Searching for Gravitational-Wave Bursts of Arbitrary Waveform

Author: Rubab Khan (Columbia University)
Mentor: Dr. Shourov K. Chatterji

One class of signal LIGO is searching for consists of short duration gravitational wave bursts of unknown waveform. Potential sources include core collapse supernovae and the coalescence of binary black holes. To detect such events, existing search algorithms project the LIGO data stream onto various time-frequency bases and then search for regions of excess signal energy. One of these search algorithms, the Q Pipeline, determines the statistical significance of events based solely on the peak signal observed in the time-frequency plane. This project investigated extensions to this approach that also consider the statistical significance of arbitrarily shaped regions in the time-frequency plane by exploiting the advantages of data clustering. After considering various aspects of different data clustering methods, density based clustering algorithms were chosen to be the best fit for our purpose due to their ability to find arbitrarily shaped clusters and reject noise. A density based clustering function has been implemented, extensively tested, and integrated with the standard Q pipeline burst search algorithm. We have shown that density based clustering is improving the performance of Q pipeline for arbitrarily shaped non-localized waveforms.

General Relativity predicts that as concentrations of mass-energy rapidly change shape (i.e. supernovae explosion, merger of astronomical binary systems, star-quake etc.) they create dynamically changing space-time ripples that propagates throughout the universe at the speed of light [1,2]. These ripples are called Gravitational Waves and cause extremely weak perturbations of locally flat space-time near earth. Unlike electromagnetic waves which propagate through space-time after being created by the incoherent motion of atoms and molecules and have a wavelength much smaller than their sources, gravitational waves are propagated as perturbations of space-time itself after being created by the coherent motion of massive astronomical sources, and have wavelengths similar to the size of their sources. Most significantly, while electromagnetic waves interact with most objects, the universe is mostly transparent gravitational waves. This ensures that the gravitational waves that reach us have not been tampered with since they were created. However, this also makes their direct observation more difficult [3] since though they carry a lot of energy, that couple very weakly into the detector.

The existence of gravitational waves has been observationally confirmed through the discovery of the binary pulsar system PSR 1913+16 whose orbit is decaying at just the rate predicted by general relativity due to energy loss through gravitational radiation [2]. While the first direct detection of gravitational waves will be very significant, the payoff will come from analyzing the detected waves which will make it possible to extract information about the physics of the extreme conditions where the waves have originated – such as very strong gravity and nuclear densities – information that is not easily accessible or totally inaccessible from electromagnetic observations.

Current efforts of building gravitational wave observatories focus on using interferometry. They include the Laser Interferometer Gravitational-wave Observatory (LIGO; three detectors; two in Hanford, Washington and one in Livingston, Louisiana), Virgo (Pisa, Italy), GEO600 (Hanover, Germany), and TAMA300 (Tokyo, Japan) [3]. Every interferometric detector is basically an extremely low noise kilometer-scale Michelson interferometer having two arms in L-shaped

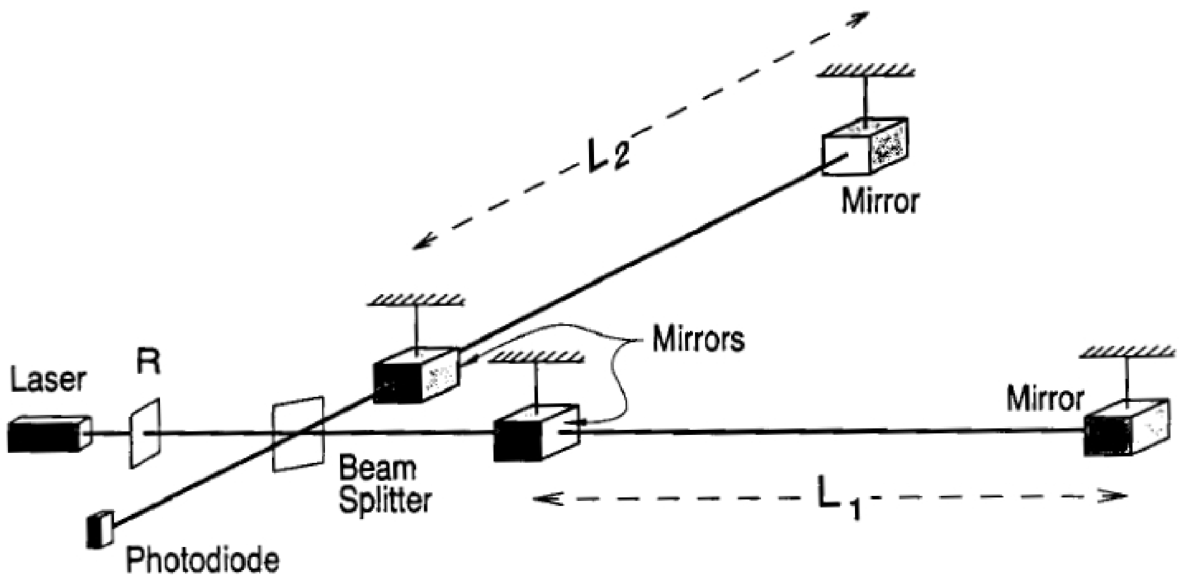

Figure-1: Basic layout of an interferometer for detecting gravitational waves (from [3]).

structure. At the simplest level, it can be imagined that as a gravitational wave passes through the interferometer alternatively stretching one arm while squeezing the other, LIGO measures this change of length and thus records information about the passing waves. The LIGO detectors have now reached their design sensitivity, and currently LIGO is collecting data as it is undergoing its yearlong fifth science run. It is expected that observation of signals at LIGO will occur near the limit of detector sensitivity, and searching for and identifying such small signals in the presence of various detector noise is a daunting task [4].

Sources of gravitational waves are classified into four major groups. The inspiral phase of coalescing compact binaries (i.e. neutron stars or black holes) causes chirp signals. Signals from very short-lived events of which we often do not have sufficient understanding to predict an expected waveform, such as merger phase of binary coalescence, core collapse supernovae, gamma ray bursts, and even unexpected sources, are classified as burst sources. Stochastic signals can be either relic gravitational-waves from the very early universe or the cumulative effect of many low amplitude sources that can give rise to a correlated random noise in multiple detectors. Spinning compact objects (pulsars) can produce periodic signals if they have asymmetric mass distributions [5]. If the waveform of GW burst is known, then matched filtering can be used. Matched filtering forms the projection of the data onto the waveforms that are to be detected, and then looks for times when the projection is large. However, for bursts of unmodeled waveform, the data under test are typically projected onto a convenient basis of abstract waveforms that are chosen to cover a targeted region of signal space, and one looks for large projections.

Q pipeline is one such unmodeled burst search algorithm that is equivalent to matched filtering for waves having sinusoidal-Gaussian waveform. It analyzes the time-frequency signal plane looking for non-overlapping tiles (approximately: pixels) that have higher energy than nearby tiles. Thus it finds the most significant "event" above a certain threshold in a given signal space. It works very well for localized signals that contain most of their energy within narrow enough

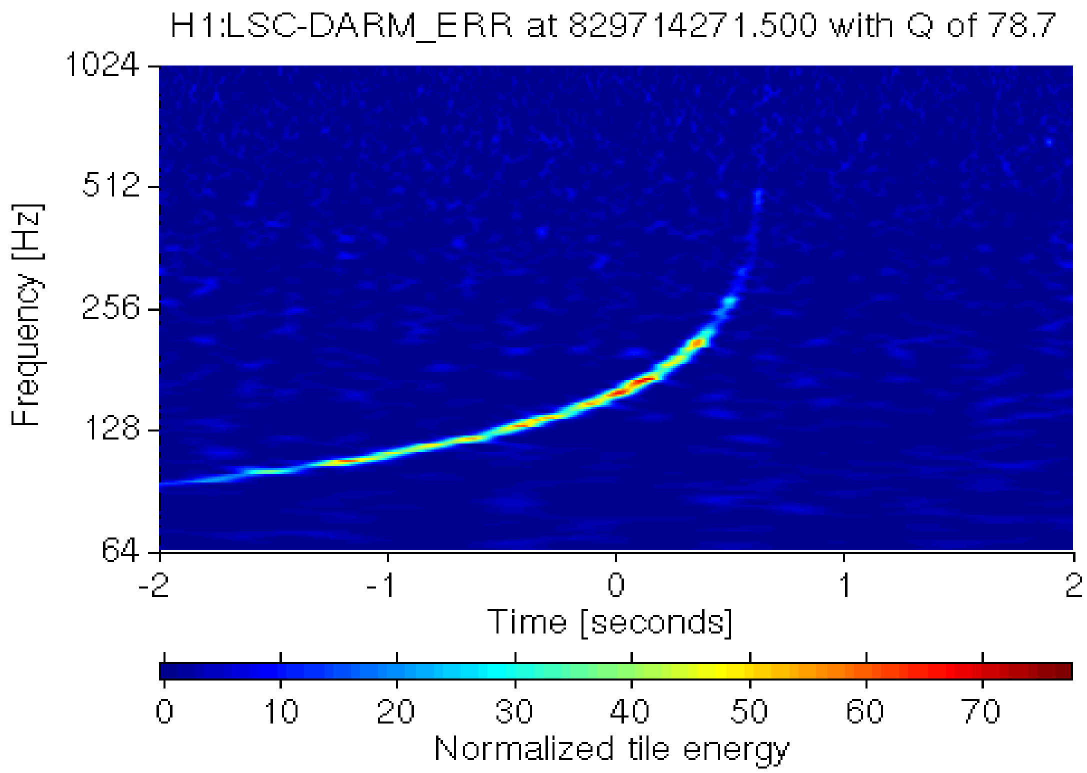

Figure-2: A hardware injection for the inspiral phase of an optimally oriented 1.4, 1.4 solar mass binary neutron star merger at 5 mega-parsecs as seen by the Q pipeline.

