converts them to the actual photon numbers associated with those clusters. The output is a list specifying the effective photon number for each observation, derived from a linear interpolation between the closest (`nPhotonPair⟦All, 1⟧`) and second-closest (`nPhotonPair⟦All, 2⟧`) photon numbers.

```
getNPhotonEff[alphaByClustPair_, nPhotonList_] :=
 Module[{alpha, clustPair, iiObs, iObs, nObs, nPhotonPair},
  alpha = Flatten[alphaByClustPair⟦All, 3⟧];
  iObs = Flatten[alphaByClustPair⟦All, 2⟧];
  clustPair = Flatten[
    Table[
     Table[alphaByClustPair⟦iClustPair, 1⟧,
      {Length[alphaByClustPair⟦iClustPair, 3⟧]}
     ],
     {iClustPair, Length[alphaByClustPair]}
    ],
    1
   ];
  iiObs = Ordering[iObs];
  alpha = alpha ⟦ iiObs ⟧;
  clustPair = clustPair ⟦ iiObs ⟧;
  nObs = Length[alpha];
  nPhotonPair = Table[nPhotonList ⟦ clustPair⟦ jObs ⟧ ⟧,
    {jObs, nObs}];
  (1 - alpha) * nPhotonPair⟦All, 1⟧ +
   alpha * nPhotonPair⟦All, 2⟧
 ]
```

7.3. Background Rejection

If background rejection is enabled, getIObsKeep returns the indices of the observations that should stay in the dataset, based on the preset parameters for rejection. peakPosList and peakValList are lists specifying the position and value of the maximum in each waveform in dat. getIObsDrop returns the indices of the waveforms that should be removed because their peak positions and values exceed their respective cutoff values, and getIObsDropA lists those whose endpoints are greater than the value cutoff, indicating that the sensor was registering a background photon at the beginning or end of the pulse. These two lists to remove are combined into iObsDrop. Each element of datPeakLoc lists the positions of the local maxima of a waveform in dat, datPeakLocVal pairs each of those positions with the waveform's value there, and datPeakLocValBig reduces datPeakLocVal to the maxima that exceed the peak value cutoff. datPeakNum gives the number of maxima in each waveform, and getIObsKeepA returns the indices that should stay in the dataset (i.e. those that are not in iObsDrop and that have fewer maxima than peakNumCut).

```
getIObsKeep[dat_, peakPosCut_, peakValCut_, peakNumCut_] :=
 Module[{datPeakLoc, iObsDropA, datPeakLocVal,
   datPeakLocValBig, datPeakNum, iObsDrop, iObsKeep,
   peakPosList, peakValList},
  {peakPosList, peakValList} = getMaxAndPosList[dat];
  (* peak position and value for each obs *)
  iObsDrop = getIObsDrop[peakPosList, peakValList,
    peakPosCut, peakValCut];
  iObsDropA = getIObsDropA[dat, peakValCut];
  iObsDrop = Union[iObsDrop, iObsDropA];
  datPeakLoc = getDatPeakLoc[dat];
  datPeakLocVal = getDatPeakLocVal[datPeakLoc, dat];
  datPeakLocValBig = getDatPeakLocValBig[datPeakLocVal,
    peakValCut];
  datPeakNum =
   Map[First, Map[Dimensions, datPeakLocValBig]];
  iObsKeep = getIObsKeepA[datPeakNum, iObsDrop, peakNumCut]
 ]
```

posMax returns the position of the first occurrence of a list's maximum.

```
posMax[z_] := First[First[Position[z, Max[z]]]]
```

getMaxAndPosList returns two lists, the first giving the position of each waveform's maximum and the second giving the maxima themselves.

```
getMaxAndPosList[dat_] :=
 Module[{peakPosList, peakValList},
  peakPosList = Map[posMax, dat];
  peakValList = Map[Max, dat];
  {peakPosList, peakValList}
 ]
```

`getIObsDrop` returns the indices of the waveforms whose peaks have values and positions greater than the cutoffs.

```
getIObsDrop[peakPosList_, peakValList_, peakPosCut_,
   peakValCut_] :=
 Module[{iObsDrop, nObs},
  nObs = Length[peakPosList];
  iObsDrop = Pick[Range[nObs],
    MapThread[And, {Thread[peakValList >= peakValCut],
      Thread[peakPosList >= peakPosCut]}]
    ]
 ]
```