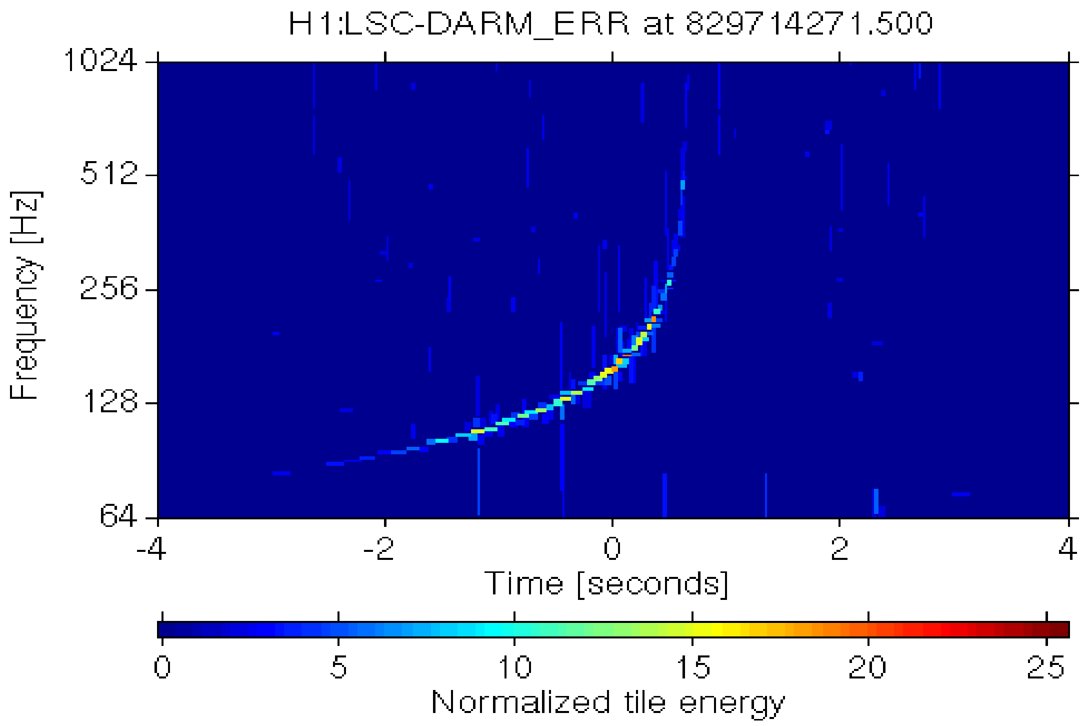

Figure-3: The Q pipeline keeps only the most significant non-overlapping tiles.

frequency bandwidth and short enough time scale to be identified by a single tile. However, for non-localized signal whose energy is distributed across multiple tiles, only identifying the highest energy tile might not be enough as it underestimates the total signal energy. Q pipeline may even miss the signal altogether if there is a large detector glitch nearby since that glitch would then be the most-significant event on the signal space.

Figure-2 shows the time frequency map of a hardware injection (a simulated signal physically injected in the detector for test purposes) of the inspiral phase of a binary neutron star coalescence at 5 mega-parsecs away as seen by the Q pipeline. Figure-3 shows the corresponding time-frequency tiles produced by the Q pipeline for that injection. In this case, the performance of the Q -pipeline is determined by the highest energy tile (the dark red tile at the center of the plot). However, since thr Q pipeline considers each tile as an individual event, it cannot identify the other tiles of the injection as part of the same signal. Also, had there been a nearby glitch with higher energy than the center tile of the injection, the Q pipeline would simply not find the injection.

Application of Clustering with Q Pipeline

Clustering is the method of grouping elements of data in meaningful classes. It is expected that the use of clustering to collect non-localized signal energy would magnify the significance of non-localized signals by clustering together multiple tiles from same signal or glitch, and thus increase the detection efficiency of Q pipeline for such signals. Potential clustering methods include partitioning, hierarchical clustering, and density based clustering. Partitioning approaches, such as the well know kmeans algorithm, was determined not to be useful for finding clusters of arbitrary number and shape. Hierarchical algorithms were tested using preexisting Matlab functions in conjugation with a customized measure of distance between tiles. It was shown that much of the injection could be clustered together as seen in Figure-4. While this shows the potential advantage of clustering, it also produces a lot of noise clusters that make identifying the most significant cluster statistically difficult. Also, some noise is clustered together with the injection cluster, which distorts information about the shape of the injection. After various considerations and testing, density based clustering has been chosen to be ideal for the purpose of this project since it is very efficient in finding arbitrarily shaped regions in the time-frequency signal space. Also, while most other clustering algorithms include noise tiles individually or as part of larger clusters, density based algorithms keep noise, or data points that could not be grouped together with other data points, out of all clusters and clearly identify them as noise.

Density based clustering allows both to search for unknown shapes of signals, and to pick up only significant clusters over a large set of data. Thus it does not clutter the output report with a list of numerous noise clusters that contain one or just a few data points. The algorithm picks a data-point that has enough neighboring data-points near itself; and for all such neighbors that individually has enough neighbors, the algorithm finds their neighbors; and so on; until it reaches a point that doesn't have enough neighbors. Thus, all point that are related through this neighbor

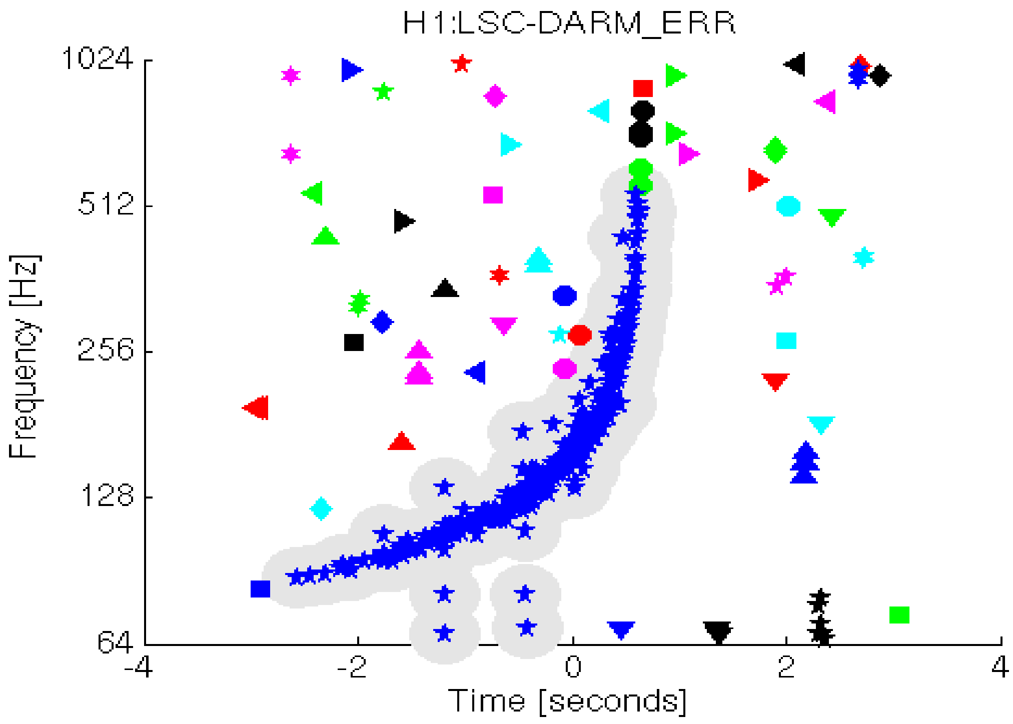

Figure-4: Hierarchical clustering clusters together most of the injection tiles, but also includes some noise tiles in that cluster. Also, a lot of individual noise clusters are produced as well.

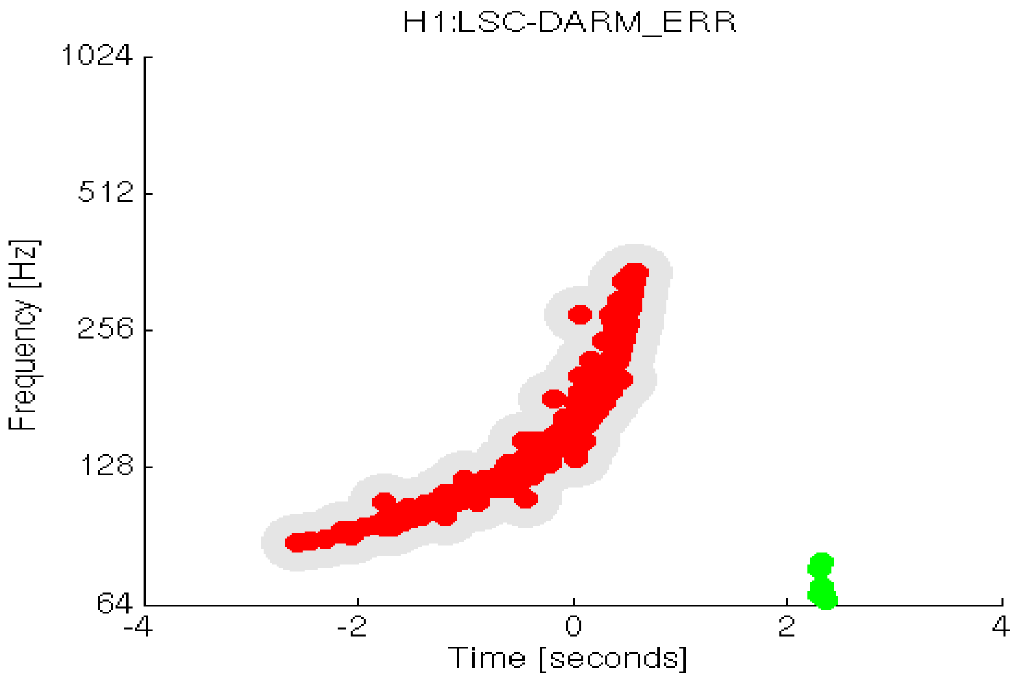

Figure-5: Density based clustering clusters together most of the signal-energy while removing most of the noise.

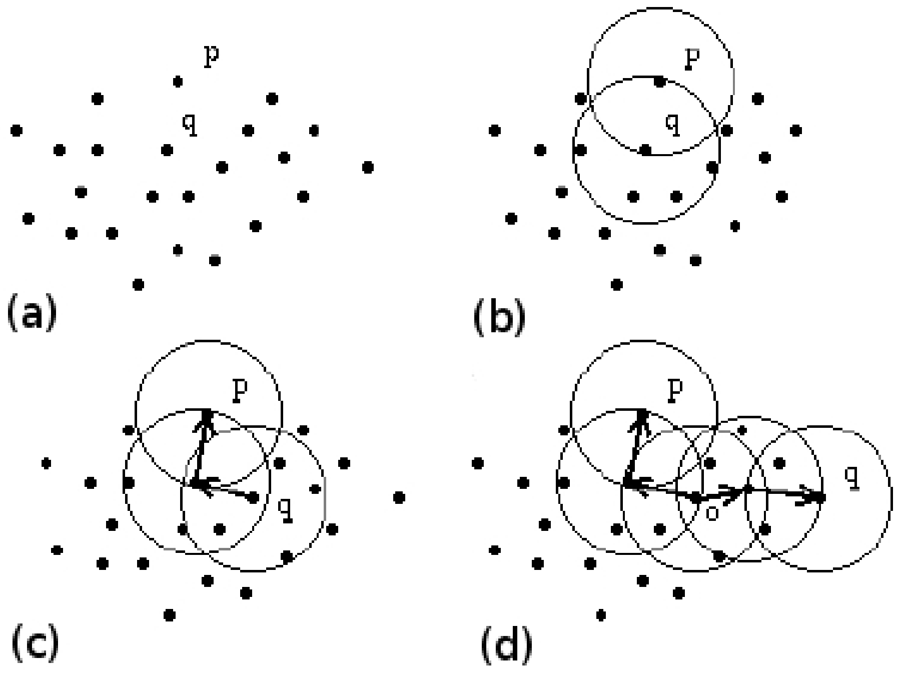

Figure-6: Density based clustering first finds a tile's nearest neighbors, then the neighbors' neighbors, and so on. (a) Data points before clustering. (b) If the density of data points “near” a point is sufficient, that point becomes a cluster seed. (c) Neighbors who then have sufficient neighbors are themselves included in the cluster. (d) This process is repeated until all points are reached that do not have sufficient neighbors. (Figure from reference [12].)

relation become one cluster. If some neighbor already belongs to a different cluster, the current cluster is merged with that cluster. Figure-6 shows building of a cluster using density based clustering algorithm in this way.

Figure-5 shows that density based clustering has clustered together the most significant part of the previously discussed injection successfully. While it loses the high-frequency end of the injection, it contains very little energy which does not significantly contribute to the duration or significance estimation of the detected trigger. Most of the energy of the injection tiles has been clustered together. Almost all the noise is removed, and we suspect the only noise cluster on the signal space to be a large detector glitch. Thus, our density based clustering function passes this initial qualitative test-case nicely.

However, as is described in the methods section, in order to maintain Q pipeline's sensitivity for localized signals without clustering while expanding it to find non-localized signals through clustering, results from Q pipeline with and without clustering are combined. This is done carefully so that the search remains sensitive to both localized and non-localized signals while avoiding double-counting any one injection. While this results in a higher false detection rate for the extended Q pipeline as compared to that of Q pipeline without clustering, this is a necessary step to ensure that extension of Q pipeline by clustering does not disregard significant triggers that it would otherwise find without clustering.

Evaluation and Implications of the Outcome

For a more formal evaluation, an extensive test program has been used. The test program loads segments of LIGO S5 detector data and runs clustering over the noise only signal space. Every "detection" on the noise-only space is considered a false detection. Then it repeatedly injects constant signal to noise ratios (SNR) signals of specific waveforms at random times with random signal parameters (bandwidth, duration, strength, mass of component stars etc.), and clusters noise-injection data over the same signal space separately. Every injection that is successfully detected after clustering is considered a successful correct detection. This data is utilized to produce receiver operating characteristic (ROC) curves for each injected signal population. ROC curves plot the false-rate of a search algorithm on the x-axis against its detection-efficiency on the y-axis as the detection threshold is varied, which in this case is a threshold on the energy of a tile or total energy of all tiles in a cluster. The resulting performance produces a characteristic curve in this two dimensional space. Detection of the lowest signal to noise ratio injection represents highest efficiency, and vice-versa.

We tested the extended Q pipeline for a total of five waveform families including two non-localized waveforms: inspiral and noise-burst; and three localized waveforms: ringdown, sinusoidal Gaussians, and Gaussians. Figures 7 and 8 show ROC curves produced for the inspiral and ringdown waveform injections. These ROC curves are produced for 200 injections at constant signal to noise ration (SNR) with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5). The red curves represent the performance of the Q pipeline without clustering and the blue curves represent the performance of the Q pipeline with clustering. The ROC's indicate that for the non-localized inspiral waveform clustering improves the performance of Q pipeline, while for the localized ringdown waveform the performance slightly deteriorates due to clustering. From the plots in Appendix-B, it is clear that for non-localized bursts, clustering improves the detection efficiency at a given threshold, while for localized bursts it remains same. In all cases, there is a slightly higher false rate which indicates that clustering is identifying higher significance background triggers that the Q pipeline without clustering would not find. By separately thresholding the clusteringa nd non-clustering triggers it is possible to recover any performance in between the two ROC curves, but such steps will require consideration of which waveforms we are most interested in.

We have shown that density based clustering is improving the performance of Q pipeline for non-localized waveforms. It is efficient at finding signals of arbitrary waveform and is good for isolating noise. Also, our implementation of the clustering algorithm is very fast and computationally efficient. Since all the testing so far has been done using single detector data, the logical next step is to test clustering for coherent and coincident searches using multiple detector data. Clustering can help us gather information about signal waveforms, their energy distribution, and many other characteristics that Q pipeline can not recognize without clustering. We recommend further research to explore these promising aspects.

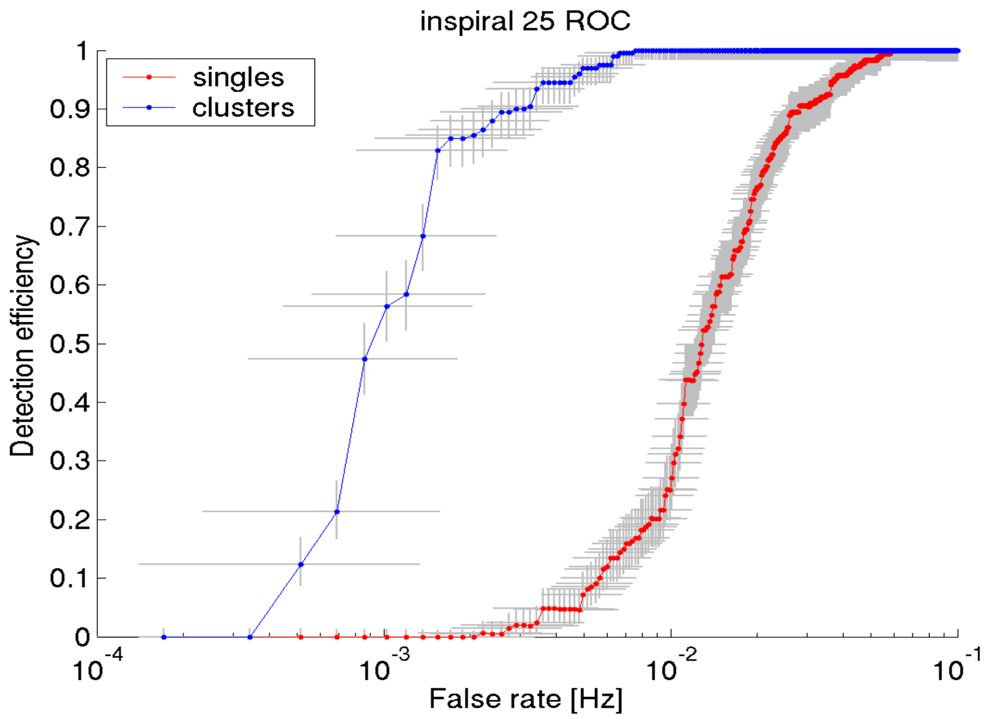

Figure-7: ROC curves for SNR of 25 (non-localized) Inspiral waveform signal.