`getIObsDropA` returns the indices of the waveforms whose endpoints have values greater than the cutoff, indicating that a background photon may have registered with the sensor at the beginning or end of the pulse.

```
getIObsDropA[dat_, peakValCut_] :=
 Module[{nObs},
  nObs = Length[dat];
  Pick[Range[nObs], MapThread[Or,
    {Thread[dat⟦All, +1⟧ >= peakValCut],
     Thread[dat⟦All, -1⟧ >= peakValCut]}]
  ]
 ]
```

`peakList` finds the local maxima of a list by comparing each element to the ones before and after it.

```
peakList[f_] :=
 Pick[Range[Length[f]],
  MapThread[And,
   {Thread[f >= RotateRight[f]],
    Thread[f > RotateLeft[f]]}]
 ]
```

`getDatPeakLoc` takes a list of waveforms and returns the locations of the maxima for each.

```
getDatPeakLoc[dat_] := Map[peakList, dat]
```

`getDatPeakLocVal` pairs each of the locations returned by `getDatPeakLoc` with the value of the corresponding waveform at that point.

```
getDatPeakLocVal[datPeakLoc_, dat_] :=
 Table[
  pair[datPeakLoc⟦iObs⟧, dat⟦ iObs, datPeakLoc⟦iObs⟧ ⟧],
  {iObs, Length[dat]}]
```

getDatPeakLocValBig finds the maxima that exceed the cutoff among those returned by getDatPeakLocVal.

```
getDatPeakLocValBig[datPeakLocVal_, peakValCut_] :=
 Table[
  Select[datPeakLocVal⟦iObs⟧, Last[#] >= peakValCut &],
  {iObs, Length[dat]}]
```

getIObsKeepA takes the list of waveforms rejected because of misplaced peaks and too-large endpoints and combines that with the information on each waveform's number of maxima to return a list of the indices of all of the waveforms that should be rejected due to background radiation.

```
getIObsKeepA[datPeakNum_, iObsDrop_, datPeakNumCut_] :=
 Complement[
  Pick[Range[Length[datPeakNum]],
   Thread[datPeakNum < datPeakNumCut]],
  iObsDrop]
```

7.4. Cluster Organization

These two functions convert between iObsOfClust and iClustOfObs, two different ways of organizing clusters of waveforms. iObsOfClust is a list of lists. Each sublist is associated with a cluster and has the indices (in dat) of the traces in that cluster. iClustOfObs is a simple list, each of whose entries corresponds to a trace and states what cluster that trace is in (0 if the trace is not in any cluster). getIClustOfObs takes iObsOfClust and returns iClustOfObs, and getIObsOfClust is its inverse function.

getIClustOfObs creates a table with an entry for each observation and then iterates through the clusters. In the table, it assigns the contents of each cluster to the appropriate cluster number.

```
getIClustOfObs[iObsOfClust_] :=
 Module[{iClustOfObs, nClust, nObs},
  nClust = Length[iObsOfClust];
  nObs = Max[Flatten[iObsOfClust]];
  iClustOfObs = Table[0, {nObs}];
  Do[iClustOfObs⟦iObsOfClust⟦iClust⟧⟧ = iClust,
   {iClust, nClust}];
  iClustOfObs
 ]
```

getIObsOfClust creates a list of observation number/cluster number pairs and then applies a GatherBy to group waveforms in the same clusters together. Mapping groupCommon onto the list creates a list of lists, each of whose first element is a cluster number and each of whose second element is a list of the indices of the waveforms in that cluster. We then sort the overall list by cluster number and extract the second entries in the sublists to create a list of lists of observation indices.

```
getIObsOfClust[iClustOfObs_] :=
 Module[{iObsGroup},
  iObsGroup =
   GatherBy[
    Transpose[{Range[Length[iClustOfObs]], iClustOfObs}],
    Last[#] &];
  Sort[Map[groupCommon, iObsGroup]] ⟦All, 2⟧
 ]
```

groupProbability takes the data, the hypothesized mean photon number for the current round, and the effective-photon-number ordering of the data, and returns a list of four things: the Poisson probability distribution (ordered pairs of photon numbers and probabilities), a list of photon numbers for the clustering it creates, a list of lists of constituent waveforms' indices for the clusters (iObsOfClust), and a list containing the average waveform in each cluster. groupProbability finds the photon number/value pairs for the Poisson distribution with the given mean photon number and then calls groupProb to organize the clusters.

```
groupProbability[dat_, nPhotonAvg_, nPhotonEff_, nSigma_] :=
 Module[{prob},
  prob = poisson[nPhotonAvg, nSigma];
  Prepend[groupProb[dat, prob, nPhotonEff], prob]
 ]
```

groupProb processes the data into clusters based on the given effective photon number list and the Poisson distribution. First we find the proper ordering of the effective photon numbers by size (nPhotonEffSortIndex); this will be needed once we have determined the sizes of the clusters to be created. The probability distribution prob is broken up into its component parts: nPhoton lists photon numbers, and probList lists their corresponding probabilities. probCum is the cumulative distribution function of probList, which we normalize to ensure that the final element is one. probIn⋱ dexStart⟦i⟧ gives the index (in the sorted list of effective photon numbers, nPhotonEffSortIndex) where cluster i should start; probIndexStop⟦i⟧ gives the index where it should stop. Both indices are inclusive, which is why prob⋱ IndexStart is offset by one from probIndexStop. (boundList ensures that the starting and stopping indices stay within the confines of the number of observations available to group.) keep weeds out the zero-length bins, and we use it to reduce nPhoton, probIndexStart, and probIndexStop down to only the nontrivial clusters. We then form iObsOfClust by finding the range of *sorted* effective photon numbers that each cluster should encompass and then using nPhotonEffSortIndex to find the observation numbers to which that range corresponds. clustMeanTrace is formed by taking the waveforms in dat of each cluster in iObsOfClust and averaging them. We then return nPhotonUse (the photon numbers of the nonempty clusters), iObsOfClust, and clustMeanTrace.

```
groupProb[dat_, prob_, nPhotonEff_] :=
 Module[{nPhotonEffSortIndex, keep, nObs, nPhoton,
   nPhotonUse, probCum, probIndex, probIndexStart,
   probIndexStop, probList, probNPhotonUse,
   probIndexStartUse, probIndexStopUse},
  nPhotonEffSortIndex = Ordering[nPhotonEff];
  {nPhoton, probList} = Transpose[prob];
  nObs = Length[dat];
  probCum = Prepend[Accumulate[probList], 0];
  probCum /= Last[probCum];
  probIndexStart = boundList[1 + Round[Most[nObs * probCum]],
    1, nObs + 1];
  probIndexStop = boundList[Round[Rest[nObs * probCum]],
    0, nObs];
  keep = Thread[probIndexStart <= probIndexStop];
  (* no zero length bins *)
  nPhotonUse = Pick[nPhoton, keep];
  probIndexStartUse = Pick[probIndexStart, keep];
  probIndexStopUse = Pick[probIndexStop, keep];
  nClust = Length[probIndexStartUse];
  iObsOfClust = Table[nPhotonEffSortIndex⟦
      Range[probIndexStartUse⟦iClust⟧,
       probIndexStopUse⟦iClust⟧]
     ⟧,
    {iClust, nClust}];
  clustMeanTrace = Table[
    getMeanOfEachTime[dat⟦ iObsOfClust⟦iClust⟧ ⟧],
    {iClust, nClust}];
  {nPhotonUse, iObsOfClust, clustMeanTrace}
 ]
```

7.5. Probability and Log-Likelihood

This is the combinatorial term of the log-likelihood objective function. From equation (15) above, $\ln \mathcal{L}_C = \ln M! - \sum \ln m_n!$, where the m_i are the cluster sizes and $M = \sum m_n$.

```
comboLogLikelihood[freqList_] :=
 LogGamma[Total[freqList] + 1.] -
  Total[LogGamma[freqList + 1.]]
```

The Poisson log-likelihood term is $\ln \mathcal{L}_P = -\mu \mathrm{M} + \sum m_n [n \ln \mu - \ln(n!)]$.

```
poissonLogLikelihood[nList_, freqList_, mu_] :=
 -mu * Total[freqList] + nList.freqList * Log[mu] -
  freqList .LogGamma[nList + 1.]
```