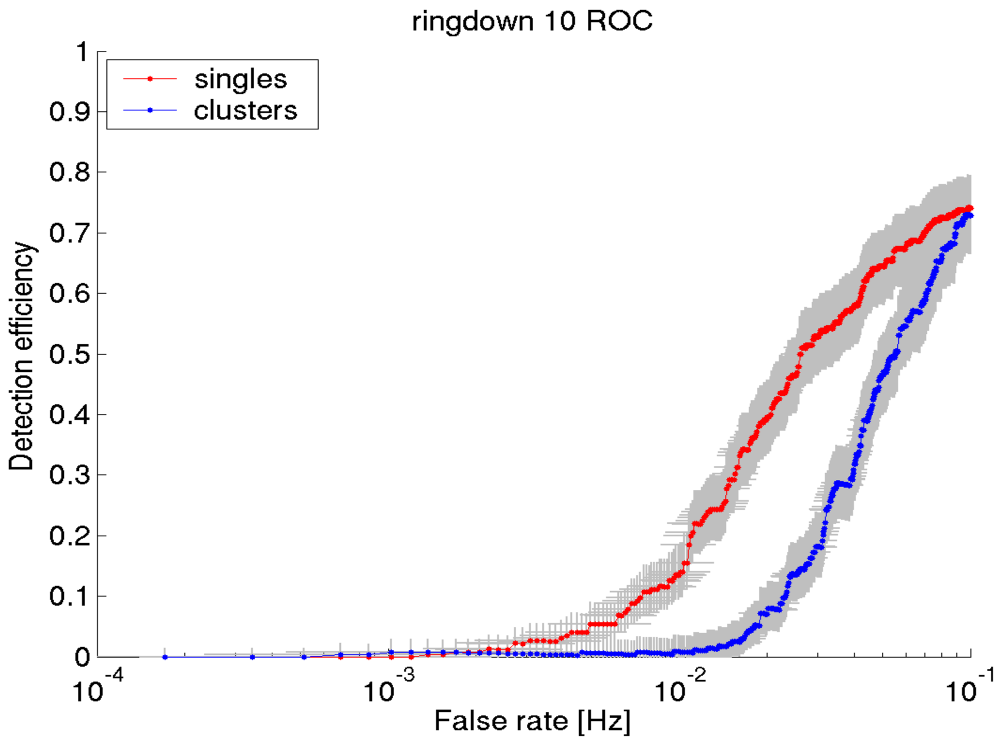

Figure-8: ROC curves for SNR of 10 (localized) Ringdown waveform signal.

METHODS

Q Pipeline. The Q pipeline is a comprehensive end-to-end analysis pipeline for the detection of gravitational-wave bursts in data from a single interferometric detector. It consists of whitening by zero-phase linear prediction, application of the discrete Q transform, thresholding on the white noise significance of Q transform coefficients, identification of the most significant set of non-overlapping time-frequency tiles, and a final stage that excludes all but the most significant time-frequency tile within a specified time window in order to prevent the redundant reporting of candidate events [7]. The analysis tool of Q pipeline is the Q-transform which is a modification of the standard short time Fourier Transform in which the analysis window duration varies inversely with frequency such that the time-frequency plane is covered by tiles of constant “Q” [7] which can be interpreted as a dimensionless quality factor for bursts which is the ratio of the center frequency to the bandwidth of the burst [6].

As the Q pipeline projects data into small regions of the time-frequency plane, an uncertainty relation applies and bursts cannot have both a well-defined frequency and a well-defined time. The minimum uncertainty signal is a "sine-Gaussian", that is a sinusoid with a Gaussian amplitude envelope: h(t) = exp(-(t - t0)^2 / (4 sigmat^2)) * sin(2 * pi * f * t). The algorithm is therefore optimal for signals that have this waveform. For signals that are less localized in the time-frequency plane, their detectability is currently determined by their maximum projection onto the space of sinusoidal Gaussians. The Q pipeline's treatment has been somewhat limited for bursts that are poorly localized in the time-frequency plane. In particular, it only considers the statistical significance of the single most significant tile with minimum time-frequency uncertainty.

In searching for statistically significant events, methods of clustering the measurements from neighboring or overlapping basis functions have been employed during this project to more optimally detect signals that are not well represented by the particular choice of basis. An improvement in the detectability of poorly localized bursts is observed. In addition, as described in [7], when evaluating the statistical significance of clusters of time-frequency tiles or testing for time-frequency coincidence between detectors, a more accurate treatment of the overlap between time-frequency tiles can be obtained by applying the mismatch formalism [8].

Distance Metric. Any clustering algorithm requires measurement of the pairwise distances between all data points, and in case of our project, the pairwise distance between all tiles. However, the tiles have varied shape which makes measurement of distance between any pair of data points rather difficult. We implemented a distance metric that takes into account these issues, and also permits inflation the distance on frequency relative to the distance in time. This latter step had to be taken in order to compensate for the fact that most non-localized signals are quite limited on the time scale while being comparatively more extended on the frequency scale. Further improvement of the distance metric can be possible. However, over the course of this project, various other metrics have been tried, and this distance function has invariably come out as the best performing one. Please see the code qdistance.m in Appendix-A for details of the distance measurement. This has been used for both the hierarchical and density based clustering.

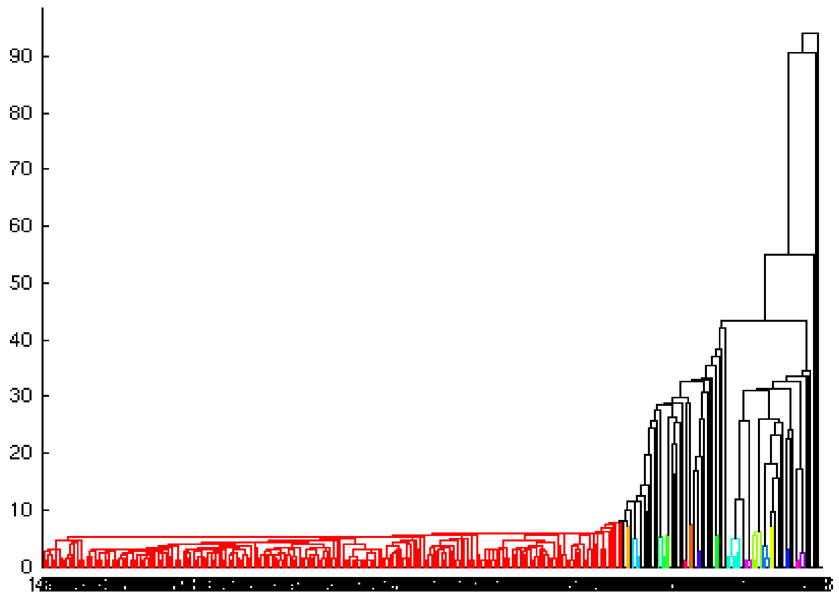

Figure-9: Dendogram for the hierarchical clustering as shown in Figure-4.

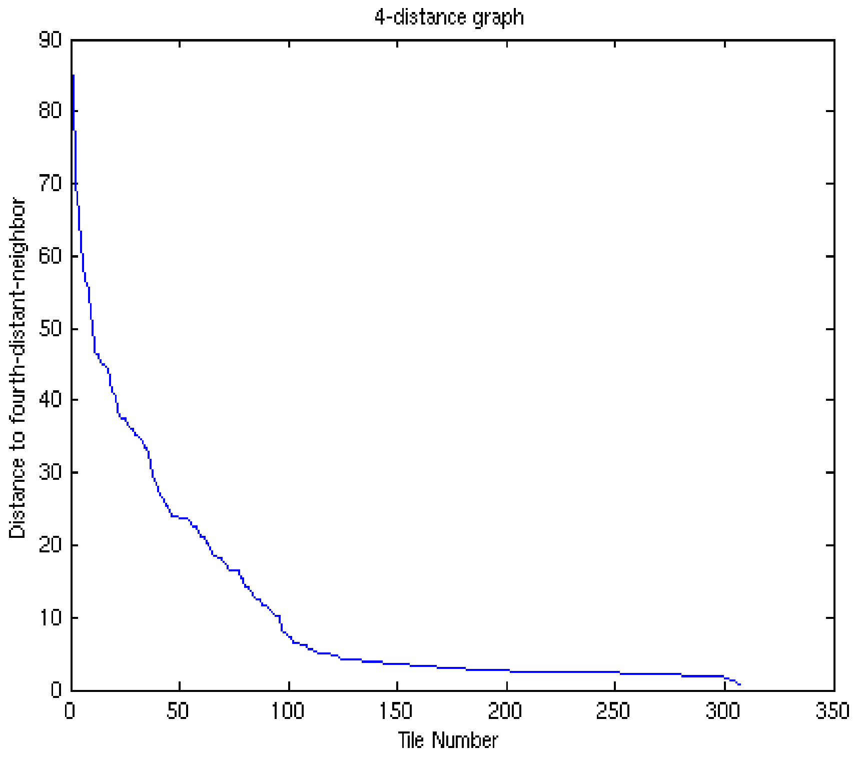

Figure-10: 4-distance graph for the density based clustering as shown in Figure-5.

Hierarchical Clustering. Hierarchical clustering first constructs smaller clusters and gradually merges them to build larger clusters. A threshold on the distance between data-points or smaller clusters is necessary for hierarchical clustering. The hierarchical clustering function that is used for the clustering as seen in Figure-4 was implemented using built in Matlab clustering functions along with the aforementioned distance metric. After measuring the distance between each pair of tiles produced by Q pipeline for a given signal space, the Matlab functions “linkage”, “dendogram”, and “cluster” were used. The resulting clustering has already been discussed, and has been shown in Figure-4. Also, Figure-9 shows the dendogram for that case. The threshold has been taken at distance 8, which corresponds to the sharp uplift of the linkage distances on the dendogram.