`probComboLogLikelihood` finds the Poisson/combinatorial log-likelihood term of the objective function.

```
probComboLogLikelihood[nList_, freqList_, mu_] :=
 poissonLogLikelihood[nList, freqList, mu] +
  comboLogLikelihood[freqList]
```

`poisson` generates about `nSigma` standard deviations of a Poisson distribution on either side of the given mean. The output is a list of ordered pairs of photon numbers and associated values of the Poisson probability mass function. The output is normalized so that it sums to one.

```
poisson[mu_, nSigma_] := Module[
  {sigma, tab},
  sigma = Sqrt[mu];
  tab = Table[{n, (N[Exp[-mu]] * mu^n) / n!},
    {n, Max[Floor[mu - sigma * nSigma], 0],
     Ceiling[mu + sigma * nSigma]}];
  Map[{First[#], Last[#] / Total[tab[[All, 2]]]} &, tab]
]
```

◻ 7.6. Deviation and Objective Function Measurement

`getSqDevClustObs` takes a list of observation traces and a list of cluster mean traces and returns a table whose element in position `⟦i, j⟧` is the mean square deviation of observation `j` from the mean trace of cluster `i`.

```
getSqDevClustObs[dat_, clustMeanTrace_] :=
 Table[
  meanSquare[dat⟦iObs⟧ - clustMeanTrace⟦jClust⟧],
  {jClust, Length[clustMeanTrace]}, {iObs, Length[dat]}
 ]
```

`getSqDevClustClust` returns a table whose element in position `⟦i, j⟧` is the average of the mean square deviations of the traces in cluster `i` from the mean trace of cluster `j`.

```
getSqDevClustClust[dat_, iObsOfClust_, clustMeanTrace_] :=
 Map[Mean,
  Table[
   getSqDevClustObs[dat⟦ iObsOfClust⟦ iClust ⟧ ⟧,
    clustMeanTrace],
   {iClust, Length[iObsOfClust]}
  ],
  {2}
 ]
```

getObjFtn takes a list of each cluster's square deviation from its mean, the constant σ relating the K-means and probabilistic components of the objective function (which can be a scalar or a list that takes on a different value for each cluster), and the log-likelihood of the Poisson distribution (with the combinatorial term included). It returns the value of the objective function.

```
getObjFtn[sqDevOfClust_, sigmaObjFtn_, logLikeProb_] :=
 Total[sqDevOfClust / (2 * sigmaObjFtn^2)] - logLikeProb
```

getSqDevInClust takes a list of observations and a mean waveform and totals up the mean square deviations of the observations to the mean. (It returns $J_n = \sum_{i \in C_n} \frac{1}{N_t} \sum \left[\mathbf{V}_i(t) - \overline{\mathbf{V}}_n(t)\right]^2$, where C_n is the list of observations, N_t is the number of time points, the $\mathbf{V}_i$ are the observations, and $\overline{\mathbf{V}}_n$ is the mean.)

```
getSqDevInClust[datInClust_, meanTraceInClust_] :=
 Total[
  Table[meanSquare[datInClust[[jObs]] - meanTraceInClust],
   {jObs, Length[datInClust]}]
 ]
```

getSqDevInClust returns a list that gives J_n, the total mean square deviation of a cluster to its mean, for each cluster n. The list returned becomes the first argument of getObjFtn.

```
getSqDevOfClust[dat_, iObsOfClust_, clustMeanTrace_] :=
 Table[
  getSqDevInClust[dat[[ iObsOfClust[[ iClust ]] ]],
   clustMeanTrace[[ iClust ]]],
  {iClust, Length[clustMeanTrace]}]
```

7.7. Reading and Filtering Data

readTES reads in the data from a set of files specified in the options, returning dat, a table whose rows give the values of particular waveforms at regular time intervals. readTES assumes that several different datasets may share a directory and that each dataset is split over some number of files, each of which consists of an equal number of unsigned 16-bit integers concatenated together in string form. iNPhoton gives the numeric label of the dataset to read (it is not equal to the mean photon number of the dataset). iDataSet lists the indices of the particular files in the dataset that should be read in. fileInfo is a list of three things: fileNamePart, a list of two strings that combine with the dataset label and the file index to create the whole filename; nSample‐PerTrace, the number of time points in each waveform; and nTracePerFile, the number of waveforms in each file of the dataset.

For example, if `iNPhoton` were 7, `iDataSet` were `{2, 6, 20}`, and `fileInfo` were `{{"[directory]/TES ", ".daq "}, 200, 512}`, then we would be looking for the files `TES7.daq02`, `TES7.daq06`, and `TES7.daq20` in `[directory]`, with 200 time points in each waveform and 512 waveforms in each of the three files.

For each file, we import the data as a list of samples, organize the samples into sublists (waveforms) of length `nSamplePerTrace`, and assign the list of waveforms to the appropriate section of `dat`. When all of the files have been read, `dat` is full and properly formatted.

```
readTES[iNPhoton_, iDataSet_, fileInfo_] :=
 Module[{fileNameA, nSamplePerTrace, nFile, fileNamePart,
   nTracePerFile, nTrace, dat, serialNumber, fileName,
   dim, datImport},
  {fileNamePart, nSamplePerTrace, nTracePerFile} = fileInfo;
  fileNameA = fileNamePart⟦1⟧ <> IntegerString[iNPhoton] <>
    fileNamePart⟦2⟧;
  nFile = Length[iDataSet];
  nTrace = nTracePerFile * nFile;
  dat = ConstantArray[{}, nTrace];
  Do[
   serialNumber = IntegerString[iDataSet⟦iFile⟧, 10, 2];
   (* padding to ensure 2 digits *)
   fileName = fileNameA <> serialNumber;
   If[Not[FileExistsQ[fileName]],
    Print["readTESdata: " <> fileName <> " does not exist"];
    Abort[]
   ];
   datImport =
    Partition[Import[fileName, "UnsignedInteger16"],
     nSamplePerTrace];
   dat⟦(iFile - 1) * nTracePerFile + 1 ;; iFile * nTracePerFile⟧ =
    datImport,
   {iFile, nFile}
  ];
  dim = Dimensions[dat];
  If[dim ≠ {nTrace, nSamplePerTrace},
   Print["readTESdata: unexpected values for dim", dim];
   Print["readTESdata:  expected ",
    {nTrace, nSamplePerTrace}];
   Abort[]
  ];
  dat
 ]
```

We can also filter the data as we read it in. `readTES` applies a "short" filter to a list, one that omits half of the frequency domain.

```
applyFilterShort[x_, filterShort_] :=
 Module[{ft, ftShort, nFT},
  ft = Fourier[x, FourierParameters → {1, 1}];
  nFT = Length[ft];
  ftShort = Flatten[
    {Take[ft, nFT / 4],
     0,
     Take[ft, {(3 / 4) * nFT + 2, nFT}]}
   ];
  Chop[InverseFourier[ftShort * filterShort,
     FourierParameters → {1, 1}] / 2]
 ]
```

This generates a Hanning filter of a given length.

```
hanningFilter[n_] := Chop[Cos[(N[Pi] * Range[0, n - 1]) / n]^2]
```

`readTESandFilter` reads in the dataset, applying a filter to each file's data before it is stored. This halves the memory consumption of the reading process, since filtration reduces the amount of information to store by half.

```
readTESandFilter[iNPhoton_, iDataSet_, nTime_, fileInfo_] :=
 Module[{datUse, dat, datFilt, nTimeUse, nSamplePerTrace,
   nTracePerFile, nFile},
  {nSamplePerTrace, nTracePerFile} = fileInfo⟦{2, 3}⟧;
  nFile = Length[iDataSet];
  datUse = ConstantArray[{}, nTracePerFile * nFile];
  Do[
   dat = readTES[iNPhoton, {iDataSet⟦iFile⟧}, fileInfo];
   hanning = hanningFilter[Last[Dimensions[dat]] / 2];
   datFilt = Map[applyFilterShort[#, hanning] &, dat];
   nTimeUse = Min[nTime, Length[hanning]];
   datUse⟦(iFile - 1) * nTracePerFile + 1 ;;
      iFile * nTracePerFile⟧ = datFilt⟦ All, Range[nTimeUse] ⟧
   ,
   {iFile, nFile}
  ];
  datUse -= Mean[datUse⟦All, 10⟧]
  (* zeroes the signal at an early point,
  before any photons have arrived *)
 ]
```