Density Based Clustering. As explained earlier, density based clustering finds neighbors, their neighbors, and so on on starting from a random data-point, and groups the related neighbors in one cluster. Also, it only looks for neighbors of those points that have “enough” neighbors “near” them. Our implementation of density based clustering algorithm takes two parameter: minimum neighbor number (how many is “enough”) and neighborhood radius (how far is “near”). As per the recommendation of the authors of [12] and some further testing, the minimum neighbor number of 4 has been accepted. The exact numerical value of the neighborhood radius is determined using a 4-distance graph (Figure-10) that has the distance of the fourth distant neighbor of every point along y-axis for every corresponding point on x-axis. The points are sorted according to descending order of their 4-distance value. Close observation of the 4-distance graph as well as some experimentation indicates that a neighborhood radius of 8 is a good choice for the current distance metric. Interestingly, this is also found to be the optimal clustering threshold for hierarchical clustering under similar conditions.

Avoiding the programming language specific technicalities, a simplistic yet accurate pseudo-code of the density based clustering function is given here. For details, please see the Matlab code in Appendix-A. The implementation has two major modules and a distance function. The main function calls the “expandCluster” function, and the latter recursively calls itself. Clustering starts at the most significant data-point first and then proceeds to the next significant data point that is not in a cluster, considering only such points as cluster seeds that have enough neighbors to ensure that the least number of loops are executed. If any qualifying member of the current cluster is found to be already in a cluster, the two clusters merge. Thus, regardless of at which data-point the algorithm starts clustering from, it will always find the same clusters though for speed optimization our density based clustering function picks the more significant data-points first.

Density Based Clustering Pseudo Code:

Clustering Function (Corresponding code is qcluster.m):

1. Measure distance from each data-point to each other data-point.
2. Count the number of neighbors within the given neighborhood radius around each data point.
3. Mark every data-point with at least N neighbors as potential cluster seeds.
4. Sort the potential cluster seeds according to significance (normalized energy) in

```
      descending order.
   5. Initiate clusterNumber at 0 .
   6.   i) Start loop over significance sorted potential  cluster seeds (most significant potential
      cluster seed first),
       ii) if data-point is not already in a cluster then,
       a) increment clusterNumber by 1,
       b) assign the clusterNumber to the data point,
       c) execute "expandCluster" function for the data-point,
       iii) else, go to next data-point,
       iv) end if,
       v) end loop over potential  cluster seeds.
```

expandCluster Function: (Corresponding code: qclusterb.m)

```
   1. Mark neighbors of the given data-point as members of current cluster.

   2.   i) Start loop over the neighbors,
       ii) if neighbor was originally an unused potential cluster seed then,
       execute "expandCluster" function for the neighbor (recursive call),
      iii) else if the neighbor was originally a member of another cluster,
       merge current cluster with that cluster,
      iv) else, go to next neighbor,
       v) end loop over neighbors.
```

Some mentionable aspects of this implementation of density based clustering is high speed, memory-efficiency, and overall simplicity. The dominant time consuming module of the clustering embedded extended Q pipeline is qtransform, compared to which the clustering takes nearly no time at all. Since the information about pairwise distances is not carried on in the recursion based clustering section, and instead the address of neighbors for each data-point is carried on, the recursive call does not tend to overwhelm the system memory. During running the test program jobs that takes 32 second clips at a time, density based clustering consistently consumed less memory resources than hierarchical clustering. The density based clustering algorithm does not use any information about waveforms, and and thus it finds clusters regardless of it's shape using very simple and intuitive logic.

References

[1] D. Sigg, "Gravitational Waves", LIGO-P980007-00-D, 1998.

[2] K. S. Thorne, "Gravitational Waves", arXiv:gr-qc/9506086 v1, 1995.

[3] S. A. Hughes et al. "New physics and astronomy with the new gravitational wave observatories", arXiv:astro-ph/0110349 v2, 2001.

[4] S. K. Chatterji, "Chapter 1: Introduction", emvogil-3.mit.edu/~shourov/thesis/chapter1.pdf,

2005.

[5] B. C. Barish and R. Weiss, “LIGO and the Detection of gravitational waves”, LIGO-P99039-00-R, 1999.

[6] S. K. Chatterji, “Chapter 3: Burst Detection”, emvogil-3.mit.edu/~shourov/thesis/chapter3.pdf, 2005.

[7] S. K. Chatterji, “Chapter 5: The Q Transform”, emvogil-3.mit.edu/~shourov/thesis/chapter5.pdf, 2005.

[8] S. K. Chatterji, “Chapter 8: Conclusion”, emvogil-3.mit.edu/~shourov/thesis/chapter8.pdf, 2005.

[9] J. Sylvestre, “Time-frequency detection algorithm for gravitational wave bursts”, arXiv:gr-qc/0210043, 2002.

[10] J. Sylvestre, “Upper Limits for Galactic Transient Sources of Gravitational Radiation from LIGO First Observations”, LIGO-P020007-00-R, 2002.

[11] H. Bantilan, “Graph Analysis”, virgo.physics.carleton.edu/Hans/index.html, 2005.

[12] M. Ester, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”, ifsc.ualr.edu/xwxu/publications/kdd-96.pdf.

Acknowledgments

I would thank my LIGO-SURF mentor Shourov Chatterji for his personal supervision and guidance in spite of the busy schedule of a post-doctoral researcher that he has to maintain. We have always been in close contact with each other in order to evaluate project status, carry out implementations, and of course, to brain-storm and optimally utilize the white-board space. He has been a great help as well as a source of inspiration throughout the project, allowing me to gain a pleasant academic experience while also directing me towards achieving the project goals.

I would thank LIGO deputy director Albert Lazzarini for recommending us to use density based clustering in the first place; and my mentors at Columbia University in the City of New York Szabi Marka and Luca Matone for their valuable guidance over the year preceding this project. I would also thank California Institute of Technology for organizing SURF, LIGO for providing this opportunity of gaining valuable research experience, and all other LIGO-SURF participants and mentors for all their cooperations over the summer.

Appendix-A: Matlab Codes

We implemented the density based clustering function in Matlab. The code qcluster.m is the implementation of the main density based clustering module, and qclusterb.m is the auxiliary module corresponding to the expandCluster module in the pseudo-code. qdistance.m is the distance function used for both hierarchical clustering and density based clustering.

qcluster.m

```
function clusters = qcluster(mosaics, distances, clusteringRadius, numberThreshold)

% Rubab Khan
% rmk2109@columbia.edu
%
% Shourov Chatterji
% shourov@ligo.caltech.edu

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   process command line arguments                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% verify correct number of input arguments
error(nargchk(1, 4, nargin));

if nargin < 3,
  clusteringRadius = 8;
end
if nargin < 4,
  numberThreshold = 4;
end

% if input distances are not in a cell array,
if ~iscell(distances),

  % insert distances into a single cell
  distances = mat2cell(distances, size(distances, 1), size(distances, 2));

% otherwise, continue
end

% force one dimensional cell array
distances = distances(:);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       validate command line arguments                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% determine number of channels
numberOfChannels = length(distances);

% validate significant event structures
for channelNumber = 1 : numberOfChannels,
  if ~strcmp(mosaics{channelNumber}.id, ...
             'Discrete Q-transform event structure'),
    error('input argument is not a discrete Q transform event structure');
  end
end

% validate distance structures
for channelNumber = 1 : numberOfChannels,
  if ~strcmp(distances{channelNumber}.id, ...
             'Discrete Q-transform distance structure'),
    error('input argument is not a discrete Q transform distance structure');
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        initialize results structures                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% create empty cell array of significant tile distances
clusters = cell(size(distances));

% begin loop over channels
for channelNumber = 1 : numberOfChannels

  % insert structure identification string
  clusters{channelNumber}.id = 'Discrete Q-transform cluster structure';

% end loop over channels
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          begin loop over channels                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% begin loop over channels
for channelNumber = 1 : numberOfChannels,
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 %                      intiialize clustering algorithm                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % number of tiles
  numberOfTiles = length(mosaics{channelNumber}.time);

  % convert distance structure to matrix
  distanceMatrix = squareform(distances{channelNumber}.distance);

  % initialize the tiles structure
  tiles = cell(1, numberOfTiles);

  tiles{1}.maximumNumberOfRecursions = 100;
  tiles{1}.numberThreshold = numberThreshold;

  % begin loop over tiles
  for tileNumber = 1 : numberOfTiles,

    % find indices of neighboring tiles
    tiles{tileNumber}.neighborTileNumbers = ...
        find((distanceMatrix(:, tileNumber) <= clusteringRadius) & ...
            (distanceMatrix(:, tileNumber) ~= 0));

    % number of neighboring tiles
    tiles{tileNumber}.numberOfNeighbors = ...
        length(tiles{tileNumber}.neighborTileNumbers);

    % if neighboring tiles exceed critical density,
    if tiles{tileNumber}.numberOfNeighbors >= numberThreshold,

      % identify tile as potential seed
      tiles{tileNumber}.clusterNumber = 0;

      % sort neighboring tiles by increasing normalized energy
      [ignore, sortedNeighborIndices] = ...
          sort(mosaics{channelNumber}.normalizedEnergy( ...
              tiles{tileNumber}.neighborTileNumbers));
      clear ignore;
      sortedNeighborIndices = sortedNeighborIndices(end : -1 : 1);
      tiles{tileNumber}.neighborTileNumbers = ...
          tiles{tileNumber}.neighborTileNumbers(sortedNeighborIndices);
```

```
% otherwise,
  else

    % identify as not a potential seed
    tiles{tileNumber}.clusterNumber = NaN;

  % continue
  end

 % end loop over tiles
 end

 % free distance matrix memory
 clear distanceMatrix;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          cluster significant tiles                    %
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % sort significant tiles by increasing normalized energy
 [ignore, sortedTileIndices] = sort(mosaics{channelNumber}.normalizedEnergy);
 clear ignore;
 sortedTileIndices = sortedTileIndices(end : -1 : 1);

 % initialize number of clusters
 clusterNumber = 0;

 % begin loop over sorted tiles
 for sortedTileNumber = 1 : numberOfTiles,

   % find tile number corersponding to sorted tile number
   tileNumber = sortedTileIndices(sortedTileNumber);

   % if current tile has not been processed,
   if (tiles{tileNumber}.clusterNumber == 0),

     % create a new cluster
     clusterNumber  = clusterNumber + 1;
     % assign current tile to new cluster
     tiles{tileNumber}.clusterNumber = clusterNumber;

     % find other tiles in the cluster
     [numberOfRecursions, tiles] = qclusterb(tileNumber, tiles, 3);
   % otherwise, skip to the next tile
   end

 % end loop over sorted tiles
 end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          collect results                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % collect cluster numbers
 for tileNumber = 1 : numberOfTiles
   clusters{channelNumber}.cluster(tileNumber) = ...
       tiles{tileNumber}.clusterNumber;
 end

 % identify unique cluster numbers
 clusteredTileIndices = find(~isnan(clusters{channelNumber}.cluster));
 uniqueClusters = ...
     unique(clusters{channelNumber}.cluster(clusteredTileIndices));

 % determine number of clusters
 clusters{channelNumber}.numberOfClusters = length(uniqueClusters);

 % compress cluster numbers
 for clusterNumber = 1 : clusters{channelNumber}.numberOfClusters,
   tilesInCluster = find(clusters{channelNumber}.cluster == ...
                       uniqueClusters(clusterNumber));
   clusters{channelNumber}.cluster(tilesInCluster) = clusterNumber;
 end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       end loop over channels                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% end loop over channels
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          return clusters                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% return to calling function
return
```