7.8. Output Organization

`outputCreateTabs` creates several list structures (global variables) of the appropriate dimensions to accommodate output from the specified number of runs of PIKA. The output is organized into a `TabView`, with each run receiving a tab that contains a nested view with graphical and textual output. `outputNameList` stores the names of the labels in the subview (which are the same for all runs), and `outputContentTable` stores the output content for each run and sublabel. (Content is stored as a list that later becomes a `Column`.) Each run's tab also has a space above its subview for other information, which is stored in `outputTopList`. Finally, there is another tab on the level of the runs with general textual (`outputGeneralLog`) and graphical (`outputGeneralGraphics`) information about the data read in.

```
outputCreateTabs[nRuns_] := (
  outputContentTable = ConstantArray[{}, nRuns];
  outputNameList = {};
  outputTopList = ConstantArray[{}, nRuns];
  outputGeneralLog = {};
  outputGeneralGraphics = {};
 )
```

`outputAdd` adds some expression to the content list for a particular name and run, creating the name's content list if it has no content already. A run number of 0 indicates output to the general information tab, and a blank name indicates output to the top space of a run's tab.

```
outputAdd[expr_, iRun_, name_] := Module[{iName},
  If[iRun == 0,
   If[name == "Log",
    outputGeneralLog = Append[outputGeneralLog, expr];
    ,
    outputGeneralGraphics =
      Append[outputGeneralGraphics, expr];
   ]
   ,
   If[name == "",
    outputTopList[[iRun]] = Append[outputTopList[[iRun]], expr];
    ,
    If[MemberQ[outputNameList, name],
     iName = First[First[Position[outputNameList, name]]];
     outputContentTable[[iRun, iName]] =
      Append[outputContentTable[[iRun, iName]], expr];
     ,
     outputNameList = Append[outputNameList, name];
     outputContentTable =
      Map[Append[{}], outputContentTable];
     outputContentTable[[iRun, -1]] = {expr};
    ]
   ]
  ]
 ] (* end Module *)
```

`outputShowTabView` is called at the end of PIKA, taking the name and content lists and formatting them into a nested `TabView`.

```
outputShowTabView[] :=
 Module[{columnsNames, labelTableNames, viewsRuns,
   columnsRuns, labelTableRuns, generalTabView,
   labelTableRunsGeneral},
  columnsNames = Map[Column, outputContentTable, {2}];
  labelTableNames =
   Table[outputNameList⟦jName⟧ → columnsNames⟦jRun, jName⟧,
    {jRun, Length[outputContentTable]},
    {jName, Length[outputNameList]}];
  viewsRuns =
   Map[TabView[#, Appearance → {"Limited", 5},
      ImageSize → Automatic] &, labelTableNames];
  columnsRuns =
   Map[Column, MapThread[Append,
     {outputTopList, viewsRuns}]];
  labelTableRuns =
   Table["Run " <> ToString[jRun] → columnsRuns⟦jRun⟧,
    {jRun, Length[outputContentTable]}];
  generalTabView =
   TabView[{"Log" → Column[outputGeneralLog],
     "Graphics" → Column[outputGeneralGraphics],
     "Options" →
      TableForm[
       Table[{varNames⟦i⟧, If[varNames⟦i⟧ == "tAnneal",
          tAnnealOrig, ToExpression[varNames⟦i⟧]]},
        {i, Length[varNames]}]]},
    Appearance → {"Limited", 15}, ImageSize → Automatic];
  labelTableRunsGeneral =
   Prepend[labelTableRuns, "General" → generalTabView];
  TabView[labelTableRunsGeneral,
   Appearance → {"Limited", 15}, ImageSize → Automatic]
 ]
```

`print` takes any number of arguments (`args`), turns them into strings, concatenates them, and prints the result to the log. `printSp` prints to an arbitrary section of output, not just the log. `iRun` and `name` indicate the run number and tab name, respectively, to which to print.

```
print[iRun_, args__] :=
 outputAdd[OutputForm[StringJoin[Map[ToString, {args}]]],
  iRun, "Log"]
```

```
printSp[iRun_, name_, args__] :=
 outputAdd[OutputForm[StringJoin[Map[ToString, {args}]]],
  iRun, name]
```

7.9. Form Input and Runner

pika creates a form with a field for each constant and option that needs to be set for PIKA to run. It then uses runFromAssociation to assign the proper values to the proper variables and run PIKA.

```
pika[] := Module[{form},
  form = FormObject[{
     Style["General", Bold],
     makeField["iNPhoton", "Integer",
      "Index of the dataset to read in", 16],
     makeField["iDataSet", ToExpression,
      "Parts of dataset to use", {0, 1, 2, 3},
      "Help" →
       "Normal expressions (e.g., Range[0,3]) will
         work here"],
     makeField["nTime", "Integer",
      "Number of time points after filtration", 512],
     makeField["mSample", "Integer",
      "Number of traces to randomly sample", 100,
      "Help" →
       "Actually a maximum number to sample, only
         reached if the population is large
         enough"],
     makeField["nDatUse", "Integer",
      "Number of traces to use (if 0, use all)", 0],
     Style["\nBackground Radiation Rejection", Bold],
     makeField["backgroundReject", "Boolean",
      "Reject background traces", False],
     makeField["peakValCut", "Integer", "Voltage cutoff",
      275],
     makeField["peakPosCut", "Integer", "Time cutoff", 17],
     makeField["peakNumCut", "Integer",
      "Cutoff for number of peaks", 2],
     Style["\nIteration and Simulated Annealing", Bold],
     makeField["nCool", "Integer",
      "Total number of optimization rounds", 60,
      "Help" →
       "Includes both greedy and simulated annealing
         rounds"],
     makeField["nGreedy", "Integer",
      "Number of greedy rounds", 60,
      "Help" →
       "Greedy rounds come before simulated annealing;
         simulated annealing will not run
         if nCool==nGreedy"],
     makeField["coolConst", "Number",
      "Simulated annealing cooling constant", 0.99],
     makeField["tAnneal", "Number",
      "Simulated annealing starting temperature", 0.02],
     Style["\nProbability", Bold],
     makeField["probDistName", "String",
      "Name of the probability distribution", "Poisson"],
     makeField["nSigma", "Integer",
      "Extent of probability distribution", 10,
      "Help" →
       "Number of standard deviations away from the
         mean to generate"],
     Style["\nOutput", Bold],
     makeField["binFract", "Number",
      "Histogram bin width (fraction of a photon)", 0.05],
     makeField["outputImageSize", "Integer",
      "Default image size for plots", 400],
     Style["\nInput", Bold],
     makeField["nPhotonAvgList", ToExpression,
      "List of mean photon numbers to test", {1.49608}],
     "The data filenames should be of the form
       data-1.daq01, with the \"data-\" and
       \".daq\" being an arbitrary prefix and
       file extension, the first number being
       iNPhoton, and the second number being
       a two-digit form of iDataSet.",
     makeField["partialFilePath", "String",
      "Path to data files", "[directory]\\TES_E0019-",
      "Help" →
       "Should include the first (generic) part of
         the filenames of the data files,
         without iNPhoton or iDataSet
         specifications"],
     makeField["fileExt", "String",
      "File extension of data files", ".daq",
      "Help" →
       "Should not include iDataSet specification"],
     makeField["nSamplePerTrace", "Integer",
      "Number of samples per trace", 8192],
     makeField["nTracePerFile", "Integer",
      "Number of traces per file", 512],
     makeField["useInputFile", "Boolean",
      "Use separate input file", False],
     makeField["pikaInput", "String", "Separate input file",
      "", "Control" →
       (FileNameSetter[#, "Open",
          {"Mathematica packages" → {"*.m"}}] &),
      "Required" → False]
    }];
  FormFunction[form, runFromAssociation][]
 ]
```