qclusterb.m

```
function [numberOfRecursions, tiles] = qclusterb(tileNumber, tiles, numberOfRecursions)

% Rubab Khan
% rmk2109@columbia.edu
%
% Shourov K. Chatterji
% shourov@ligo.caltech.edu

% apply default arguments
if nargin < 3
  numberOfRecursions = 3;
end

% if recursion limit is exceeded
if (numberOfRecursions >= tiles{1}.maximumNumberOfRecursions),

  % reset current tile
  tiles{tileNumber}.clusterNumber = 0;

  % return to calling function
  return;

% otherwise continue
end

% increment recursion number counter
numberOfRecursions = numberOfRecursions + 1;

% determine number of significant tiles
numberOfTiles = length(tiles);

% begin loop over neighboring tiles
for neighborNumber = 1 : tiles{tileNumber}.numberOfNeighbors,

  % determine tile number for neighbor tile
  neighborTileNumber = tiles{tileNumber}.neighborTileNumbers(neighborNumber);

  % determine current cluster number
  currentClusterNumber = tiles{tileNumber}.clusterNumber;

  % determine cluster number for neighbor tile
  oldClusterNumber = tiles{neighborTileNumber}.clusterNumber;
  % if neighbor tile has not been processed,
  if oldClusterNumber == 0

    % assign neighbor tile to current cluster
    tiles{neighborTileNumber}.clusterNumber = currentClusterNumber;

    % continue to build cluster
    [numberOfRecursions, tiles] = ...
        qcluster2b(neighborTileNumber, tiles, numberOfRecursions);

  % if neighbor tile is already in another cluster
  elseif ((oldClusterNumber > 0) && ...
          (oldClusterNumber ~= currentClusterNumber)),

      % merge current cluster into old cluster
      for testTileNumber = 1 : numberOfTiles,
        if (tiles{testTileNumber}.clusterNumber == currentClusterNumber),
          tiles{testTileNumber}.clusterNumber = oldClusterNumber;
        end
      end

      % report status
      if (tiles{neighborTileNumber}.numberOfNeighbors < ...
          tiles{1}.numberThreshold),
        mergeType = 'border tile';
      else
        mergeType = 'maximum recursion tile';
      end

  % otherwise,
  else

    % assign neighbor tile as border tile of current cluster
    tiles{neighborTileNumber}.clusterNumber = currentClusterNumber;

  % continue
  end

% end loop over neighboring tiles
end

% return to calling function
return;
```

qdistance.m

```
function distances = qdistance(significants, durationInflation, ...
                               bandwidthInflation);

% Rubab Khan
% rmk2109@columbia.edu
%
% Shourov Chatterji
% shourov@ligo.caltech.edu

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                process command line arguments                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% verify correct number of input arguments
error(nargchk(1, 3, nargin));

% default tile inflation factors
if nargin < 2,
  durationInflation = 1.0;
end
if nargin < 3,
  bandwidthInflation = 1.0;
end

% if input events are not in a cell array,
if ~iscell(significants),
  % insert significant events into a single cell
  significants = mat2cell(significants, size(significants, 1), ...
                          size(significants, 2));

% otherwise, continue
end

% force one dimensional cell array
significants = significants(:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               validate command line arguments                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% determine number of channels
numberOfChannels = length(significants);
```

```
% validate significant event structures
for channelNumber = 1 : numberOfChannels,
  if ~strcmp(significants{channelNumber}.id, ...
            'Discrete Q-transform event structure'),
    error('input argument is not a discrete Q transform event structure');
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                 initialize distances structures                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% create empty cell array of significant tile distances
distances = cell(size(significants));

% begin loop over channels
for channelNumber = 1 : numberOfChannels

  % insert structure identification string
  distances{channelNumber}.id = 'Discrete Q-transform distance structure';

% end loop over channels
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    begin loop over channels                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% begin loop over channels
for channelNumber = 1 : numberOfChannels,

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                create pairwise list of tiles                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % number of significant tiles
  numberOfSignificants = length(significants{channelNumber}.time);

  % number of unique significant tile pairs
  numberOfPairs = numberOfSignificants * (numberOfSignificants - 1) / 2;

  % build list of pairwise indices
  [pairIndices1, pairIndices2] = meshgrid(1 : numberOfSignificants);
  pairIndices1 = tril(pairIndices1, -1);
  pairIndices2 = tril(pairIndices2, -1);
  pairIndices1 = pairIndices1(:);
  pairIndices2 = pairIndices2(:);
  pairIndices1 = pairIndices1(find(pairIndices1));
  pairIndices2 = pairIndices2(find(pairIndices2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                  determine tile properties                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % extract significant tile properties
  time1 = significants{channelNumber}.time(pairIndices1);
  time2 = significants{channelNumber}.time(pairIndices2);
  frequency1 = significants{channelNumber}.frequency(pairIndices1);
  frequency2 = significants{channelNumber}.frequency(pairIndices2);
  q1 = significants{channelNumber}.q(pairIndices1);
  q2 = significants{channelNumber}.q(pairIndices2);
  normalizedEnergy1 = significants{channelNumber}.normalizedEnergy(pairIndices1);
  normalizedEnergy2 = significants{channelNumber}.normalizedEnergy(pairIndices2);

  % determine significant tile dimensions
  bandwidth1 = 2 * sqrt(pi) * frequency1 ./ q1;
  bandwidth2 = 2 * sqrt(pi) * frequency2 ./ q2;
  duration1 = 1 ./ bandwidth1;
  duration2 = 1 ./ bandwidth2;

  % apply tile inflation factors
  duration1 = duration1 * durationInflation;
  duration2 = duration2 * durationInflation;
  bandwidth1 = bandwidth1 * bandwidthInflation;
  bandwidth2 = bandwidth2 * bandwidthInflation;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                  determine tile distances                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % determine time and frequency distance scales
  timeScale = (duration1 .* normalizedEnergy1 + ...
               duration2 .* normalizedEnergy2) / ...
              (normalizedEnergy1 + normalizedEnergy2);
  frequencyScale = (bandwidth1 .* normalizedEnergy1 + ...
                    bandwidth2 .* normalizedEnergy2) / ...
              (normalizedEnergy1 + normalizedEnergy2);

  % compute normalized time and frequency distance between tiles
  timeDistance = abs(time2 - time1) / timeScale;
  frequencyDistance = abs(frequency2 - frequency1) / frequencyScale;

  % determine normalized euclidean distance between tiles
  distances{channelNumber}.distance = sqrt(timeDistance.^2 + 30 * frequencyDistance.^2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                           end loop over channels                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% end loop over channels
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                return statistically significant events                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% return to calling function
return
```

Appendix-B: ROC Curves and Other Analyses Plots

Following are the ROC curves, number of tiles vs. energy plots, false-rate vs. energy plots, and efficiency vs. energy plots for the five waveforms tested. Figures 11 through 15 show plots for Inspiral injections at SNR of 25, Noise-burst injections at SNR of 25, Ringdown injections at SNR of 10, sinusoidal Gaussian injections at SNR of 10, and Gaussian injections at SNR of 10.

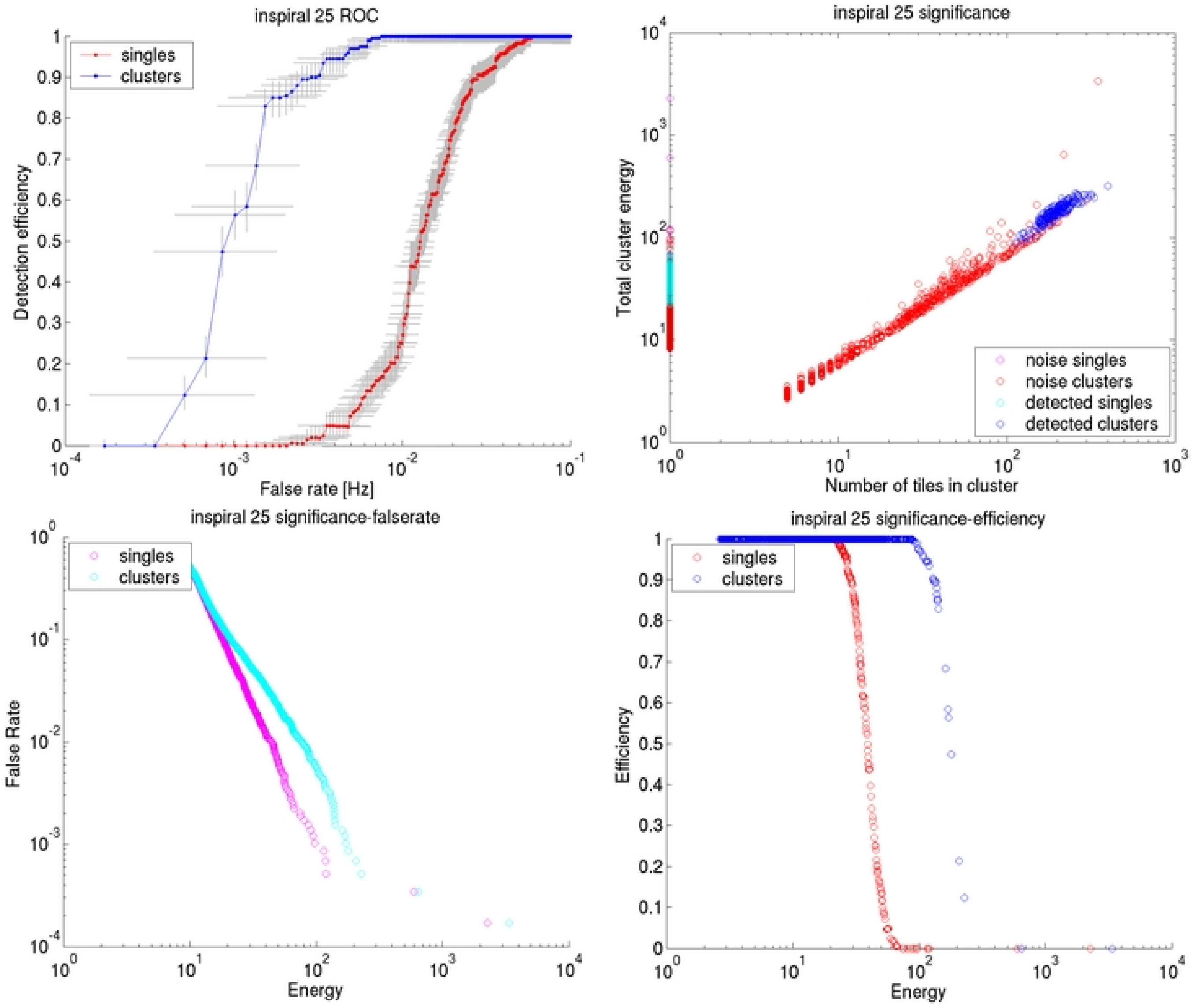

Figure-11: ROC curve, number of tiles vs. energy plot, false-rate vs. energy plot, and efficiency vs. energy plot for 200 Inspiral injections at constant signal to noise ration (SNR) of 25 with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5).

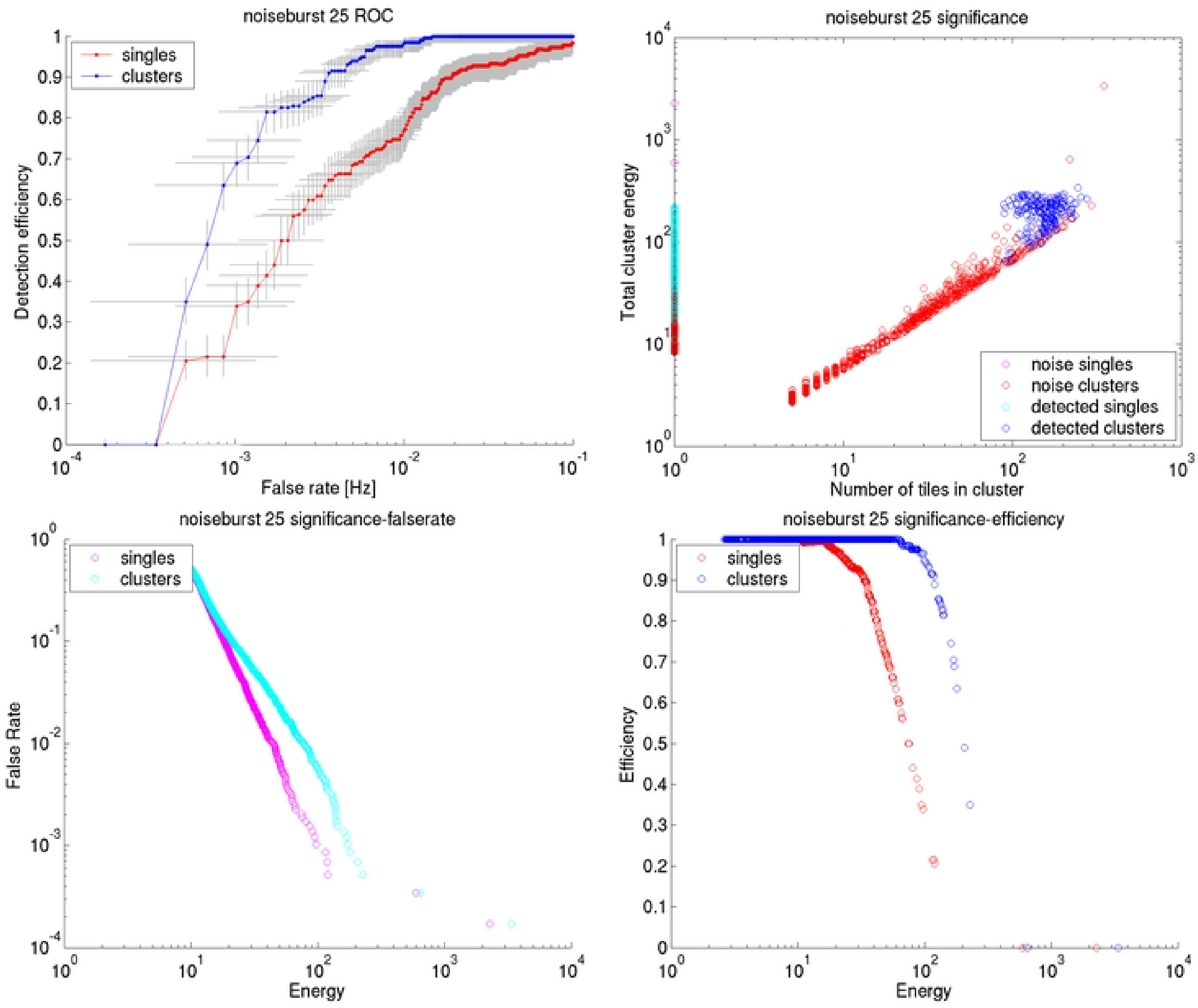

Figure-12: ROC curve, number of tiles vs. energy plot, false-rate vs. energy plot, and efficiency vs. energy plot for 200 Noise-burst injections at constant signal to noise ration (SNR) of 25 with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5).