makeField creates a rule that, when used in a FormObject, generates a field to set the variable with name name and type type to the entered value, with descriptive text labelPart and default value defaultContents. args takes optional extra arguments that specify additional options for the field.

```
makeField[name_, type_, labelPart_, defaultContents_,
  args___] := name → <|
   "Interpreter" → type,
   "Label" → Overlay[{name, "\t\t  |   " <> labelPart}],
   "Input" → defaultContents,
   args
  |>
```

runFromAssociation takes an association between variable names (strings) and the values that should be assigned to them and assigns the proper value to the symbolic variable associated with each name. It then runs PIKA (using the main function from Section 3) with those global variables set.

```
runFromAssociation[formAssoc_] := (
  varNames = Keys[formAssoc];
  varValues = Values[formAssoc];
  Do[
   Clear[Evaluate[varNames⟦iVar⟧]];
   Evaluate[ToExpression[varNames⟦iVar⟧]] =
    varValues⟦iVar⟧,
   {iVar, Length[varNames]}
  ];
  runPIKA[]
 )
```

References

[1] Z. H. Levine, T. Gerrits, A. L. Migdall, D. V. Samarov, B. Calkins, A. E. Lita, and S. W. Nam, "Algorithm for Finding Clusters with a Known Distribution and Its Application to Photon-Number Resolution Using a Superconducting Transition-Edge Sensor," *Journal of the Optical Society of America B*, **29**(8), 2012 pp. 2066–2073. doi:10.1364/JOSAB.29.002066.

[2] T. Gerrits, B. Calkins, N. Tomlin, A. E. Lita, A. Migdall, R. Mirin, and S. W. Nam, "Extending Single-Photon Optimized Superconducting Transition Edge Sensors beyond the Single-Photon Counting Regime," *Optics Express*, **20**(21), 2021 pp. 23798–23810. doi:10.1364/OE.20.023798.

[3] Z. H. Levine, B. L. Glebov, A. L. Pintar, and A. L. Migdall, "Absolute Calibration of a Variable Attenuator Using Few-Photon Pulses," *Optics Express*, **23**(12), 2015 pp. 16372–16382. doi:10.1364/OE.23.016372.

[4] Z. H. Levine, B. L. Glebov, A. L. Migdall, T. Gerrits, B. Calkins, A. E. Lita, and S. W. Nam, "Photon-Number Uncertainty in a Superconducting Transition Edge Sensor beyond Resolved-Photon-Number Determination," *Journal of the Optical Society of America B*, **31**(10), 2014 pp. B20–B24. doi:10.1364/JOSAB.31.000B20.

[5] P. E. Black, "Greedy Algorithm," in *Dictionary of Algorithms and Data Structures* (V. Pieterse and P. E. Black, eds.), Feb 2, 2005. www.nist.gov/dads/HTML/greedyalgo.html.

[6] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, **220**, 1983 pp. 671–680. doi:10.1126/science.220.4598.671.

B. P. M. Morris and Z. H. Levine, "The Poisson-Influenced K-Means Algorithm," *The Mathematica Journal*, 2016. dx.doi.org/doi:10.3888/tmj.18-3.

Additional Material

1. Dataset 16, $\bar{N} \approx 1.49608$: TES_E0019-16.daq00 through TES_E0019-16.daq03

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-16.daq00.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-16.daq01.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-16.daq02.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-16.daq03.zip

2. Dataset 19, $\bar{N} \approx 2.99213$: TES_E0019-19.daq00 through TES_E0019-19.daq03

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-19.daq00.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-19.daq01.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-19.daq02.zip

www.mathematica-journal.com/data/uploads/2016/03/TES_E0019-19.daq03.zip

About the Authors

Brian P. M. Morris is a senior in the Science, Mathematics, and Computer Science Magnet Program at Montgomery Blair High School in Silver Spring, Maryland, was in the Summer High School Intern Program at the National Institute of Standards and Technology, and is a 2016 Intel Science Talent Search Semifinalist.

Zachary H. Levine is a physicist at the National Institute of Standards and Technology in the Infrared Technology Group and is a Fellow of the American Physical Society. He is a graduate of MIT and the University of Pennsylvania.

Brian P. M. Morris
Montgomery Blair High School
Silver Spring, Maryland 20901

Zachary H. Levine
National Institute of Standards and Technology
Gaithersburg, Maryland 20899-8441
zlevine@nist.gov