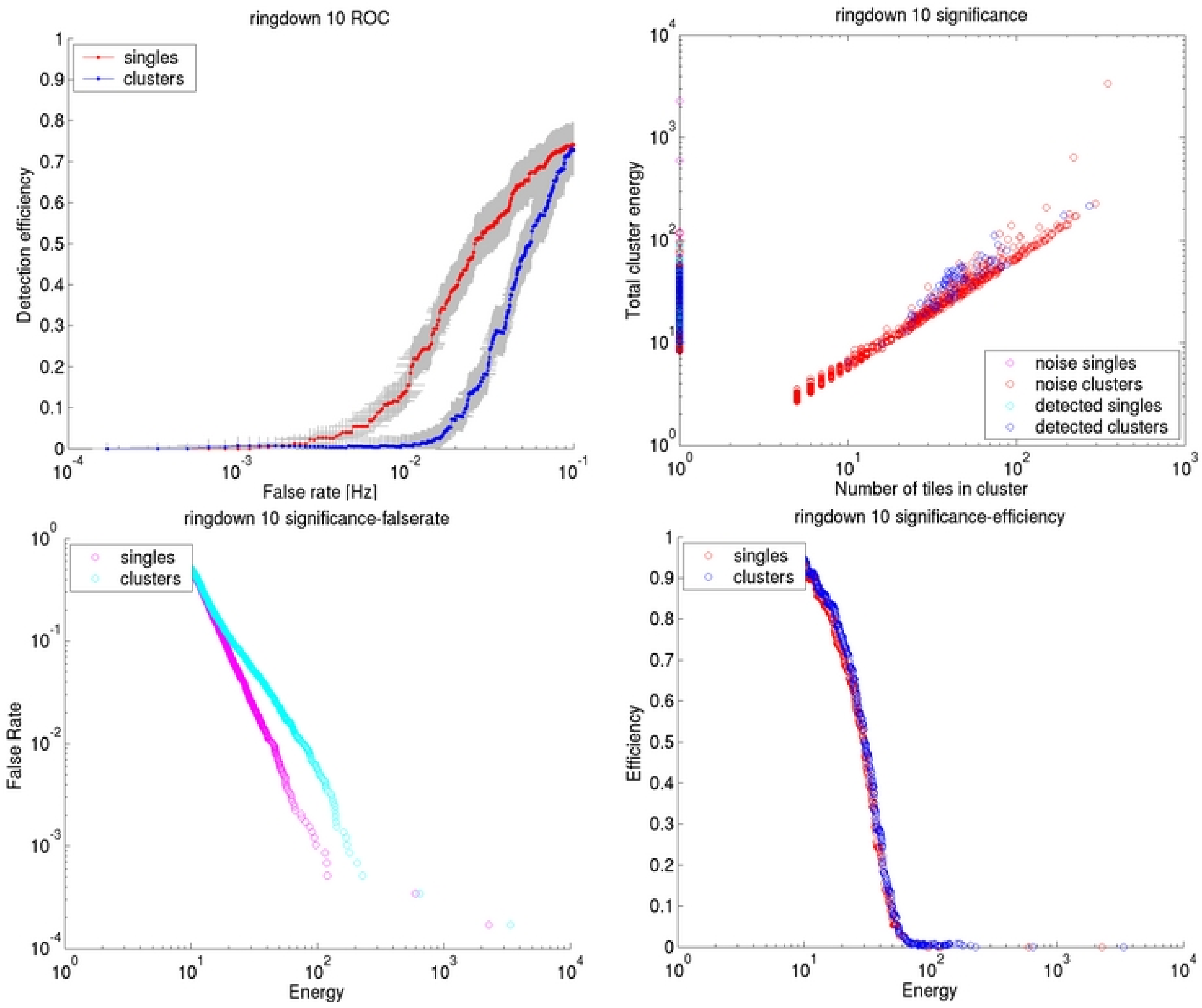

Figure-13: ROC curve, number of tiles vs. energy plot, false-rate vs. energy plot, and efficiency vs. energy plot for 200 Ringdown injections at constant signal to noise ration (SNR) of 10 with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5).

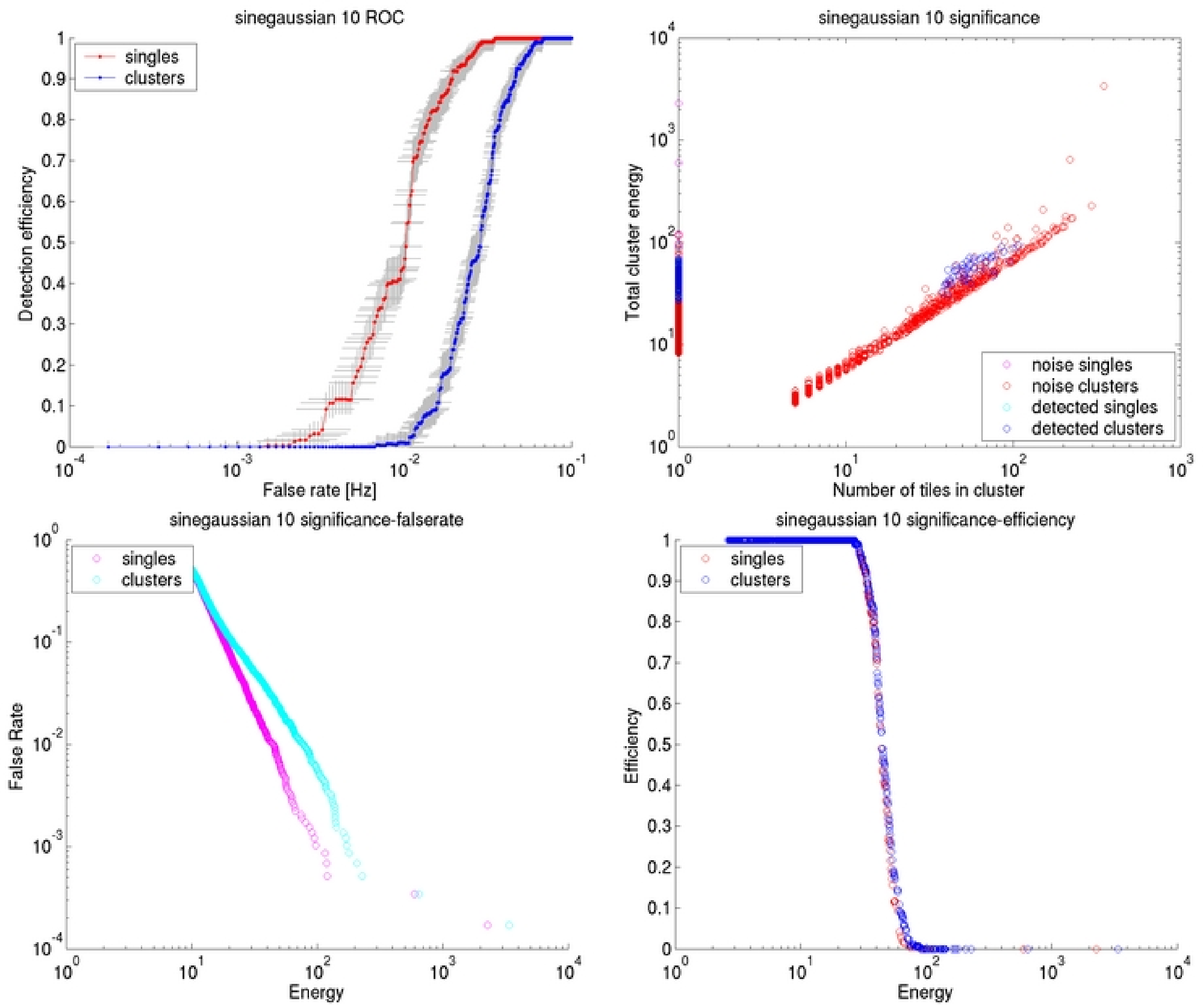

Figure-14: ROC curve, number of tiles vs. energy plot, false-rate vs. energy plot, and efficiency vs. energy plot for 200 sinusoidal Gaussian injections at constant signal to noise ration (SNR) of 10 with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5).

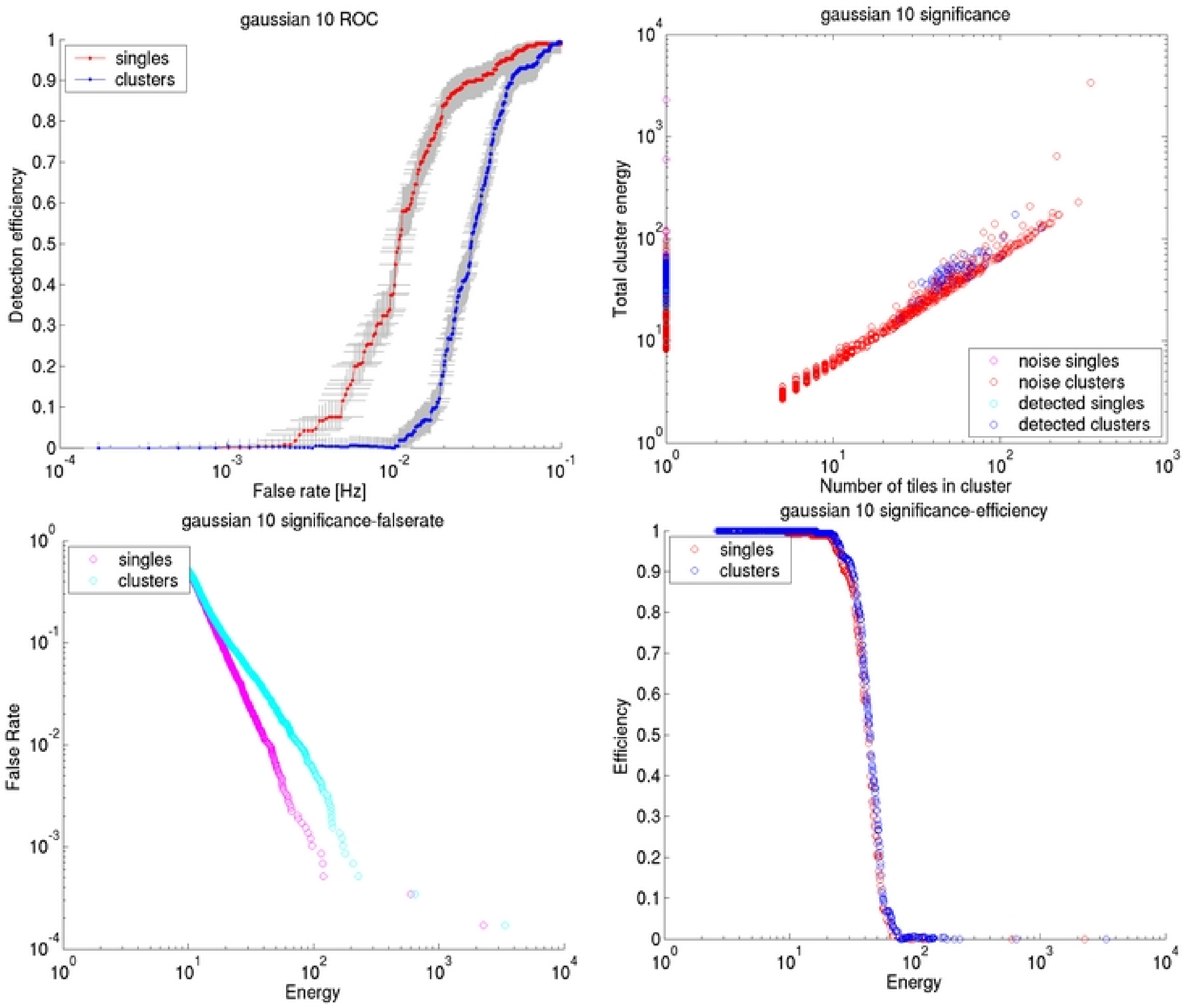

Figure-15: ROC curve, number of tiles vs. energy plot, false-rate vs. energy plot, and efficiency vs. energy plot for 200 Gaussian injections at constant signal to noise ration (SNR) of 10 with one injection per 32 second LIGO data collected during the ongoing fifth science run (S5).